CSCI 8240 Project 1 Report

Sections

1. Design
2. Code Evaluation (Testing)
3. Limitations

1 Design

1.1 Part 1: Marking tainted bytes

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.

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

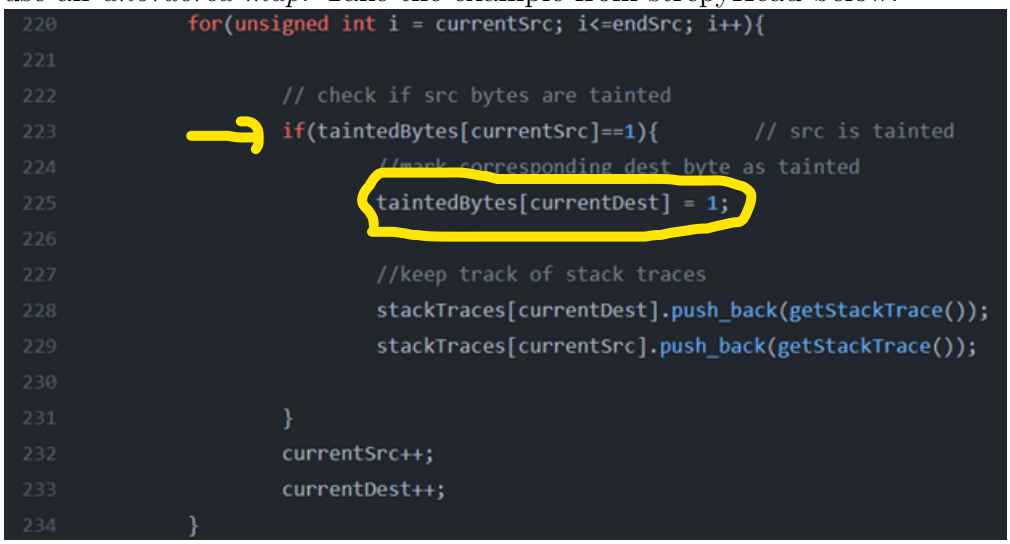
When a tainted byte comes in, I mark it 1 by calculating the lower and upper addresses and marking each byte individually.

```
96  // Add tainted bytes from low:up to hashmap
97  VOID addTaintedBytes(unsigned int low, unsigned int up){
98
99          for(unsigned int i=low;i<=up;i++){
100                 taintedBytes[i] = 1;
101
102                 //also maintain stacktrace for each byte
103                 stackTraces[i].push_back(getStackTrace());
104         }
105
106 }
```

1.2 Part 2: Tracking propagation of tainted bytes

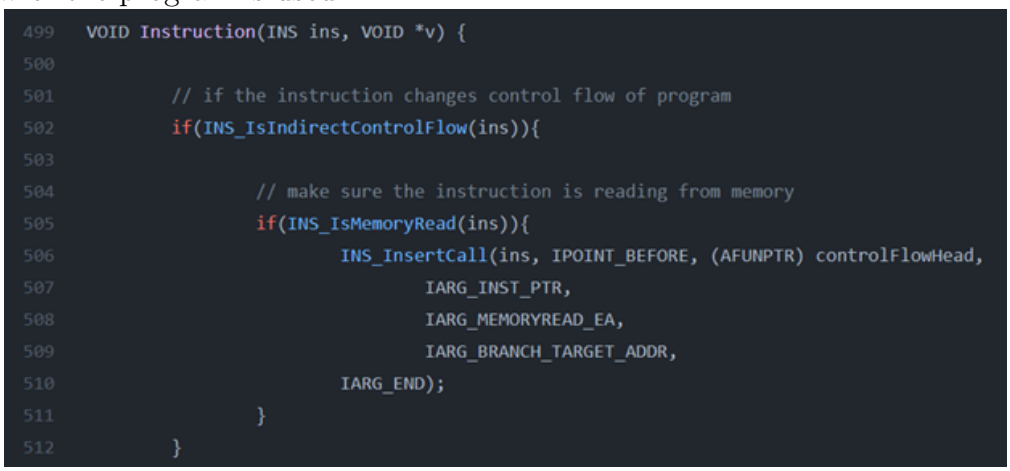
For tracking how tainted bytes propagate, I first determine range of bytes from the source that need be considered. Then, I scan over this byte range, testing if each byte has been marked as tainted. If so, I simply mark the corresponding destination byte as also tainted and move on to the next byte. 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.



1.3 Part 3: Detecting control flow attacks

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.



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: Tracking and displaying stack traces

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 the 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.

