Software Security
Static analysis tool for 64-bit architecture

Miguel Freire $_{84145}$, Ricardo Graça $_{93994}$

1 Introduction

In today's world, it's unthinkable to release any untested piece of software. If not tested, user input may drastically affect the behaviour of the program in a non intended way. Therefore, a lot of tools have been created in order to detect all sorts of vulnerabilities. These tools can be **static** if the analysis is performed on the source/object code, or **dynamic** if the analysis is done during the execution of the program [1]. Nowadays, most modern compilers already incorporate static analysis, making it easier for the program to be less susceptible to exploits. With that said, the main objective of this project is to build a static analysis tool to detect different types of vulnerabilities (enumerated below) in small pieces of **64-bit** x86 assembly code.

- **Variable overflow**
- **RBP overflow**
- **Return address overflow**
- **Invalid write access** - To non-assigned memory or to assigned memory out of the frame.

2 Design of the tool

Before starting the implementation, in **python (v3.7.0)**, we first looked at multiple examples and discussed on how could we tackle them. We started off by drawing the stack for each one of them, to get a better understanding on how to identify such vulnerabilities. For example, given the input assembly provided in the problem statement, we got the **main stack frame** represented in figure 1. From there, we started planning how we would build our tool.

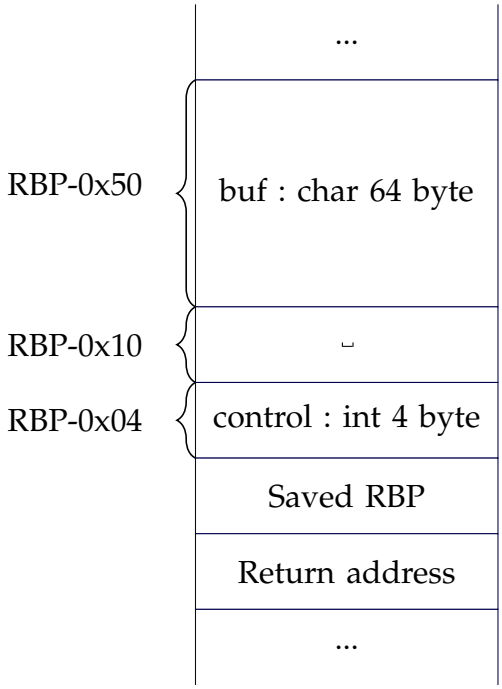

Figure 1. Stack example of the main frame

For all cases, we assume the worst, i.e, **the user will input a string with infinitely many characters**. In this case, imagining that a call to the C library function **gets** is made on the variable **buf** (it will overwrite all addresses below it), we can get all the types of vulnerabilities. So, by observing, we can say that it is certain that a call to **gets** will output at least three vulnerabilities of the ones mentioned. If instead of **gets** we call **fgets** we could not infer such thing, unless we had information about the arguments passed to it. If for say, the second argument of **fgets**, the input size, was 97, then it would print the same vulnerabilities as **gets** on this particular example. Taking this into

account, we decided to treat each dangerous function in isolation, since they behave in different ways depending on the context call.

As suggested in the tips, we started by creating a memory and a register model to keep track of all the values in each address. This is helpful, whenever there's a call to some function and we need to know what values are being inputted. We also created a way of emulating each assembly instruction, giving special attention to the **call** instruction since this is the one who is responsible for calling the dangerous functions, but **not the only** one that can output vulnerabilities. For example, considering the example **34_fgets_direct_access_invalidacc**, the instruction responsible for writing in unreserved memory (address EBP-0x10 on figure 1) is **mov**:

buf1[65] := 'a'

$\equiv$

mov BYTE PTR [rbp-0xf],0x61

In our implementation, the only way to get some sort of vulnerability is when at least one of these two instructions is executed.

