Applying formal methods for software vulnerability detection & analysis

Laurent Mounier

VERIMAG / Université Grenoble-Alpes

Mai 2017

Vérimag (Université Grenoble-Alpes, CNRS, Grenoble INP)

PACSS team (Preuve et Analyse de Code pour la Sûreté et la Sécurité)

- lead by Marie-Laure Potet and David Monniaux
- main research domains :
  - Abstract interpretation and decision procedures
  - Proofs of correctness using Coq
  - Code analysis for security

Code analysis for security (2010-) :

- (binary) code analysis for security : vulnerability, exploitability
- code robustness to fault injection
- application domains :
  general purpose and/or open-source software,
  certification & common criteria,
  IoT & industrial systems (Scada)

Vérimag (Université Grenoble-Alpes, CNRS, Grenoble INP)

PACSS team (Preuve et Analyse de Code pour la Sûreté et la Sécurité)

- lead by Marie-Laure Potet and David Monniaux
- main research domains :
  - Abstract interpretation and decision procedures
  - Proofs of correctness using Coq
  - Code analysis for security

Code analysis for security (2010-) :

- (binary) code analysis for security : vulnerability, exploitability
- code robustness to fault injection
- application domains :
  general purpose and/or open-source software,
  certification & common criteria,
  IoT & industrial systems (Scada)

Context

Binsec (and other projects : Sertif, Sacade, Aramis, SecureIoT)
Collaborations :
CEA, CESTI, Tiempo, etc.
Cyber Alpes Institute
Securimag ?

Outline

Software Vulnerability Detection and Analysis

Using formal approaches : which challenges ?

Combining Static and Concolic execution for UaF detection

Lightweight runtime reverse engineering of binary code

Conclusion

Software Security

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

"correctly operate" ?

- correctness of security functionalities (crypto, access control, etc.)
- control-flow integrity (no crash, no arbitrary code execution)
- confidentiality & integrity of the (sensible) code/data

→ mostly what the SW **should not do** ...

"malicious attacks" ?

attackers = **users** + **execution plateform**, knowing :

- the target software : code + libraries, vulnerabilities, patches
- the execution environment : OS/HW architecture & protections

→ **much beyond** "unexpected input/execution conditions"

secure software ≠ robust/safe/fault-tolerant software

Software Security

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

"correctly operate" ?

- correctness of security functionalities (crypto, access control, etc.)
- control-flow integrity (no crash, no arbitrary code execution)
- confidentiality & integrity of the (sensible) code/data

→ mostly what the SW **should not do** ...

"malicious attacks" ?

attackers = **users** + **execution plateform**, knowing :

- the target software : code + libraries, vulnerabilities, patches
- the execution environment : OS/HW architecture & protections

→ **much beyond** "unexpected input/execution conditions"

secure software ≠ robust/safe/fault-tolerant software

Software Security

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

"correctly operate" ?

- correctness of security functionalities (crypto, access control, etc.)
- control-flow integrity (no crash, no arbitrary code execution)
- confidentiality & integrity of the (sensible) code/data

→ mostly what the SW **should not do** ...

"malicious attacks" ?

attackers = **users** + **execution plateform**, knowing :

- the target software : code + libraries, vulnerabilities, patches
- the execution environment : OS/HW architecture & protections

→ **much beyond** "unexpected input/execution conditions"

secure software ≠ robust/safe/fault-tolerant software

Security code analysis $\neq$ functional code analysis

Code robustness w.r.t. attacker models

- well-defined concrete attack scenarii (e.g., CVEs), attack trees, etc.
- threat quantification (attacker expertise, probability of triggering a vuln)
- formal models of attackers
- etc.

Expected outcomes :

$\rightarrow$ code security to be evaluated in its execution context ...

- qualification/quantification of the vulnerability assessment
  e.g., bug prioritization, rating the attack potentials for certification
- produce a PoC or an *exploit* :
  exploitability analysis $\rightsquigarrow$ concrete attack example
- etc.

Software vulnerabilities

A software bug that could be exploited to gain privileges on a computer system [Wikipedia]

Several vulnerability classes

- Memory safety (buffer overflow, dangling pointer, etc)
- Unsecure input handling, improper use of an API, resource leaks
- Unsecure use of the OS/HW components (side-channels), etc.

