SWRhapsody

Binary instrumentation and symbolic execution

于9月 5, 2018由**SWRhapsody**发布

Warning

Some of the code in this article may not be correct, for I failed in compiling the Intel Pin tool with z3 solver, detail is explained in the last section.

Introduction

This exercise is using pin and z3 to solve some simple crack me. I mainly use the idea in [1].

And the source code of the crackme

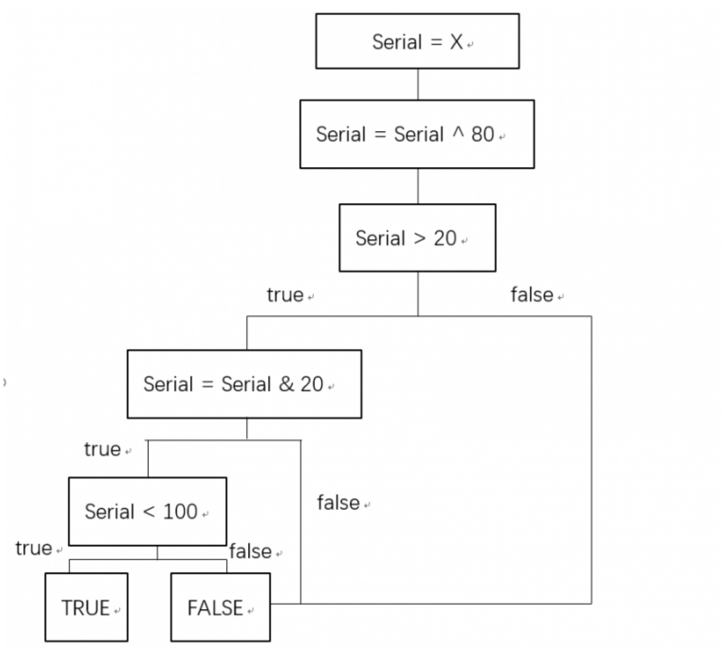
```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>

char *serial = "\x30\x39\x3c\x21\x30";

int main(void)
{
  int fd, i = 0;
  char buf[260] = {0};
  char *r = buf;

  fd = open("serial.txt", O_RDONLY);
  read(fd, r, 256);
  close(fd);
  while (i < 5){
    if ((*r ^ (0x55)) != *serial)
      return 0;
    r++, serial++, i++;
  }
  if (!*r)
    printf("Good boy\n");
  return 0;
}
```

The idea to use pin and z3 to solve this crack me is simple, as this crackme require a specific serial

SWRhapsody

from the file and after some computation, the code check whether is correct.

Consider a crackme with flowing logic



If we want to know what is the right serial we just need to solve equation

```
(x^80>20)&&(x^80&20)<100
```

z3 can solve the equation for us, and we can just track the data flow during the code execution using instrumentation to construct all the equations we need to compute the serial we want.

Code

Let's start coding part. First let's consider the logic of the crackme. The crackme read the serial from the "serial.txt" using open, and xor the first 4 char with 0x55 then compare with the right one. We could start our pin tool when crackme begin to open a file, and taint the memory read from the file. Then follow the tainted data to check whether it satisfy the required condition.