Another thing that we had to consider was the position of the **null byte** in the buffer. Considering the following buffer with some data already in it:

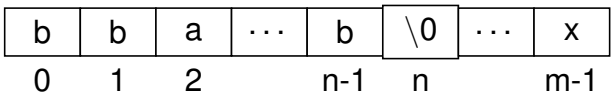

Figure 2. Example of some buffer with data

The size of the buffer on figure 2 is **m**, but if we copied it to a smaller sized buffer ($m' < m$), with the use of **strcpy**, we can't say for sure if there exists some overflow unless we track the position of the **null byte**. In this example, if $m' > n$ then a buffer overflow will not occur! Having these important details sorted out, we can now explain how we detect each vulnerability in the following subsections.

2.1 VAROVERFLOW

To detect a variable overflow, the first thing we did was to get the address of the buffer that is being targeted to be overflowed, let's call this address A. Then, depending on the dangerous call, we got the number of bytes that are going to be written to that buffer, let's call it n (n can be infinite). Getting the value of n depends on each dangerous call, for example, for the C library function **snprintf**, n will be the length of the concatenations of the strings passed to the arguments. Having A and n we can easily see if there is a variable overflow, we just get a new address $B = A + n$ and see if for some address, $C > A$, of the variables in the same frame : $C < B$. If this is the case, then we are sure that some variable was overflown.

2.2 RBPROVERFLOW, RETOVERFLOW and SCORRUPTION

For these three vulnerabilities we used the same reasoning in subsection 2.1. But instead of checking the address of all variables in the frame, we just put $C = RBP = 0$ for the saved RBP overflow, $C = RET = 8$ for the return pointer overflow and $C = RET = 16$ for writing outside of the frame. All these operations were considering RBP as the origin since all variable addresses in the same frame are in the form [RBP-<HEX>].

2.3 INVALIDACCS

In order to detect if some unreserved memory space was being written in some way, we first calculated the intervals of unreserved memory. We can see that in figure 1, the address RBP-0x10 all the way up to RBP-0x4 excluded is not reserved memory. Therefore, we add the interval [-0x10,-0x4[to some list L and do this process until all gaps are in the list. After getting all these spaces, once again, we use the same reasoning explained in 2.1, but now we check for intersections of intervals. Considering that $[A, B[$ is the interval where we are able to write, then we will check if for some L_i, $L_i \cap [A, B[\neq \emptyset$. If this is the case, we got ourselves a writing operation in a non-reserved space.

3 MAIN DESIGN OPTIONS

When the program loads the input file, functions, variables, and instructions are converted to internal representations. The main internal object is Program. Program contains all information related to the code you wish to analyze. It has all functions saved in an internal representation, each function contains all the instructions, variables and unused space in the form of stack intervals related to that frame. The Program also has a virtual memory and virtual registers to keep track of all the variables and temporary values. The virtual memory is implemented as a dictionary indexed by address, and the register also as a dictionary indexed by its name in the x64 Architecture and the value is a tuple formed by the registers value and the function where he was last altered. There also a zero flag to keep track of compare operations (ZF). While analyzing, if a vulnerability is found its record is saved in a list that is written to a file when the analysis ends.

The tool starts by loading the input file and convert all information to internal representation, then it starts by executing the main function in a sequential way, each instruction at a time. If the instruction handles registers and memory operations the program replicates all operations in the virtual registers and virtual memory (**add**, **sub**, **mov**, **lea**). For every instruction call the program, first, checks if it is a user-defined function or a dangerous libc function. If it is a dangerous libc function defined in our tool, then the arguments are analyzed and the operation replicated looking for overflow vulnerabilities. Else, if it is a user-defined function the program jumps to that frame and starts executing those function instructions. When it finds a **ret** instruction it returns to the previous frame. Our tool does not handle all libc functions only those defined in our tool. The full list follows in appendix A. This tool also supports other types of execution besides sequential. It supports conditional execution and generic jumps. When a **jmp** instruction appears the program jumps to the address defined as the argument of **jmp**. It does not support loops but it handles simple conditional structures such as if-else. When a **cmp** instruction appears, the program replicates the comparison and saves the result in the Zero Flag, then when a **je** or **jne** instruction is executed the Zero Flag is checked and the jump is done to the address given as argument if the condition is true.

