REDOCS2020 - Automatic exploit generation

Maxime Bélair ¹  Manh-Dung Nguyen ²  Emilien Fournier ³
Tristan Benoit ⁴  Gabriel Sauger ⁴

Subject by: Jules Villard

¹Orange Labs / IMT atlantique - maxime.belair@imt-atlantique.fr
²CEA LIST & Université Grenoble Alpes - manh-dung.nguyen@cea.fr
³ENSTA Bretagne / Lab-STICC - emilien.fournier@ensta-bretagne.org
⁴LORIA - firstName.lastName@loria.fr
Problem Overview

Context

- Codebases are bigger than ever
- More bugs than ever!
Problem Overview

Context

- Codebases are bigger than ever
- More bugs than ever!

Les développeurs gèrent un volume de code 100 fois plus important maintenant qu'en 2010 dans plus de langages, pour plus de plateformes que jamais.

Une complexité qui a un impact personnel sur eux

Le 1er octobre 2020 à 17:41, par Stéphane le calme | 32 commentaires
Problem Overview

Context

- Codebases are bigger than ever
- More bugs than ever!

Facebook runs on at least 64 millions of line of code. Imagine the debugging.
Background

Static Analysis

- Processes the code of the program
- Don’t execute it!
Problem overview

Formal challenge

"Can we automatically turn static analysis reports into executable confirming the vulnerability of a program?" - Jules Villard, Facebook
Weakness example: the Buffer Overflow

- Classic attack
- Allow to modify other variables in memory
- A lot of consequences
  - Crash the program (e.g. Segfault)
  - Hijack the execution (e.g. Cracks)
  - Privilege escalation (e.g. Root Shells),
  - ...

Problem overview

Context
The Buffer Overflow – Technically

```c
strcpy(A, "excessive")
```
The Buffer Overflow – Technically

\[ \text{strcpy}(A, \ "excessive") \]
The Buffer Overflow – Technically

\[ \text{strcpy}(A, \text{"excessive"}) \]
Weakness example: the Buffer Overflow

```c
int main() {
    short valid = 0;
    char buf[8];
    strcpy(buf, "excessive");
    printf("%d", valid); // Overflow !!!
}
```
Program bug example

- Example of vulnerability – 101 Buffer overflow
- Exploit it manually
- Automatize it through Infer later!
Infer tool

C / Objective C
C++ / Java

INFER

List of potential bugs
- Static analysis tool from Facebook
- **Capture** phase, then **Analysis** phase
Practical approach

Program
Practical approach

Program

Find a bug in program
Practical approach

Program

Find a bug in program

Find inputs to reach it
Practical approach

1. Program
2. Find a bug in program
3. Find inputs to reach it
4. Create exploit!
This week’s challenge

Given the Infer information about bugs of a program A, find a set of inputs that crashes A
Table of content

1 Problem overview
   - Context
   - Program bug example
   - Infer tool
   - Practical approach

2 Proposed approaches
   - Model checking
   - Symbolic approach : SMT
   - Fuzzing technique

3 Conclusions and perspectives
   - Results comparison
   - Future Work
Proposed approaches
Model checking

Overview

- Formal
- Intuitive
- Automated
- Provides counter-examples
- State-space explosion
Obelix : Modular Model-Checker

Highlights

- Modular model-checker
- 3 languages : DVE, EMI-UML, C/C++
- 12+ algorithms
- Bitstate hashing
- Swarm model-checking
Obelix: Generic Language Interface

Proposed approaches
- Model checking
Obelix : Generic Language Interface

```latex
structure \mathcal{M} (C : Type) :=
  (initial : set C)
  (next : C \rightarrow set C)
  (is_safe : C \rightarrow bool)
```
Application: How?
Application: How?
A few results

Results

- Custom model-checker
- 16 reachability algorithms supported
Symbolic approach: SMT

Find output using a logic formula
CFG and BFS

Proposed approaches

Symbolic approach: SMT
Proposed approaches

Symbolic approach: SMT

**Compiler**

\[
437 + 734 \xrightarrow{\text{LEXER}} \begin{array}{c}
\text{NUM} \\
437
\end{array} + \begin{array}{c}
\text{NUM} \\
734
\end{array} \xrightarrow{\text{PARSER}} \begin{array}{c}
\text{expr} \\
437 + 734
\end{array}
\]

abstract syntax tree
Proposed approaches

Symbolic approach: SMT

Smallfoot Intermediate Language

41: DeclStmt
VARIABLE_DECLARED(isValid:unsigned char); [line 26, column 5]
*isValid:unsigned char=(unsigned char)0 [line 26, column 5]