The start of our code is below

```
    void removeTaintedReg(REG reg);

    void taintMemory(UINT64 addr);

    void removeTaintedMemory(UINT64 addr);

    bool isMemoryTainted(UINT64 addr);
    bool isRegTainted(REG reg);

    UINT64 getRegID(REG reg);
    VOID setRegID(REG reg, UINT64 id);

private:
};

//global vars
//tainted memory and variable manager
TaintedManager tainted_mgr{};

//out put file stream
std::ofstream outfs;

//global flags
//global expr id
UINT64 uniqueID = 1;

//is last syscall is open
bool isLastOpen = false;

//use a list to store since file can be opened multi times
std::list<UINT64> target_file_fd;

//-----------------------------------------------------------------------

//----------------------------------------------------------------------
//smt,z3 related vars
z3::context z3_context;
//store all constrant expr on memory
//uniqueid starts from 1 this vector starts form 0
std::vector<z3::expr> z3_exprs;
//the serial we need to know
z3::expr target_expr = z3_context.bool_const("x");

char goodSerial[32] = {0};
unsigned int offsetSerial = 0;

VOID Syscall_entry(THREADID threadIndex, CONTEXT *ctxt,
                   SYSCALL_STANDARD std, VOID *v)
{
    if (PIN_GetSyscallNumber(ctxt, std) == __NR_open)
    {
        //check is going to open target file

        std::string filename(reinterpret_cast<char *>(PIN_GetSyscallArgument(ctxt, std, 0)));

        if (filename == KnobTaintFile.Value())
        {
            isLastOpen = true;
        }
```

```
        target_file_fd.remove(fd);
    }
    else if (PIN_GetSyscallNumber(ctxt, std) == __NR_read)
    {
        UINT64 fd = static_cast<UINT64>((PIN_GetSyscallArgument(ctxt, std, 0)));
        UINT64 start = static_cast<UINT64>((PIN_GetSyscallArgument(ctxt, std, 1)));
        UINT64 size = static_cast<UINT64>((PIN_GetSyscallArgument(ctxt, std, 2)));

        if (std::find(target_file_fd.begin(), target_file_fd.end(), fd) == target_file_fd.end(
            return;
        //tainted memory
        for (UINT64 i = 0; i < size; ++i)
            tainted_mgr.taintMemory(start + i);
        //show some msg
        std::cout << "[TAINT]\t\t\tbytes tainted from " << std::hex << "0x" << start << " to (
    }
}

VOID Syscall_exit(THREADID thread_id, CONTEXT *ctxt, SYSCALL_STANDARD std, void *v)
{
    if (isLastOpen)
    {
        //get the file desc and push it to the list
        target_file_fd.push_back(PIN_GetSyscallReturn(ctxt, std));
        isLastOpen = false;
    }
}
//-------------------------------------------------------------------------
/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */
/*   argc, argv are the entire command line: pin -t <toolname> -- ...    */
/* ===================================================================== */

int main(int argc, char *argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv))
        return Usage();

    //Sets the disassembly syntax to Intel format. (Destination on the left)
    PIN_SetSyntaxIntel();

    PIN_AddSyscallEntryFunction(Syscall_entry, 0);
    PIN_AddSyscallExitFunction(Syscall_exit, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

We start our monitoring with syscall open, close and read. If program read content form “serial.txt” into memory, we taint the memory as the begin of our analysis. Following the taint memory, we can track how serial is computed. But we need to know when should we start build the equation. Let’s just take a look at assembly code.

```
//compiled on ubuntu 18.04 with g++ 8.0.1
```

```
ab0:   83 f0 55               xor    $0x55,%eax
ab3:   89 c2                  mov    %eax,%edx
ab5:   48 8b 05 54 15 20 00   mov    0x201554(%rip),%rax        # 202010 <serial>
abc:   0f b6 00               movzbl (%rax),%eax
abf:   38 c2                  cmp    %al,%dl
ac1:   74 07                  je     aca <main+0xd1>
ac3:   b8 00 00 00 00         mov    $0x0,%eax
ac8:   eb 5e                  jmp    b28 <main+0x12f>
aca:   48 83 85 e8 fe ff ff   addq   $0x1,-0x118(%rbp)
ad1:   01
ad2:   48 8b 05 37 15 20 00   mov    0x201537(%rip),%rax        # 202010 <serial>
ad9:   48 83 c0 01            add    $0x1,%rax
add:   48 89 05 2c 15 20 00   mov    %rax,0x20152c(%rip)        # 202010 <serial>
ae4:   83 85 e0 fe ff ff 01   addl   $0x1,-0x120(%rbp)
```

From the assembly code we can see that the serial is moved into eax and then compared with the correct one. So our equation would be start with the mov instruction and end with the cmp instruction. I use the TaintedManager to manage all tainted data and when register is tainted, I will assign a unique id to the register, and create a expr. Each unique id is correspond one unique expr.

Don't forget we only want to inspect the tainted memory which come from "serial.txt", and we need to follow these data, so after move data from tainted memory, we tainted the register and keep tracking it until the register is filled with "clean" data.