→ Low-level vs high-level vulnerabilities

In practice

- Programming Language does mater
  (type safety, undefined behaviors, reflection, etc.)
- Compiler does matter
  (memory layout, optimization level, warning flags, etc.)
- OS and HW does matter (cache hierarchy, memory management, etc.)

Looking at the defense side ?

A multi-level protection system . . .

- Programming languages
- Coding rules and patterns
- Compilers
  + contre-mesures fault injections
  + CFI
- Operating System
- Hardware

But still a major concern . . .

- 14724 CVE in 2017
  - grew 31 % compared to 2016
  - one third of them have public exploits
- Vulnerability Summary for the Week of March 27, 2017 :
  https://www.us-cert.gov/ncas/bulletins/SB17-093
- And still a lucrative market . . .
  https://zerodium.com/program.html

Existing approaches for vulnerability detection

- Dynamic Analysis
- Static analysis
- Fuzzing

Fuzzing and dynamic analysis

Dynamic analysis

source/binary code instrumentation for runtime error detection
Valgrind (DBI) and AdSan (CTI)
able to detect most memory safety errors
(stack and heap overflows, use-after-free, memory leaks, etc.)
between 2x and 20x slowdown
(almost) no false positives . . .

Fuzzing

Exploitability analysis

Peu de choses sur exploitabilite ! (ex. AFL)
reverse ? IDA Pro ... (de facto tool), etre compatible ?
qqs papiers, CGC ?

On the academic side

Code example 1 (vulnerability and exploitability)

```
int main (int argc, char *argv[])
{ char x=0 ;
  char t1[8] ;
  int i;
  for (i=0;i<=atoi(argv[2]);i++)
     t1[i]= atoi(argv[1]) ;
  if (x != 0)     printf("You win !\n") ;
  else            printf("You loose ...\n") ;
  return 0 ; }
```

Variable `x` is never modified ⇒ expected result = `You loose`?

`example1 2 7`, `exemple1 2 11`, `example1 2 17`:
`You loose ...`, `You win ...`, `non termination`

`example1 with stack protection:`
`You loose ...`, `*** stack smashing detected ***`, `*** stack smashing detected ***`

Code example 1 (vulnerability and exploitability)

```
int main (int argc, char *argv[])
{ char x=0 ;
  char t1[8] ;
  int i;
  for (i=0;i<=atoi(argv[2]);i++)
     t1[i]= atoi(argv[1]) ;
  if (x != 0)     printf("You win !\n") ;
  else            printf("You loose ...\n") ;
  return 0 ; }
```

Variable `x` is never modified ⇒ expected result = `You loose`?

`example1 2 7, exemple1 2 11, example1 2 17:`
`You loose ..., You win ..., non termination`

`example1 with stack protection:`
`You loose ..., *** stack smashing detected ***, *** stack smashing detected ***`

Code example 1 (vulnerability and exploitability)

```
int main (int argc, char *argv[])
{ char x=0 ;
  char t1[8] ;
  int i;
  for (i=0;i<=atoi(argv[2]);i++)
     t1[i]= atoi(argv[1]) ;
  if (x != 0)     printf("You win !\n") ;
  else            printf("You loose ...\n") ;
  return 0 ; }
```

Variable `x` is never modified ⇒ expected result = `You loose`?

`example1 2 7, exemple1 2 11, example1 2 17:`
`You loose ..., You win ..., non termination`

`example1 with stack protection:`
`You loose ..., *** stack smashing detected ***, *** stack smashing detected ***`

Code exemple 2 : VerifyPIN and fault injection

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size)
{
  int i; BOOL status = C_FALSE; BOOL diff = C_FALSE;
  for(i = 0;i<size;i++) if(a1[i]!= a2[i]) diff = C_TRUE;
  if(i!=size) countermeasure();
  if (diff==C_FALSE) status=C_TRUE; else status=C_FALSE;
  return status;
}