40: DeclStmt
VARIABLE_DECLARED(f:int); [line 28, column 5]
n$33=*argv:char** [line 28, column 18]
n$34=*n$33[1]:char* [line 28, column 18]
n$35=_fun_open(n$34:char*,0:int) [line 28, column 13]
*&f:int=n$35 [line 28, column 5]

39: Call_func_read
n$31=*&f:int [line 29, column 10]
n$32=_fun_read(n$31:int,&pwd:void*,(unsigned long)32:unsigned long) [line 29, column 5]
Proposed approaches

Symbolic approach : SMT

```
1 ; This example illustrates basic arithmetic and
2 ; uninterpreted functions
3
4 (declare-fun x () Int)
5 (declare-fun y () Int)
6 (declare-fun z () Int)
7 (assert (>= (* 2 x) (+ y z)))
8 (declare-fun f (Int) Int)
9 (declare-fun g (Int Int) Int)
10 (assert (< (f x) (g x x)))
11 (assert (> (f y) (g x x)))
12 (check-sat)
13 (get-model)
```
Proposed approaches

Symbolic approach: SMT

Input Language

Restriction:

- No pointer
- Only int
- No exterior function
- Only use argv as a vector of int
Buffer Overflow example

Does not fit our example:

- Array
- char* (String)
- Standard functions: memcpy
Proposed approaches

Symbolic approach: SMT

Compiler

Does not fit our example:

- Array
- char* (String)
- Standard functions: memcpy

Principle

*What we won’t understand won’t hurt us.*
Symbolic approach : SMT

SMT Output

(assert (= isvalid_0 0))
(assert (= n33_0 argv_0))
(assert (= n34_0 argv_0))
(assert (= n35_0 _fun_open_0))
(assert (= f_0 n35_0))
(assert (= n31_0 f_0))
(assert (= n32_0 _fun_read_1))
(assert (= n26_0 argc_0))
(assert (not (<= n26_0 2)))
(assert (= n2_0 argv_0))
(assert (= n2_0 73))
(assert (= n3_0 argv_0))
(assert (= n3_0 110))
(assert (= n4_0 argv_0))

(assert (= n5_0 argv_A_4))
(assert (= n5_0 101))
(assert (= n6_0 argv_A_5))
(assert (= n6_0 114))
(assert (= n14_0 _fun_strlen_2))
(assert (= n15_0 _fun_checkpwd_3))
(assert (= isvalid_1 n15_0))
(assert (= n11_0 argv_0))
(assert (= n12_0 argv_A_6))
(assert (= n13_0 _fun_memcpy_4))
(assert (= n7_0 isvalid_1))
(assert (= n7_0 1))
(assert (= n8_0 _fun_valid_5))
(assert (= n4_0 102))
Proposed approaches

Symbolic approach: SMT

- CFG
  - BFS
  - Path to bug
  - Compiler

- STM-LIB2
  - SMT Solver
- Fun. calls
- Inputs access

Data recombination

Input to reach the bug!
Proposed approaches

Symbolic approach: SMT

SMT Solver Response

```plaintext
{
    'argc_0': 3, '_fun_valid_5': 0, 'n8_0': 0, 'n7_0': 1,
    'isvalid_1': 1, '_fun_memcpy_4': 0, 'n13_0': 0,
    'argv_A_6': 0, 'n12_0': 0, 'argv_0': 0, 'n11_0': 0,
    'n15_0': 1, '_fun_checkpwd_3': 1, '_fun_strlen_2': 0,
    'n14_0': 0, 'n6_0': 114, 'argv_A_5': 114, 'n5_0': 101,
    'argv_A_4': 101, 'n4_0': 102, 'argv_A_3': 102,
    'n3_0': 110, 'argv_A_2': 110, 'n2_0': 73,
    'argv_A_1': 73, 'n26_0': 3, '_fun_read_1': 0,
    'n32_0': 0, '_fun_open_0': 0, 'n31_0': 0, 'f_0': 0,
    'n35_0': 0, 'argv_A_0': 0, 'n34_0': 0,
    'n33_0': 0, 'isvalid_0': 0, 'pwd_0': 0
}
```
REDOCS2020 - Automatic exploit generation

Proposed approaches

Symbolic approach: SMT

Evaluation

```
1  Function calls
2    Call NUMBER 5  :  valid()  ==  0
3 Call NUMBER 4  :  memcpy(cmd,n12,45)  ==  0
4 Call NUMBER 3  :  checkpwd(pwd,n14)  ==  1
5 Call NUMBER 2  :  strlen(0)  ==  0
6 Call NUMBER 1  :  read(n31,pwd,32)  ==  0
7 Call NUMBER 0  :  open(n34,0)  ==  0

8 Input
9    argc  ==  3
10   argv_A_0
11   argv[1]  ==  0
12   argv_A_1
13  argv[0]  ==  73
14  argv_A_2
15   argv[1]  ==  110
16  argv_A_3
17   argv[2]  ==  102
18  argv_A_4
19  argv[3]  ==  101
20  argv_A_5
21   argv[4]  ==  114
22
23 COMMAND GENERATED
24 Infer
```

\[ \approx 0.30 \text{ seconds} \]
Fuzzing Technique

Third approach: Directed Fuzzing
Fuzzing technique

**Idea:** Generate random inputs in the hope of crashing programs

> 100 fuzzers (Google, Facebook, Microsoft) in recent years

Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale
Coverage-guided greybox fuzzing [AFL, libFuzzer]