```
VOID ReadMem(UINT64 insAddr, std::string insStr, UINT32 opCount, REG reg_r, UINT64 memAddr)
{
    //we only want to inspect mov instruction
    if (opCount != 2)
        return;

    //check whether memory address is tainted
    if (tainted_mgr.isMemoryTainted(memAddr))
    {
        std::cout << std::hex << "[READ in " << memAddr << "]\t" << insAddr << ": " << insStr
        std::cout << "[Constraint]\t\t"
                  << "#" << std::dec << REG_StringShort(reg_r) << " = 0x" << std::hex << std:::
                  << static_cast<UINT64>(*(reinterpret_cast<char *>(memAddr))) << std::endl;

        //tainted the register
        tainted_mgr.taintReg(reg_r);
        std::cout << "[Tainted]\t" << REG_StringShort(reg_r) << " is now tainted\n";

        //create a new constrant on this memory
        //as we already know the serial is in hex
        std::stringstream ss;
        //each id is in #id format
        ss << "#" << uniqueID;
        tainted_mgr.setRegID(reg_r, uniqueID++);
        //
        target_expr = z3_context.bv_const(ss.str().c_str(), 64);
        z3_exprs.push_back(target_expr);
    }
}
```

When tainted data is moved from memory to the register, we taint the register and create a z3 expr

SWRhapsody

Then we need to finish the computation, the main operation in this crackme is xor, so we do not need to implement add or decl. When code is performing the xor on a tainted register, update the z3 expr related to the register.

```
VOID xorRegReg(REG reg_l, REG reg_r, std::string insDis)
{
    //xor a tainted register with a tainted reg
    if (tainted_mgr.isRegTainted(reg_l) && tainted_mgr.isRegTainted(reg_r))
    {
        std::cout << "[XOR REG REG] " << insDis << "\n";
        //get the id and update the constrant
        UINT64 id_l = tainted_mgr.getRegID(reg_l);
        UINT64 id_r = tainted_mgr.getRegID(reg_r);
        assert(id_l != 0 && id_r != 0);

        z3_exprs[id_l - 1] = z3_exprs[id_l - 1] ^ z3_exprs[id_r - 1];
    }
}
```

After code execute for some while we should reach the cmp instruction, this one is simple, we just let z3 solve the equation we have built.

```
VOID cmpRegReg(REG reg_l, REG reg_r, CONTEXT *ctx)
{
    z3::solver s(z3_context);
    if (!tainted_mgr.isRegTainted(reg_l))
    {
        if (tainted_mgr.getRegID(reg_r) != 0)
        {
            //solve the equation
            s.add(z3_exprs[tainted_mgr.getRegID(reg_l)] == z3_exprs[tainted_mgr.getRegID(reg_r
            s.check();
        }
        else
        {
            s.add(z3_exprs[tainted_mgr.getRegID(reg_l)] == static_cast<int>(PIN_GetContextReg(
        }

        assert(s.check() == z3::check_result::sat);

        z3::model m = s.get_model();
        std::cout << "[Z3 Solver]-------------------------------------" << std::endl;
        unsigned int goodValue;

        Z3_get_numeral_uint(z3_context, target_expr, &goodValue);
        std::cout << "The good value is 0x" << std::hex << goodValue << std::endl;
        goodSerial[offsetSerial++] = goodValue;
        std::cout << "[Z3 Solver]-------------------------------------" << std::endl;
    }
}
```

This tool can only solve one char a time. We write the correct value back to the “serial.txt”. Repeat running it for several time to solve the crackme.

Source code https://github.com/Iceware/blog_code/blob/master/2018/taint_memory.cpp

SWRhapsody

When I was trying to compile the code and ran it, the pin complained it couldn't find "libz3.so" and after added the shared library into pin shared library search path pin complained `Unable to load libz3.so: dlopen failed: empty/missing DT_HASH in "libz3.so" (built with --hash-style=gnu?)` . After google it I found the pin is not using a standard libc library and I need compile z3 with pinCRT to make it work, but when I was trying to do this I met the same problem discussed in [2] and [3]. As this is only a exercise, I want to come back later to see whether I can solve this problem.

Reference

[1] http://shell-storm.org/blog/Binary-analysis-Concolic-execution-with-Pin-and-z3/#3.5

[2] https://github.com/sslab-gatech/qsym/issues/9

[3] https://github.com/s5z/zsim/issues/109

分类: EXERCISE

---

0 条评论

发表评论

名称 *

电子邮件 *

网站

SWRhapsody

发表评论

近期文章

近期评论

文章归档

SWRhapsody

分类目录

SWRhapsody

EXERCISE

CodeQL部分源码简读

Introduction CodeQL 也用了不少时间了，从最初的不会 阅读更多…

CHEAT SHEET

CodeQL 部分使用记录

前言 CodeQL 是一个代码分析引擎，主要原理是通过对代码进行构建并 阅读更多…

EXERCISE

Java 反编译工具

在复现 CVE-2019-15012 遇到了一个非常坑的地方，使用 J 阅读更多…

ABOUT