Analyzing Binaries Using angr

Jason Jeremy Iman 13514058
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13514058@std.stei.itb.ac.id

***Abstract*—Binaries are commonly found within the field of Informatics. However, analyzing and understanding one can be difficult. In order to simplifies and automates the analysis of a binary, angr, a binary analysis framework, can be used. angr provides a semi-automatic concrete and symbolic approach of analyzing a binary.**

Keywords—binary; angr; analysis;

I. INTRODUCTION

Binaries which often referenced as executables are files that causes computer to perform a set of instructions. These files contain traditional machine code or bytecode that is directly understandable by computers as opposed to other files which require parsing to be meaningful. There are many types of computer architecture but most of them have machine code that is written with a language called assembly. However, difference architecture does have multiple differences of instruction sets and registers of the language.

Binaries are found everywhere within a machine. Operating systems, command-line, and even softwares consist of binaries. However, as much as we use binaries, understanding a binary can be difficult. Most programs and softwares are coded using a higher-level language in which even the coder can not understand the binary. While in some cases analyzing a binary is not a necessities as proofreading the source code is considered enough, there are cases of missing source code and bugs within binaries. As such, it is very important to be able to analyze a binary.

There are two ways of analyzing a binary, statically by reading the code and dynamically by observing the execution of the binary. These techniques are often combined in order to fully understand a binary. However, most of the time, those steps are done manually. While it is possible for humans to analyze a binary, it would not be as consistent as machines. In order to do that, students from computer lab of UC Berkeley created a framework called angr that automatically analyze a binary statically and dynamically.

angr is capable of analyzing a binary through concrete and symbolic execution in which each variable is treated as symbolic variable along a concrete execution path. angr can also create a control-flow graph in order to understand the binary better. Those capabilities of angr combined with static analysis allows angr to find flaws, vulnerabilities, and specific exeution paths within a binary.

II. THEORY

A. Assembly

Assembly often abbreviated asm, is a low-level programming language. As it directly communicate with hardwares, this language varies with computer architecture. Most of the variations, if not all, follow the same rules which are

1. use architectural registers to store data,
2. use flags to indicate result of computations,
3. have segments that indicate the type of data stored,
4. written with binary code,
5. each binary code can be interpreted as instructions by the computer architecture.

These are some instructions of assembly language of x86/IA-32 architecture

1. mov var1 var2, move a copy of var2 to var1
2. add var1 var2, add var2 to var1
3. lea var1 var2, copy the address of var2 to var1
4. push var1, place var1 on top of stack

The variable referenced as var1 and var2 in the example above can be changed into registers such as eax, ebx, and esp or addresses such as 0x6042, 0x4026, and 0x8048. Each of the instructions have a specific address in order for the machine to keep track of the execution.

B. angr

angr is a python framework for analyzing binaries. angr combines static, dynamic, and symbolic approach to create a consistent binary analysis. angr is capable of analyzing binary with certain instructions

1. loading the binary,
2. generate a path group object,

3. exploring and analysing path group,
4. adding constraints,
5. adding hooks to functions,
6. manipulate states.

The execution contains states consisting of binary's registers, memory content, files, and environment condition.

III. CASE STUDIES

In order to learn how angr works, several case studies will be tested. Each of the analysis and approach will be presented below.

A. *Carnegie Mellon University's Binary Bomb*

CMU's binary bomb consists of 6 phases in which for each phase student must prevent the execution to reach function called *bomb_explode* by providing correct input. This assignment is supposed to be solve by reading the binary's assembly. First until fourth phase's assembly code snippets will be shown and briefly explained below

- Phase 1

```
0x0000000000400ee0 <+0>:  sub    rsp,0x8
0x0000000000400ee4 <+4>:  mov    esi,0x402400
0x0000000000400ee9 <+9>:  call   0x401338
0x0000000000400eee <+14>: test   eax,eax
0x0000000000400ef0 <+16>: je     0x400ef7
0x0000000000400ef2 <+18>: call   0x40143a
0x0000000000400ef7 <+23>: add    rsp,0x8
0x0000000000400efb <+27>: ret
```

The first phase calls function of *read_string* at 0x401338 and compare it with string provided at address of 0x402400. If it is not equal then call *bomb_explode*.