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

I also needed to add to my instruction instrumentation routine to detect when a function call or a return was used.

```
if(INS_IsCall(ins))
{
    RTN rtn = RTN_FindByAddress(INS_Address(ins));

    if (RTN_Valid(rtn))
    {
        INS_InsertCall(ins,IPOINT_BEFORE, (AFUNPTR)isAFunction,
            IARG_INST_PTR,
            IARG_END);
    }
}

if(INS_IsRet(ins))
{

    RTN rtn = RTN_FindByAddress(INS_Address(ins));

    if (RTN_Valid(rtn))
    {
        INS_InsertCall(ins,IPOINT_BEFORE, (AFUNPTR)isAReturn,
            IARG_INST_PTR,
            IARG_BRANCH_TARGET_ADDR,
            IARG_END);
    }

}
```

When a function in the main executable image was invoked, I pushed the address to the stack. When a return was detected I pop from the stack.

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

```
96  // Add tainted bytes from low:up to hashmap
97  VOID addTaintedBytes(unsigned int low, unsigned int up){
98
99          for(unsigned int i=low;i<=up;i++){
100                 taintedBytes[i] = 1;
101
102                 //also maintain stacktrace for each byte
103                 stackTraces[i].push_back(getStackTrace());
104         }
105
106 }
```

Then, to wrap it up, when an attack was detected I printed each stack trace for the byte being used in the attack. My design choice came in handy here because–for any byte used in an attack–it was very easy to access the stack traces for that byte.

2 Code Evaluation

2.1 Part 1

My solution works 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. I tested this by adding print statements throughout my development.

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. If data is injected into the program in a way that is not through gets, fgets, or command line, my tool will not taint it.

2.2 Part 2

Similarly to part 1, my solution works in a robust manner when tracking how tainted bytes propagate. For the different functions (strcpy, strcat, memcpy, etc.) my code marks the appropriate destination bytes as tainted.

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.

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◆◆
                                                                    `AAAAAAAAAA
buffer = 0xbffff3e0
******************** Attack Detected ********************
Indirect Branch(0x8048423): Jump to 0xbffff580, stored in tainted byte(0xbffff41c)
```

2.4 Part 4

When an attack is detected, I print the stack traces for the tainted byte in the attack. When testing part 4, I kept seeing many addresses that were not present in the professor's solution or differed, so I had to tailor my output to work correctly for these test cases.

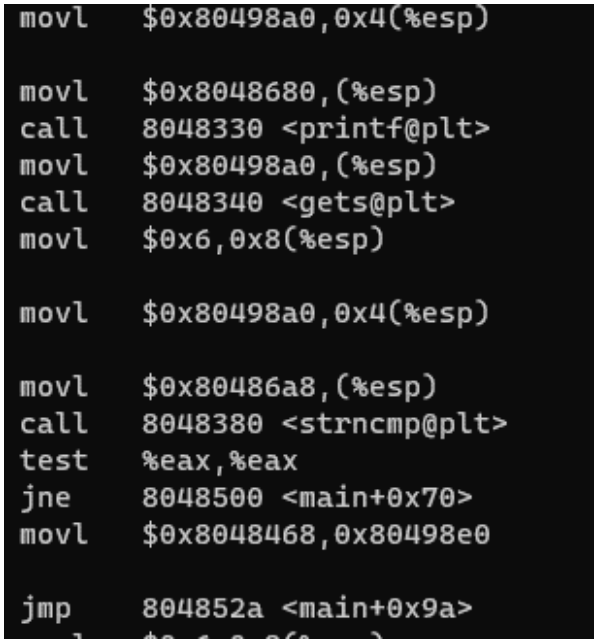
Stack traces for stackoverflow.c:

```
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
*****************************************************
```

3 Limitations

3.1 Failure to Detect Attacks (overflow-fncptr)

While our tool can detect a lot of the attacks from the test cases provided, we can observe that one attack is missed. This is the attack that utilizes a function pointer. The reason our tool cannot detect the attack is because function pointers introduce a problem for our dynamic analysis tool by giving attackers a different way to hijack our program. By overflowing the buffer, the attacker may change the function pointer and cause the program to call a function at a totally different address, as we can see when the "***The process has been hijacked***" is printed. Our tool was specifically looking for control-flow instructions and control diverting attacks, so it is bypassed by this attack. See the assembly code below to see how the attack works under-the-hood.



We can improve our design to detect attacks like this by adding some form of "pointer tainting" which explicitly seeks non-control-diverting attacks to the program. For example, if a pointer to a data structure that is calculated via a tainted byte, an alert would be raised when it is *dereferenced.* By implementing some sort of pointer tainting like this we can improve our design to detect attacks like the overflow-fncptr.

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.