fenix.tecnico.ulisboa.pt

FenixEdu™

FenixEdu

25-32 minutes

To achieve an in-depth understanding of a security problem. To carry out a hands-on approach to the problem, by implementing a tool for tackling it. To analyse its underlying security mechanism according to the guarantees that it offers, and to its intrinsic limitations. To understand how the proposed solution relates to the state of the art of research on the security problem. To develop collaboration skills.

Format

The Project is presented in Section 2 as a problem, and its solution should have the following two components:

1. An experimental part, consisting in the development of a tool, that is to be presented and discussed.
2. A report (max 4 pages), that describes briefly
   - the experimental part: design of the tool, the main design options, and the output of the tool for a few examples (maximum 2 page)
   - discussion: discusses the guarantees provided by the tool, as well as its limitations, and possible extensions that would provide more security guarantees.

Submissions (updated)

Projects are to be solved in groups of 2 or 3 students, that should register via Fenix by **9 November**. The reports should be submitted also via Fenix, as a pdf.

- The submission deadline for the code is **16 November 23:59**. Please submit a zip file containing your code.
- The submission deadline for the report is **23 November 23:59**. Please submit a zip file containing both the report and updated code.
  - *We will accept updates to the code submitted by 16 November with this submission. You should describe your changes from the previous submission in the report.*

Evaluation

The experimental part will be evaluated based on the submitted work and a demo and a discussion that will take place during the lab classes of the **last two weeks before Christmas break.**

The report will be evaluated according to the following criteria:

- Quality of writing - structure of the report, clarity of the ideas, language
- Content - relevance and value of the ideas that are conveyed
- Depth - understanding of the state of the art, connection with experimental work

All sources should be adequately cited. Plagiarism will be punished according to the rules of the School.

A large class of vulnerabilities in applications originates in programs that enable user input information to affect the values of certain parameters of security sensitive functions. In other words, these programs encode an illegal information flow, in the sense that low integrity -- tainted -- information (user input) may interfere with high integrity parameters of sensitive functions (so called sensitive sinks). This means that users are given the power to alter the behavior of sensitive functions, and in the worst case may be able to induce the program to perform security violations such as following a non-intended execution flow.

The aim of this project is to study how vulnerabilities in binaries can be detected statically by means of taint and input validation analysis.

This part consists in the development of a static analysis tool for identifying data and control flow integrity violations when inputs are not subject to proper input validation. Static analysis is a general term for techniques that verify the behavior of applications by inspecting their code (typically their source code). Static analysis tools are complex, so **the purpose is not to implement a complete tool**. Instead, the objective is to implement a tool that analyses small pieces of x86 assembly code.

The piece of code below, both in C and (simplified) assembly in Intel syntax, can easily be seen to have a buffer overflow. The base pointer of `fun1` is overwritten by `buf2` if one introduce `16+` characters to the `fgets`.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void fun1(char buf[]){
  char buf2[16];
  strcpy(buf2, buf);
}

int main() {
  int control;
  char buf[64];

  control = 13;
  fgets(buf, 20, stdin);

  fun1(buf);
}

<main>
400589:    push   rbp
40058a:    mov    rbp,rsp
40058d:    sub    rsp,0x50
400591:    mov    DWORD PTR [rbp-0x4],0xd
400598:    mov    rdx,QWORD PTR [rip+0x200aa1]        # 601040
<stdin@@GLIBC_2.2.5>
40059f:    lea    rax,[rbp-0x50]
4005a3:    mov    esi,0x14
4005a8:    mov    rdi,rax
4005ab:    call   400480 <fgets@plt>
4005b0:    lea    rax,[rbp-0x50]
4005b4:    mov    rdi,rax
4005b7:    call   400567 <fun1>
4005bc:    mov    eax,0x0
4005c1:    leave
4005c2:    ret

<fun1>
400567:    push   rbp
400568:    mov    rbp,rsp
40056b:    sub    rsp,0x20
40056f:    mov    QWORD PTR [rbp-0x18],rdi
400573:    mov    rdx,QWORD PTR [rbp-0x18]
400577:    lea    rax,[rbp-0x10]
40057b:    mov    rsi,rdx
40057e:    mov    rdi,rax
400581:    call   400470 <strcpy@plt>
```