BOOL verifyPIN_5()
{
g_authenticated = C_FALSE;
if(g_ptc >= 0) { g_ptc--;
  if(byteArrayCompare(g_userPin,g_cardPin,PIN_SIZE)==C_TRUE)
    if(byteArrayCompare(g_cardPin,g_userPin,PIN_SIZE)==C_TRUE)
     {g_ptc = 3; g_authenticated = C_TRUE; return C_TRUE; }
       else countermeasure(); }
return C_FALSE;
}
```

⇒ code hardened with counter-measures

Analysing the source code is not enough . . .

Binary code, WYSINWYX (→ only the compiler ouptut does matter)

- optimisation, protections can disappear
- effects of unspecified/undefined behaviours (> 200 cases in C)
- memory layout :
  stack, heap, exception handler, method tables, etc
- whole source code is not available :
  library, close-source, obfuscated code, etc.

Combined analyses :

- from high-level to binary level, including the compiling process and execution plateform
- countermeasures can be introduced and combined at each level

Binary code analysis

→ **A preliminary step :** understanding 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 . . .

But undecidable in general ! (key issue = distinguishing code from data)

→ **Next steps :** static and or dynamic assembly code analysis . . .

Static code analysis (abstract interpretation)

$\rightarrow$ abstract semantics to (over-) approximate the code behaviour

Pros :

- correctness w.r.t. disassembled code $\rightsquigarrow$ no false negatives
- scalability

Cons :

- how to take into account libraries, interactions with the OS, ...
- difficulty to produce a PoC or an exploit

Difficulties inherent to assembly code :

- data structure & CFG recovery (function frames, call/return, etc.)
- adapted memory models, uninitialized values (`esp`, `ebp` ...)

Tools : CodeSonar, Veracode, Binsec, Bap, etc.

Dynamic and Concolic Execution (white-box fuzzing)

$\rightarrow$ code execution + code instrumentation + symbolic reasonning

Pros :

- "only" requires execution and instrumentation facilities
- can exhibit vulnerable executions

Cons :

- execution time overhead
- incompleteness $\rightsquigarrow$ false negatives
  (input generation to be driven by a guiding strategy or coverage criteria)

Security application :

- fuzzing $\rightsquigarrow$ crashes $\rightsquigarrow$ derive some exploits ?
- combination with dynamic checkers (AdSan, Valgrind)

Tools : AFL (fuzzer), SAGE, S2E, AngR, Triton (DSE)...

Code analysis for security at Vérimag

Software vulnerability detection and analysis

- Combination of static and dynamic analyses
- Combination of high-level and low level analyses
- Counter-measures analysis and attack models

2 main applications :

- Software vulnerability detection and analysis
  (ANR Binsec 2013-2017, Josselin Feist's thesis)
- Robustness evaluation against fault injections
  (ASTRID Sertif 2014-2017, Louis Dureuil's thesis)

Outline

Outline

Use after Free example

```
1  p=malloc(sizeof(int));
2  p_alias=p;                // p and p_alias points
3                            // to the same addr
4  read(f,buf,255);         // buf is tainted
5
6  if(strncmp(buf,"BAD\n",4)==0)
7   { free(p);               // exit() is missing
8   }
9  else{ ..          // some computation
10     }
11
12 if(strncmp(&buf[4],"is a uaf\n",9)==0)
13     { p=malloc(sizeof(int));  }
14 else{ p=malloc(sizeof(int));
15        p_alias=p;  }
16
17 *p=42 ;                  // not a uaf
18 *p_alias=43 ;          // uaf if 6 and 14 = true
```

Use after Free example

```
1  p=malloc(sizeof(int));
2  p_alias=p;                 // p and p_alias points
3                             // to the same addr
4  read(f,buf,255);           // buf is tainted
5
6  if(strncmp(buf,"BAD\n",4)==0)
```

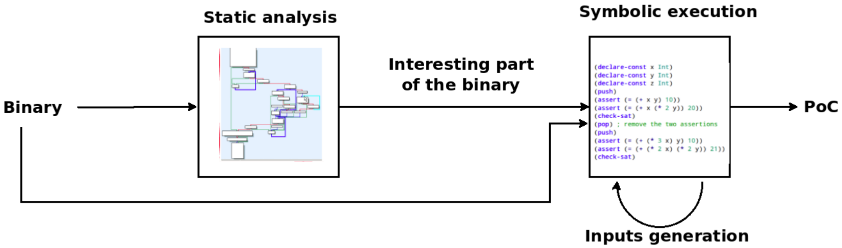
- Difficult to detect (distant events, reasoning with heap, ..)
- No easy "pattern" (like for buffer overflow / string format)
- Lots of *Use-After-Free* in browsers and in other apps (proftpd CVE-2011-4130, privoxy CVE-2015-1031, openssh...)

