Static Analyser

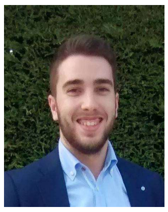
Report

Software Security

Alameda

Group 21

Mafalda Ferreira 81613



Leonardo Epifânio 83496

Pedro Lopes 83540

Description of the Experimental Part

Design of the tool

For this project we chose Java 8, using the OOP paradigm. The tool parses the JSON input using Jackson API, and instantiates the objects associated with each instruction of the target program. Then, it organizes them into basic blocks.

The algorithm that creates the basic blocks of the program is very simple, each instruction starts as a basic block, after that, it needs to determine the leaders' set (a leader is the first instruction in a basic block), with the following rules: a) the first statement is a leader, b) any statement that is the target of a goto is a leader, c) any statement that immediately follows a goto is a leader. For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program. Now, with the basic blocks, it builds the control flow graph (CFG) by checking the last statement of the basic block and adding, to the adjacency list, the basic block that it will jump to next (in the case of a conditional jump, there will be 2 basic blocks to where it can jump to).

For the internal representation of the memory, we created three different objects:

Registers
The Registers object holds all of the 17 data registers from the x86-64 architecture and, internally, each register is a *long* value (64 bits). The structure recognizes and works with all the x86-64 register subsection names, for instance, `ebx` represents the 32 LSB of the `rbx` register. The read/write operations on the registers are adapted having in mind all the idiosyncrasies that these operations have, in order to achieve full compatibility with the architecture.

MemoryPosition
The MemoryPosition is an object that represents a byte in memory. It holds the value of the byte itself and a reference to another object called Variable that is used later to map this byte and keep track of the memory state. The Variable object will represent if a byte of memory belongs to an allocated variable, or if it belongs to a return address, or the base pointer address. In case the memory is not mapped (the memory is allocated but it's not assigned), the MemoryPosition object will reference an empty Variable.

StackMemory
The StackMemory object is the core of the program. It represents a stack and behaves like a Linux process stack with little-endian format. It grows from the higher addresses to the lowest (due to the fact that we're using Java, and the internal implementation was made with a list, this isn't fully true, but it is abstracted by the StackMemory's API). It holds an ordered collection of MemoryPosition and operates over that collection. The most important operations are: allocate, deallocate, map, readByte, readValue, writeByte and writeValue. Every time that a writeByte/writeValue is called, it verifies if it's writing in a valid position according to the current memory state. If not, it will write, and then return a Vulnerability object with all the details describing the newfound vulnerability.

Main design options

The generation of the basic blocks, CFG and the execution is implemented using the Visitor Design Pattern, whose state is maintained and operated by calls to the objects described previously. The execution of the CFG consists of a DFS. At the end of the DFS, a list of vulnerabilities are outputted and serialized to JSON objects via Jackson API and dumped to a file with the extension ".output.json".

Output of the tool

We created an example to test our tool. This is the source:

```
1.   void fun1(char *buf1, char *buf2){
2.      int lazy;
3.      char buf4[64];
4.      sprintf(buf4,"%s%s", buf1, buf2);
5.   }
6.
7.   int main() {
8.     char buf3[64]
9.     char buf2[32];
10.    char buf1[16];
11.     int control = 41;
12.     fgets(buf2, 32, stdin);
13.     read(STDIN_FILENO, buf1, 16)
14.     if(control == 41)
15.       fun1(buf1, buf2);
16.     else
17.       strcpy(buf2, buf1);
18.     return 0;
19.   }
```

Given that: lazy@[rbp-0x8], buf4@[rbp-0x48], buf3@[rbp-0x40], buf2@[rbp-0x60], buf1@[rbp-0x70] and control@[rbp-0x78]

Our tool outputted the following 5 vulnerabilities:

Nº	Vulnerability	Vuln Function	Address	Function	Overflow Var	Overflown Var	Overflown Address	op
1	VAROVERFLOW	main	4005ef	strcpy	buf2	buf3	-	-
2	VAROVERFLOW	fun1	4005ab	sprintf	buf4	lazy	-	-
3	INVALIDACCS	fun1	4005ab	sprintf	buf4	-	rbp-0x4	-
4	RBPOVERFLOW	fun1	4005ab	sprintf	buf4	-	-	-
5	INVALIDACCS	fun1	40058f	-	-	-	rbp-0x5a	mov

This returns the expected vulnerabilities. This happens because of the read in line 13, where the program puts 16 bytes in the buffer and none of them is a null byte. We also tried the public tests and we got the expected result in all of them, except for test 34 where we got the same vulnerability but with a different overflown address. Our tool outputted 0x10 while the provided solution says it is 0xf instead. We know that our solution is correct for that test.

Discussion

Guarantees provided by the tool

Under the restrictions and the simplified model described in the project's specification, our tool is able to detect variable overflows, RBP overflows, return address overflows and invalid write accesses (to non-assigned memory in the current frame, or to memory out of the current frame).