If there is a frame overflow the tool stores all vulnerabilities related to that overflow but it does not halt, it continues to execute all remaining instruction trying to find more vulnerabilities.

4 OUTPUT EXAMPLES

For testing our tool, we build a script, **test.sh**, that automatically tells us if the basic and advanced tests were successful. Some of the tests were not successful and this is due to the fact that we don't stop the search for vulnerabilities whenever there's a **SCORRUPTION**!

Considering the following program in C, since control=41, then buf1 is copied to buf2 using the **strcpy**. But we don't know how the variables are put in the stack. For that we need to look into the addresses given by the assembly.

```
int main() {
  int control;
  char buf2[32];
  char buf1[64];

  control = 41;
  fgets(buf1, 64, stdin);
  if (control ==  41)
    strcpy(buf2, buf1);
  else
    strncpy(buf2, buf1, 32);

  return 0;
}
```

- **rbp-0x04** - address of control
- **rbp-0x30** - address of buf2
- **rbp-0x70** - address of buf1

With these addresses, we can see that putting 64 byte into **buf2** can overwrite the interval [EBP-0x30, EBP+10[, therefore it will print all

possible vulnerabilities, except for the SCORRUPTION.

python3 bo-analyser.py 41_ifs.json

Will produce the file 41_ifs.output.json containing:


```
[
    {
        "vulnerability": "VAROVERFLOW",
        "vuln_function": "main",
        "address": "4005f2",
        "fnname": "strcpy",
        "overflow_var": "buf2",
        "overflown_var": "control"
    },
    {
        "vulnerability": "INVALIDACCS",
        "vuln_function": "main",
        "address": "4005f2",
        "fnname": "strcpy",
        "overflow_var": "buf2",
        "overflown_address": "rbp-0x10"
    },
    {
        "vulnerability": "RBPOVERFLOW",
        "vuln_function": "main",
        "address": "4005f2",
        "fnname": "strcpy",
        "overflow_var": "buf2"
    },
    {
        "vulnerability": "RETOVERFLOW",
        "vuln_function": "main",
        "address": "4005f2",
        "fnname": "strcpy",
        "overflow_var": "buf2"
    }
]
```


5 UPDATES

- **[CHANGED]**: The output is now written into file instead of being written in the stdout.
- **[CHANGED]**: Organized the structure of the project in modules, instead of a unique file.
- **[ADDED]**: Implemented the remaining dangerous calls - *sprintf* and *snprintf*.
- **[ADDED]**: Implemented *jmp*, *je*, *jne* and *cmp* operations.
- **[ADDED]**: Additional code comments and documentation

6 CONCLUSION

The implemented tool supports all the basic and advanced proposed functionality. By using an emulation approach it can detect when overflow vulnerabilities occur and what parts of the stack are affected by the overflow. If a frame overflow happens it will detect all related vulnerabilities and it will also continue execution to try to find further vulnerabilities in other stack frames. It also finds format strings vulnerabilities but only with a max of two format string operators and without any additional characters. The tool is conservative, i.e. if there is a slight chance to occur a vulnerability it will report it, for example, if there is a read from stdin where an arbitrary number of bytes can be read then it will report all types of overflow vulnerabilities. The tool also supports simple conditional structures such as if-else's and also generic jumps and calls to user-defined functions, it does not support any kind of loop structures. It also does not support all libc functions and all opcodes from the x64 architecture. All the supported opcodes and dangerous libc functions are presented in appendix A. For simple programs, this tool can efficiently track static buffer overflows.

APPENDIX A
IMPLEMENTED FUNCTIONALITY

A.1 Implemented opcodes

ret, *leave*, *nop*, *push*, *pop*, *call*, *mov*, *lea*, *sub*, *add*, *cmp*, *jmp*, *je*, *jne*

A.2 Implemented libc dangerous functions

gets, *strcpy*, *strcat*, *fgets*, *strncpy*, *strncat*, *scanf*, *fscanf*, *read*, *sprintf*, *snprintf*

REFERENCES

[1] A. Ghahrai, *Static Analysis vs Dynamic Analysis in Software Testing*, 2017, https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/.