```
12 if(strncmp(&buf[4],"is a uaf\n",9)==0)
13     { p=malloc(sizeof(int));  }
14 else{ p=malloc(sizeof(int));
15       p_alias=p;  }
16
17 *p=42 ;                    // not a uaf
18 *p_alias=43 ;              // uaf if 6 and 14 = true
```

Josselin Feist's thesis approach

Combining adequately static and DSE analyses

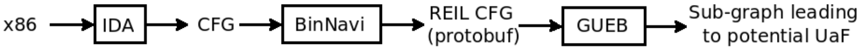
- **Static analysis** to extract potential vulnerable paths
- **Dynamic Symbolic Execution** to confirm Use-after-Free
- application to real codes (binary pbs + scalability)



Static analyzer : GUEB

Static analysis features

- dangerous path discovery : pointer and aliases, inter-procedural
- Use-After-Free characterization : 2 heap models



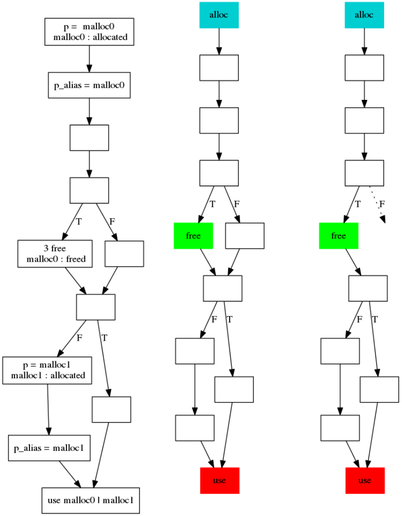
Scalability features

- some (unsound) heuristics : loop unrolling and inlining, ...
- a very separated memory model taking into account uninitialized memory (`ebp`, `esp`, ...)

(Ocaml) Open source : https://github.com/montyly/gueb

Resulting slice on Example