When our tool finds a vulnerability caused in a function call, it will continue the analysis, verifying if other subsequent function calls are safe. For example, if the program has a ***gets*** followed by a ***strcpy***, we will analyze the vulnerabilities in the ***strcpy*** regardless of whether we found any in ***gets***.

Limitations of the tool

- **False Negatives**

1) Our tool doesn't detect a buffer overflow if, in an assignment, the memory position we are trying to write to, is mapped to a variable. For example:

```
1. int control;
2. int buffer[64];
3. buffer[65] = 'A';
```

Assume the *control* variable is located in the address 0x70 and the buffer is located at 0x40 (0x70 - 0x40 = 0x30 = 64 bytes). The 3rd line translates to the assembly operation `MOV [0x70],0x41`. Since the address 0x70 belongs to a mapped variable (control), our tool will accept this as a valid operation. We can also verify that our tool won't detect invalid access if we also try to read. This vulnerability can be exploited since we can read or write in unauthorized variables, which can result in anything from changing the program flow, to leaking memory.

- **False Positives**

1) Our tool will wrongly detect stack corruption if the program has a function that receives a reference to a variable. For example:

```
1. void main() {
2.   char buffer[64];
3.   func1(buffer);
4. }
```

```
5. void func1(char * buffer) {
6.   char buffer2[32];
7.   strcpy(buffer, buffer2);
8. }
```

In line 7, when we try to write in the `buffer` variable, our tool will detect that its address is out of the stack frame (because it is in the *main* stack frame) and will output a *SCORRUPTION* vulnerability. This could be avoided if we could verify whether or not the address we're trying to write to is passed in the arguments.

2) Our tool will also detect invalid access if the program tries to write in the space that's used for the current function arguments. Our tool only maps the variables declared in the function, so the extra memory that is allocated in the beginning of the frame remains unmapped throughout the execution. When the program tries to write in that space, our tool will output *INVALIDACCS*, since that memory is not allocated. One possible way to avoid this would be mapping all the allocated memory above the memory, mapped with variables, as "good". However this is not conservative enough, because it can lead to false-negatives. For instance, if there is alignment in memory between arguments, accessing that chunk of memory should return an *INVALIDACCS*, and using this approach it will not. For that reason, we decided not to implement that way.

3) We also detected an inconsistency related to the control flow of the program. For instance, if we have:

```
1. if (i == 0)
2.   //some code
3. if (i != 0)
4.   //some other code
```

The algorithm that generates the control flow graph will generate 4 different flows, including one flow that includes the blocks inside both of the *if* conditions and another that includes none. In the given example, these paths would be absurd. Statically it is very hard to avoid this kind of limitation. Our best option would be checking the condition statically whenever it's possible and reduce the set of possible paths. When it's impossible (it depends on the provided input) we would need heuristics in order to approximate the probable result.

Possible extensions

We designed this solution in way that makes it somewhat scalable. To extend the capabilities of the analyser, the general structure of the program would be kept the same. There could, then, be added more operations and vulnerable functions, and the implementation would be mainly focused on extending the visitors and the VulnerableFunctions class to support these new cases.

We could make the tool more precise if we could somehow combine the information provided by the JSON input with the C source code. This would enable us to:

- Know the arguments of each function, allowing us to solve the 1st and 2nd false positives we described in the tool's limitations. Since our tool is capable of knowing the variable associated with an address if that memory is mapped, for the 1st one, we could verify if, in the context of a function, the address we are trying to access belongs to a variable passed in the arguments. For the 2nd one, knowing the arguments and their types, we could map those addresses in the memory, making the access valid.

- In the context of a direct write access, know if the address we are trying to access belongs to the variable performing the access. For example, given the instruction `a[10] = 20`, we could verify if the address `a[10]` belonged to the `a` variable (using arithmetic), enabling us to know if it was in fact an invalid access vulnerability.

These changes would make the program less efficient because it would have more information to deal with. It would also need to perform lexical analysis, parse the source code and gather all the needed information, and process it in order to relate it with the JSON input. Hence, more verifications would be needed becoming less scalable.

Changes from previous submission

1. Filename parsing fixed.
    - The program would try to write the JSON output file into the wrong directory.
2. Implemented a project requirement that was missed in the first delivery:
    "The **output** should be a JSON object stating all the existent vulnerabilities of the program ***EXCEPT when there exists an overflow of the return address in which case you should only output the vulnerabilities for the function that is vulnerable to this overflow.***"
    - In the previous delivery, it would output all the vulnerabilities found in all of the functions regardless whether or not there was a return address overflow. The class in which we made changes for this implementation was Executor.
3. Refactoring of VulnerableFunctions class. Improved readability and scalability.
4. Adding relevant documentation.
5. JUnit tests fixed. (Unnecessary for the project, just to help in the development).