May the 11th - Fuzzing project report

Goal 🏆

- **Concentrate** on **PinTool** (Dyninst if I have time)
- **Change** my **counting conditional branches script** by using **Image instrumentation** rather than Routine instrumentation
- **Understand** the difference between **static library** and **dynamic library** (very important !!)
- **Test** my new script on **real** application/binary
- **Begin** to read some **papers** on **current fuzzing technics** and **their problems** (understanding step)
- **Begin** to collect **applications** on **Fuzzbench** before the second step of my internship

What I have done the last week ? 👨‍💻

PinTool : Image instrumentation (IMG)

Resources

[Pin: IMG: Image Object]() : The doc for IMG object
[Dynamic Binary Instrumentation and Pin]() : A huge doc with example to understand some IMG object functions

Explanation

After the meeting, I had to change my script. It must have to use Image instrumentation because of **symbol tables**. In fact, if during the dynamic instrumentation of a binary, the symbol line for the main function doesn't exist, my script **will don't work**. So, I have to concentrate on the **main "executable"** and not the **main "function"**. **Image instrumentation** allows me that.

New version of the `imagecountbranches.cpp` :

```
//
// This tool prints a trace of image load, image limits of main executable and the number of
//

#include "pin.H"
#include <iostream>
#include <fstream>
#include <stdlib.h>
using std::ofstream;
using std::string;
using std::endl;


KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "imagecountbranches.out", "specify file name");

ofstream TraceFile;

// The running count of branches is kept here
static UINT64 branchcount = 0;

// The IMG binary
IMG MainBinary;

const char * StripPath(const char * path)
{
        const char * file = strrchr(path,'/');
        if (file)
                return file+1;
        else
                return path;
}

// Checks if the instruction comes from the binary being instrumented.
BOOL CheckBounds(ADDRINT addr) {
    if(addr < IMG_HighAddress(MainBinary) && addr > IMG_LowAddress(MainBinary)){
        return true;
    }
    return false;
}

// Pin calls this function every time a new img is loaded
// It can instrument the image, but this example does not
// Note that imgs (including shared libraries) are loaded lazily

VOID Image(IMG img, VOID *v)
{
    if (IMG_IsMainExecutable(img)) {
            MainBinary = img;

        TraceFile << "[*] Main Binary Image: " << IMG_Name(img) << endl;
        TraceFile << "[+] Image limits 0x" << IMG_LowAddress(img) << " - 0x" << IMG_HighAddre
    }
}

// This function is called before every branch is executed
VOID BranchCount(ADDRINT addr, BOOL taken)
{
    if(CheckBounds(addr)) {
        branchcount++;
    }
}

VOID Instruction(INS ins, VOID *v)
{
        // Condition to insert a call (conditional branch and in the "main" executable)
        if (INS_IsBranch(ins) && INS_HasFallThrough(ins) == true) {

                // Insert a call to BranchCount before every branch, no arguments are passed
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)BranchCount, IARG_INST_PTR, IARG_BRANCH_T
        }
}

// This function is called when the application exits
// It closes the output file.
VOID Fini(INT32 code, VOID *v)
{
    TraceFile << "Number of conditional branches = " << branchcount << endl;

    if (TraceFile.is_open()) { TraceFile.close(); }
}

/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
    PIN_ERROR("This tool prints a log of image load, image limits of main executable and the
            + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}

/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */

int main(int argc, char * argv[])
{
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    TraceFile.open(KnobOutputFile.Value().c_str());

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(Image, 0);

    // Function
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

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

    return 0;
}
```

Static library vs. Dynamic library

Resources

[Linux Basics: Static Libraries vs. Dynamic Libraries]() : Simple read for understand difference between static and dynamic libraries
[How to compile gcc with static library?]()
[How to compile a static library in Linux?]()
[gcc options manual]()
[What do 'statically linked' and 'dynamically linked' mean?]()
[Difference between static and shared libraries?]()

Understanding

It's essential to understand their **differences**. In one hand, **static library** is a **library** which, during the compilation with the binary, give **one big binary** which contains **all the functions of the library** + the **main executable**. In the other hand, **dynamic library** is a **library** also but, during the compilation, the binary contains **only the main executable**. If he wants to use a function of the library during the runtime, he searches it on the memory if he exists or loads it.