- Phase 2

```
...
0x0000000000400f05 <+9>:  call   0x40145c
0x0000000000400f0a <+14>: cmp    DWORD PTR
[rsp],0x1
0x0000000000400f0e <+18>: je     0x400f30
0x0000000000400f10 <+20>: call   0x40143a
0x0000000000400f15 <+25>: jmp    0x400f30
0x0000000000400f17 <+27>: mov    eax, DWORD PTR
[rbx-0x4]
0x0000000000400f1a <+30>: add    eax,eax
0x0000000000400f1c <+32>: cmp    DWORD PTR
[rbx],eax
0x0000000000400f1e <+34>: je     0x400f25
0x0000000000400f20 <+36>: call   0x40143a
0x0000000000400f25 <+41>: add    rbx,0x4
0x0000000000400f29 <+45>: cmp    rbx,rbp
0x0000000000400f2c <+48>: jne    0x400f17
...
```

The second phase calls function of *read_six_numbers* at and check if the first number is 1 and the subsequent number is 2 times the previous number within the loop at the address of 0x400f17-0x400f2c. Thus the correct input would be 1 2 4 8 16 32.

- Phase 3

```
...
0x0000000000400f51 <+14>: mov    esi,0x4025cf
0x0000000000400f56 <+19>: mov    eax,0x0
0x0000000000400f5b <+24>: call   0x400bf0
0x0000000000400f60 <+29>: cmp    eax,0x1
0x0000000000400f63 <+32>: jg     0x400f6a
0x0000000000400f65 <+34>: call   0x40143a
0x0000000000400f6a <+39>: cmp          DWORD   PTR
[rsp+0x8],0x7
...
0x0000000000400fb7 <+116>:jmp    0x400fbe
0x0000000000400fb9 <+118>:mov    eax,0x137
0x0000000000400fbe <+123>:cmp       eax,DWORD  PTR
[rsp+0xc]
0x0000000000400fc2 <+127>:je     0x400fc9
0x0000000000400fc4 <+129>:call   0x40143a
...
```

The third phase calls function of *scanf* at 0x400bf0. This question is about jump table, so the first input of scanf is used to jump to certain address and the second input is checked with specific number such as written at address 0x400fbe. There are multiple correct solution to this phase.

- Phase 4

```
0x0000000000401024 <+24>: call   0x400bf0
0x0000000000401029 <+29>: cmp    eax,0x2
0x000000000040102c <+32>: jne    0x401035
0x000000000040102e <+34>: cmp          DWORD   PTR
[rsp+0x8],0xe
0x0000000000401033 <+39>: jbe    0x40103a
0x0000000000401035 <+41>: call   0x40143a
0x000000000040103a <+46>: mov    edx,0xe
0x000000000040103f <+51>: mov    esi,0x0
0x0000000000401044 <+56>: mov       edi,DWORD  PTR
[rsp+0x8]
0x0000000000401048 <+60>: call           0x400fce
<func4>
0x000000000040104d <+65>: test   eax,eax
0x000000000040104f <+67>: jne    0x401058
0x0000000000401051 <+69>: cmp          DWORD   PTR
[rsp+0xc],0x0
0x0000000000401056 <+74>: je     0x40105d
```

This fourth phase calls function of *scanf* at 0x400bf0. It scans 2 number and checks if the output of first number passed on to *func4* (0x400fce) is equal to 0 and second number. The *func4* itself is a variation of binary

search with the output being the number of search required for the first argument within the range of 0 to 14.