```
1   p=malloc(sizeof(int));
2   p_alias=p; // p and p_alias points
3          // to the same addr
4   read(f,buf,255); // buf is tainted
5
6   if(strncmp(buf,"BAD\n",4)==0){
7    free(p);
8    // exit() is missing
9   }
10  else{
11   .. // some computation
12  }
13
14  if(strncmp(&buf[4],"is a uaf\n",9)
        ==0){
15    p=malloc(sizeof(int));
16  }
17  else{
18    p=malloc(sizeof(int));
19    p_alias=p;
20  }
21
22  *p=42 ; // not a uaf
23  *p_alias=43 ; // uaf if 6 and 14 =
        true
```



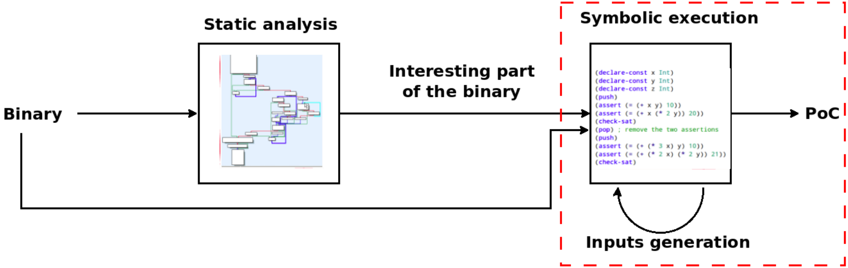
GUEB : Experimentations

Results

- Several experiments : UaF detection accuracy (no real existing benchmark), applicability to real applications (below) , scalability (400 binaries).
- Several new *Use-After-Free* and referenced CVE found (CVE-2015-5221, CVE-2015-8871, CVE-2016-3177)

name	#REIL ins	time	#UAF	#EP	max size reached
alsabat	99 933	7s	1	10	0
gnome-nettool (-OO)	226 514	16s	4	56	0
gnome-nettool	260 882	17s	7	76	0
gifcolor *	233 303	21s	15	13	0
jasper *	2 154 927	4m23s	255	205	5
accel-ppd	3 907 862	5m5s	35	299	0
openjpeg *	2 170 081	6m10s	329	305	12

Dynamic Symbolic Execution



DSE Features

Exploration strategy

- Guided by slices and distance metrics
- C/S policies (ISSTA 2016)

A condition to determine real UaF

- reinforcing the path predicate Π with the set of constraints :

$$a_f = a_m \wedge \; a_u \in [a_m, a_m + size_{alloc} - 1] \qquad (1)$$

- Data-dependency between a_f and a_m and between a_u and a_f (no symbolic value for a_m violating (1)).

An iterative process to discover a reachable initial state

- Obtain a model $m = (i, s)$ from Π, extract constraints C on s from $P(i)$ and resolve $\Pi \wedge C$ and so on ...

Implementation in the BinSec platform + XP

BINSEC/SE

- based on the BinSec open source platform offering semantic binary level analyses : disassembly, simulation, symbolic execution, static analysis
- Our DSE : selection strategies, guiding modules and heuristics
- http://binsec.gforge.inria.fr/tools

Jasper (Jasper-JPEG-200 CVE-2015-5221)

- 20 mins
- 9 test cases generated, one triggering the *Use-After-Free*
- PoC :

MIF
component

Experimental validation of our approach

Name	Time	MIF line	UAF found	# Paths
DSE (in BINSEC/SE)				
WS-Guided+*LDH*	20*m*	3*min*	**Yes**	9
WS-Guided	6*h*	3*min*	*No*	44
DFS(*slice*)	6*h*	3*min*	*No*	68
DFS	6*h*	3*min*	*No*	354
standard fuzzers (arbitrary seed)				
AFL	7*h*	$< 1min$	*No*	174†
Radamsa	7*h*	$> 1h$	*No*	$\sim$ 1000000‡
standard fuzzers (MIF seed)				
AFL (*MIF input*)	$< 1min$	$< 1min$	**Yes**	< 10
Radamsa (*MIF input*)	$< 1min$	$< 1min$	**Yes**	< 10

† *AFL generates more input, 174 is the number of unique paths.*
‡ *For radamsa it is not trivial to count the number of unique path.*

Table: `JasPer` evaluation

Combination is fruitful

$\Rightarrow$ An end-to-end approach for vulnerability detection,
with scalability and binary code concerns.

Binary level static analysis

- dangerous/vulnerable *path discovery* concerns
  $\rightsquigarrow$ a set of **CFG slices** to explore (possibly incomplete)
- scalability, but with some unsoud heuristics

Dynamic Symbolic Execution

- guided trace exploration towards given CFG slices
- C/S policies
- dedicated heuristics to speed up the exploration

Outline

Software Vulnerability Detection and Analysis

Using formal approaches : which challenges ?

Combining Static and Concolic execution for UaF detection

Lightweight runtime reverse engineering of binary code

Conclusion

Conclusion

Code analysis for security

- reasoning on binary code, with :
  - an attacker model
  - beyond normal executions (after a crash, on a corrupted code)

  $\rightarrow$ needs to consider non standard semantics ...
- more than “bug finding”
  - exploit generation, qualitative and quantitative assessment
  - counter-measure analysis (accuracy and efficiency)

Adaptable tools suit

- from high level to low level analyses
- adaptable certification process (TEE, IoT, ...)

Security in Grenoble-Alpes community

SCCyPhy :

Structure the research & education community in computer security and cryptography :

- Design and Analysis of Cryptographic Components (DACC)
- Code Protection and Analysis (CAP)
- Security and Privacy for Pervasive Systems (SPPS)

Members : Université Grenoble-Alpes, Grenoble INP, INRIA, CEA

Next events :