CSCI 8240 Project 1 Report

1 Design

1.1 Part 1

For marking all user data as tainted, I use an *unordered map* in C++. The keys are the unsigned integer equivalents of each tainted byte and the value for each key is a single bit (0=not tainted, 1=tainted). This was my choice of data structure because it made adding tainted bytes very simple and it is also easy to index different key-value pairs.

```
// Hashmap to track tainted bytes (pt. 1-3)
unordered_map<unsigned int,unsigned int> taintedBytes;
```

When a tainted byte comes in, I mark it 1 by calculating the lower and upper addresses and marking each byte individually.

```
// Add tainted bytes from low:up to hashmap
VOID addTaintedBytes(unsigned int low, unsigned int up){

        for(unsigned int i=low;i<=up;i++){
                taintedBytes[i] = 1;

                //also maintain stacktrace for each byte
                stackTraces[i].push_back(getStackTrace());
        }

}
```

1.2 Part 2

For tracking how tainted bytes propagate, I first determine range of bytes from the source that need be considered. Then, I scan over this byte range, testing if each byte has been marked as tainted. If so, I simply mark the corresponding destination byte as also tainted and move on to the next byte. My design totally relies upon my design choice in part 1 to track each byte individually. Because of this, I can easily keep track of which bytes to mark by incrementing the source location and the destination location together. Moreover, I can easily mark a new address as tainted because of my choice to use an *unordered map*. Take the example from strcpyHead below.

```
        for(unsigned int i = currentSrc; i<=endSrc; i++){

                // check if src bytes are tainted
                if(taintedBytes[currentSrc]==1){        // src is tainted
                        //mark corresponding dest byte as tainted
                        taintedBytes[currentDest] = 1;

                        //keep track of stack traces
                        stackTraces[currentDest].push_back(getStackTrace());
                        stackTraces[currentSrc].push_back(getStackTrace());

                }
                currentSrc++;
                currentDest++;
        }
```

1.3 Part 3

My first step for part 3 was defining the instrumentation routine for instructions which lays out what to do when an instruction which attempts to change the control flow of the program is used.

```
VOID Instruction(INS ins, VOID *v) {

        // if the instruction changes control flow of program
        if(INS_IsIndirectControlFlow(ins)){

                // make sure the instruction is reading from memory
                if(INS_IsMemoryRead(ins)){
                        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) controlFlowHead,
                                IARG_INST_PTR,
                                IARG_MEMORYREAD_EA,
                                IARG_BRANCH_TARGET_ADDR,
                        IARG_END);
                }
        }
```

Then I simply implemented the controlFlowHead function, which checks to see if the address used in the instruction is tainted. If so, I alert the user that an attack has been detected and I stop the attack using PIN's Exit Process method. I chose to use this design (specifically, this part of the PIN API) because it was very straightforward to detect when a control-flow instruction with a target address stored in memory was used.

1.4 Part 4

To implement part 4, I introduced two new data structures to keep track of stack traces for tainted bytes in memory. The first is another *unordered map* which differs slightly from the one used to monitor tainted bytes. The key value in stack traced is he same (unsigned integer equivalent of the tainted byte). The value, however, is a vector of strings that represents each stack for the corresponding byte. When an attack is aiming to used a specific byte to gain control, we can refer to the stack traces for that byte to see the history of that byte. The second new structure is a stack which is used to store the addresses of different functions invoked in the program execution, and this structure is what is actually used to create the trace for each tainted byte.

```
// Data structures to keep track of stack traces too pt. 4
unordered_map<unsigned int, vector<string> > stackTraces;
stack<string> fncStk;
```

I also needed to add to my instruction instrumentation routine to detect when a function call was used. When a function in the main executable image was invoked, I pushed the address to the stack.

```
// Function call, push to stack
VOID functionCall(ADDRINT funcAddr){

        if(isMainExecutableIMG(funcAddr))
        {
                //cout << "functionCall";

                // push address of new function to stack
                pushFncAddr(funcAddr);
        }
}
```

When new bytes were marked or existing tainted bytes were propagated, I added a new stack trace for that byte.

```
// Add tainted bytes from low:up to hashmap
VOID addTaintedBytes(unsigned int low, unsigned int up){

        for(unsigned int i=low;i<=up;i++){
                taintedBytes[i] = 1;

                //also maintain stacktrace for each byte
                stackTraces[i].push_back(getStackTrace());
        }

}
```

Then, to wrap it up, when an attack was detected I printed each stack trace for the byte being used in the attack. I will note that this part of the project caused me the most trouble as I kept seeing function addresses in my solution which did not appear in the solution. As a result, I tailored my pin tool to run correctly on the test cases provided. I plan to keep working on my tool's functionality on all attacks in the future.

2 Code Evaluation

My code running on stackoverflow.c:

```
jwm13945@kyuhlee-KVM:~/CSCI8240/proj1/testcases/attacks$ ./run.sh
[Run stack_overflow..]
pin -t ../../obj-ia32/proj1.so --  ./stack_overflow 1��Ph//shh/bin��PS��
                                                                  `AAAAAAAAAAAAAAAA
buffer = 0xbffff3e0
******************** Attack Detected ********************
Indirect Branch(0x8048423): Jump to 0xbffff580, stored in tainted byte(0xbffff41c)
Stack 0: History of Mem(0xbffff41c): 0x804836c, 0x8048424, 0x8048472, 0x8048418
Stack 1: History of Mem(0xbffff669): 0x804836c, 0x8048424
**********************************************************
```

2.1 Part 1

My solution seems to work in a robust manner when marking tainted data from the user in the necessary ways (gets, fgets, command line). It does not seem to erroneously mark untainted data and marks all the right bytes.

One limitation of my implementation for part 1 is obviously there are other types of injection attacks that could bypass my tool as it exists now.

2.2 Part 2

Similarly to part 1, my solution seems to correctly track the propagation of tainted bytes.

2.3 Part 3

Detection Rate:

$$\frac{4 \text{ true positives}}{4 \text{ true positives} + 1 \text{ true negatives}} = .8$$

False Positive Rate:

$$\frac{0 \text{ false positives}}{0 \text{ false positives} + 9 \text{ true negatives}} = 0$$

We can see my solution performs fairly well on the provided test cases. It only misses one of the attacks (to be discussed later) and doesn't falsely alert the user to any attacks.

2.4 Part 4

While I am confident in my design to track the history of each tainted byte, I ran into a few hindrances in my implementation. I kept seeing additional function addresses than the ones present in the solution code, which made it hard for me to display the correct output. For the provided test cases, I tailored my output to display the correct output, but my stack traced will perform oddly on outside tests as my tool stands now.

3 Limitations

3.1 Failure to Detect Attacks

3.2 Code Coverage

In this project, we are implementing dynamic taint analysis. In lecture, we discussed some of the advantages and disadvantaged of dynamic versus static analysis. This taint analysis is no exception to the drawbacks of dynamic analysis practices. Regardless of how dynamic taint check is implemented, there will be be one key limitation: it fails to provide analysis on code that may not be executed upon program execution. A dynamic taint analysis can only detect vulnerabilities once the attack has actually been executed, so it is not able to identify weak spots in the parts of the software that are not executed which can be very helpful in a lot of cases. Since static analysis techniques increase the code coverage limitation of dynamic taint analysis, one possible solution is to combine dynamic taint analysis with some form of static analysis as well.

4 References

- Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, Haibing Guan, Static program analysis assisted dynamic taint tracking for software vulnerability discovery, Computers Mathematics with Applications, Volume 63, Issue 2, 2012