In order to solve these problems using angr first we need to load the binary to angr. It can be done using this line of code

```
proj = angr.Project('bomb')
```

After the binary is successfully loaded, set the starting position of execution, position to avoid, and end position. For the first phase these would be the line of code needed

```
start = 0x400ee0

bomb_explode = 0x40143a

end = 0x400ef7
```

Then initiate the starting position as the start variable

```
state = proj.factory.blank_state(addr=start)
```

After that symbolicly link the input with address of string

```
arg = state.se.BVS("string", 8 * 128)

state.memory.store(0x603780, arg)

state.add_constraints(state.regs.rdi == bind_addr)
```

After all the binding and state set, run the program with the constraints of bomb_explode

```
path = proj.factory.path(state=state)

ex = proj.surveyors.Explorer(start=path,
   find=(end,),avoid=(bomb_explode,),
        enable_veritesting=True)

ex.run()
```

Running the code for the first phase outputs "*Border relations with Canada have never been better.*", a valid answer in about 2 minutes time. Solving the second phase with a similar code with the difference of symbolic link being with 6 integers provide an output of "*1 2 4 6 8 16 32*", a valid output in about 5 seconds.

This shows that symbolic analysis performs better when arithmetic operation is used. In the second phase, the program mathematically calculate which number is supposed to be next as opposed to trying each possible character in the first phase.

The third phase have multiple possible solution. In order to get all of the solutions the program needs to be changed. First, it needs to have all corect path stored. One of the possible solution to that is to use a queue. The code execution is changed to do a while loop until the queue is empty with the queue is filled with all new possible states.

The solution provided turns out creating a big amount of possible solution path which in turns make the program use a big amount of memory and takes a lot of time. In order to solve it the program needs to prune the subset of all solution path inside the queue. Implementing that reduces the executing time to about 10 seconds.

The fourth phase consist of passing an input to a recursive function. There are no constraints that can be added to this phase. As such, solving this phase using the first phase's program would be the optimal approach. The program found a valid answer in about 5 minutes time.

As recursive functions are called multiple times, functions that have an infinite or big recursive depth will take a lot of time to solve by angr. A similar program with phase 4 with changes in the *func4* minimum and maximum value to 0 and 50 is tested using the code for phase 4. This code found a valid answer in more than 10 minutes time.

In order to solve the problem angr actually provide a parameter called LAZY_SOLVES that can be removed. Removing the parameter, using

```
remove_option={simuvex.o.LAZY_SOLVES}
```

optimizes the runtime. This makes the path exploration more efficient by ensuring unsatifiable path are not traversed.

IV. CONCLUSION

angr provides a way of analyzing a binary. Certain types of problems such as mathematical comparison can be solved by angr automatically and effectively. However, some problems such as recursion and string matching is more effectively solve manually. Some others are not solvable by angr as it have a huge amount of possible states. As such, angr is best used on a complex binary that consist of many mathematical calculations and comparisons.

ACKNOWLEDGMENT

First and foremost, praises and thanks to God the Almighty for His blessings throughout writing this paper. I would like to thank my parents for guiding me up until this point. I also would like to express my sincere gratitude to Dr. Ir. Rinaldi Munir, MT, Dr. Eng. Ayu Purwarianti, ST.,MT, and Dessi Puji Lestari ST,M.Eng.,Ph.D. for the guidance throughout the semester in IF3280 course.

REFERENCES

[1] Shoshitaishvili, Yan et al, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," IEEE Symposium on Security and Privacy, 2016.

[2] Stephens, Nick et al, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," NDSS, 2016.

[3] Shoshitaishvili, Yan et al, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," NDSS, 2015.

[4] Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. Intel Corporation. 1999. pp. 442 and 35. Retrieved 18 November 2010.

STATEMENT

I hereby declare that this paper is my own work and not a copy, translation, nor plagiarism of somebody else's work.

Bandung, May 5 2017

Jason Jeremy Iman 13514058