About the script and the binary analysis, a binary **"statically linked"** or **"dynamically linked"** gives us **different results**. Why? **Image instrumentation** allows me to concentrate on the main "executable". However, in the case of **"statically linked"**, all of the functions of libraries are in the binary, so in our main "executable". So, we instrument some parts of code that we don't want (functions of libc for example).

Test on /bin/ls binary

Result

File information

```
sh-5.0# file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=2f15ad836be3339dec0e2e6a3c637e08e48aacbd, for GNU/Linux 3.2.0, stripped
```

Conclusion: /bin/ls is "dynamically linked" so our results didn't take into account functions of libraries, only our main "executable" of the binary

/bin/ls command without options

```
sh-5.0# /opt/pin-3.18-98332-gaebd7b1e6-gcc-linux/pin -t output/imageload.so -- /bin/ls
branchcount.cpp  generate.sh  imageload.cpp  imageload.out  output  pin.log  pintool.log  procbranchcount.cpp  test
sh-5.0# cat imageload.out
[*] Main Binary Image: /usr/bin/ls
[+] Image limits 0x94384437637120 - 0x94384437771087

Count of conditional branches = 2796
```

/bin/ls command with -la option

```
sh-5.0# /opt/pin-3.18-98332-gaebd7b1e6-gcc-linux/pin -t output/imageload.so -- /bin/ls -la
total 48
drwxr-xr-x 4 root root 4096 May 11 11:44 .
drwx------ 1 root root 4096 May 11 11:54 ..
-rw-r--r-- 1 root root 2720 May  5 18:08 branchcount.cpp
-rw-r--r-- 1 root root 1867 May  5 15:41 generate.sh
-rw-r--r-- 1 root root 3705 May 10 12:37 imageload.cpp
-rw-r--r-- 1 root root   89 May 11 12:30 imageload.out
drwxr-xr-x 2 root root 4096 May  6 18:52 output
-rw------- 1 root root  101 May  6 18:51 pin.log
-rw------- 1 root root  102 May 11 11:45 pintool.log
-rw-r--r-- 1 root root 3685 May  5 18:06 procbranchcount.cpp
drwxr-xr-x 3 root root 4096 May  6 16:43 test
sh-5.0# cat imageload.out
[*] Main Binary Image: /usr/bin/ls
[+] Image limits 0x94022341275648 - 0x94022341409615

Count of conditional branches = 6752
```

Reading some papers

Resources

[Magma: A Ground-Truth Fuzzing Benchmark]()

[FairFuzz : A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage]()

[Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization]()

What did I understand for now ? (not read entirely all of the papers)

Magma: : it's a ground-truth fuzzing benchmark. Their benchmarks based on real targets and bugs that satifies some properties (diversity, verifiability, usability) to have relevant benchmarks between fuzzers.

FairFuzz: I appreciated the reading of FairFuzz. This fuzzer based on AFL use a new fuzzing technic : "target rare branches" using mutation mask strategy algorithm. Why? The modern fuzzer such as AFL not cover large regions of code. So, the goal of FairFuzz it's to cover them.
Mutation masking strategy algorithm :

> We propose a novel lightweight mutation masking strategy to increase
> the chance of hitting the program regions that are missed by
> previously generated inputs.

Collecting some applications on FuzzBench

Resources

[Experiment data with FuzzBench]()

[FuzzBench Data]()

Explanation

I have to choose **some applications** for the **testing part** of the internship. With the **applications**, I have to find the **inputs** and the **bugs/CVE** that go with. I took `libxml2-v2.9.2` to understand the approach and repeat it with others applications.

Where I stopped ? 🛑

Collection applications and read again some parts of papers.