Goal: Cover as many paths as possible

- Feedback: code coverage (e.g., lines, branches)
- Mutation operators: bitflip, insert/delete/overwrite bytes ...
Intuitions of directed approach

- **Goal**: Reach predefined targets (e.g., recent code changes, vulnerable functions, patches, static reports)
- New distance-based input metric
- Favor inputs that are "closer" to targets
  - Select them more often
  - Produce more mutants from those inputs
Directed greybox fuzzing [AFLGo]

"Directed Greybox Fuzzing", M. Böhme et al, CCS’17
Putting it all together

Program
Putting it all together

Program → ✗ → Bug trace

main.c : line 24
main.c : line 12
main.c : line 28
Putting it all together

Program → Bug trace

main.c : line 24
main.c : line 12
main.c : line 28

AFLGo
(Directed Greybox Fuzzing)
Putting it all together

Program → Bug trace → Tested program
  main.c : line 24
  main.c : line 12
  main.c : line 28

AFLGo
(Directed Greybox Fuzzing)

Seed input
Putting it all together

Program → Bug trace

main.c : line 24
main.c : line 12
main.c : line 28

Tested program

Seed input

AFLGo
(Directed Greybox Fuzzing)

Inputs to reach the bug!
Implementation

- Python script to parse Infer's output "report.json"
  - Targets: ['b0f_infer.c:24', 'b0f_infer.c:37']
- Generate a script to run AFLGo https://github.com/aflgo/aflgo

```json
{
    "bug_type": "BUFFER_OVERFLOW_LI",
    "qualifier": "Offset added: 45 Size: 32.",
    "severity": "ERROR",
    "line": 37,
    "column": 25,
    "procedure": "main",
    "procedure_start_line": 24,
    "file": "b0f_infer.c",
    "bug_trace": [
        {
            "level": 0,
            "filename": "b0f_infer.c",
            "line_number": 24,
            "column_number": 1,
            "description": "<Length trace>"
        },
        {
            "level": 0,
            "filename": "b0f_infer.c",
            "line_number": 24,
            "column_number": 1,
            "description": "Array declaration"
        },
        {
            "level": 0,
            "filename": "b0f_infer.c",
            "line_number": 37,
            "column_number": 25,
            "description": "Array access: Offset added: 45 Size: 32"
        }
    ],
}
```
Evaluation

Run the fuzzer 5 times against the buffer overflow example (due to randomness)

<table>
<thead>
<tr>
<th>Run</th>
<th>Time-to-Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4m 44s</td>
</tr>
<tr>
<td>2</td>
<td>18m 4s</td>
</tr>
<tr>
<td>3</td>
<td>17m 11s</td>
</tr>
<tr>
<td>4</td>
<td>14m 50s</td>
</tr>
<tr>
<td>5</td>
<td>18m 13s</td>
</tr>
<tr>
<td>Average</td>
<td>14 m 36 s</td>
</tr>
</tbody>
</table>
Conclusions and perspectives
Results comparison

SMT

+ Quick
+ Function complexity
- Semantic limitations

Fuzzing

+ Can be efficient
- Does not always find the solution

Model Checking

+ Respect the semantic
- Combinatorial explosion
Other Approaches – Hybrid methods

- Hybrid methods
  - Prune interesting branches with Fuzzing + Heuristics
  - SMT on subproblems
Future Work – Automatic exploit generation

- We can trigger bugs
- How to automatize their exploits?
- Create an attack model
- Find all possible exploits
- → FULLY AUTOMATED EXPLOITS!
void onBufferOverflow(env, bufsz, entrypoint)
    if (canOverWriteRip())
        addExploit("Can rewrite on rip!");
    for (var = 0 to env.nb_vars)
        if (var.rewritable && var.used)
            addExploit("Can Hijack the program!");
/* [... ] */
Next steps to improve the framework

- **General**
  - Test on code with several path to the same bug

- **Fuzzing**
  - Resolve magic-bytes comparisons, (e.g. `strcmp`)
  - Distance computation is long with large programs

- **SMT**
  - Add more types
  - Support arbitrary memory accesses
  - Improve function with hooks

- **Model Checker**
  - Partial order reduction
  - Hardware model-checking
It works on my machine
Thank you!