```
400586:    nop
400587:    leave
400588:    ret
```

Your tool should be able to:

- given a program in json format (details on the structure of this json in Section Input/Output format) containing
  - a subset of these assembly instructions;
  - and the information where each variable occurs in the frame;
- given a set of dangerous functions.

return the analysis of whether

1. *(Basic Analysis - clarification)* there exists a **variable overflow** (some variable is overflown by another). In our example this does not happen but it would be present in the case we had `fgets(buf, 77, stdin)` instead of `fgets(buf, 20, stdin)`; `buf` would overflow `control`.
2. *(Basic Analysis)* there exists a **`RBP` overflow** (the value stored in `RBP` is overflown, ie, the `saved RBP`; in our example `fgets(buf, 20, stdin)` will lead to an overflow of `RBP` in `fun1`);
3. *(Basic Analysis - clarification)* there exists a **return address overflow** (the value stored in `RBP+0x8` is overflown, ie, the `saved RIP`). In our example this does not happen but it would be present in the case we had `fgets(buf, 89, stdin)` instead of `fgets(buf, 20, stdin)`. It would overflow `RBP+0x8` in `main`). It would also happen in the case we had `fgets(buf, 25, stdin)` as it would would overflow `RBP+0x8` in `fun1`.
4. *(Advanced Analysis)* there exists an **invalid write access** to non-assigned memory in the current frame (in our example any access to registers in the range `[RBP-0x10, RBP-0x05]` in `main`);
5. *(Advanced Analysis - clarification)* there exists an **invalid write access** to memory out of the current frame (in our example any access to registers *not* in the range `[RBP-0x50, RBP+0x10]` in `main`, or any access to registers *not* in the range `[RBP-0x20, RBP+0x10]` in `fun1`).

You may assume our simplified model where

- we will only consider overflow vulnerabilities that occur in the stack;
- **(updated)** the list of dangerous functions you need to consider is here;
  - `scanf` and `printf` families and `read` are now part of the *Advanced* solution;
- **(updated)** the list of assembly instructions you need to consider is here;
  - `call` to generic functions is now part of the *Advanced* solution; as a consequence there is only function `main` and call to dangerous functions in the *Basic* solution;
  - removed `pop`;
- there are no loops nor recursion in the program;

- all the memory needed in the stack is allocated at the beginning of the function;
- all variables are referrenced with respect to their frames `RBP` (and `RIP` for static values);
- **(clarification)** no arithmetic operations over pointers nor registers (except for `rbp` and `rsp` that are needed to move the stack up and down at the beginning of a function; `sub rsp,0x70` is ok; `sub rax,0x10` is not ok).
- *(only for Advanced)* consider *direct write access*, as a way of corrupting memory (eg, `a[10]=20`);
- *(only for Advanced)* direct access to array positions is not indexed by a variable, ie, there are no `a[i]` occurences in the code.
- *(only for Advanced)* consider `jmp` and conditional statements.

64-bit vs 32-bit architectures

The developed tool should aim at a 64-bit architecture. The main impact in the solution resides on passing arguments in registers, rather than passing them in the stack. These avoids extra `push` instructions at the beginning of the function, and pop in the end, to store, respectively restore, the values in the registers once a function is called, respectively returns.

Also, the registers in 64-bit architecture are 64-bits, ie, double the size of the 32-bit architectures. The (32-bit) registers available in 32-bit architectures have an equivalent (64-bit) register in 64-bit architecture with an `R` instead of `E`. The full list of 64-bit registers is

```
RAX,RBX,RCX,RDX,RDI,RSI,R8,R9,R10,R11,R12,R13,R14,R15,RBP,RSP,RIP
```

The arguments of functions are passed in the following order in the registers

```
RDI,RSI,RDX,RCX,R8,R9 + stack for arguments beyond the 6th (with 7th argument
being the one on top)
```

respectively for the 1st, 2nd, 3rd, 4th, 5th, and 6th argument of a function.

Running the tool

The way to call your tool depends on the language in which you choose to implement it (you can pick any you like), but **it should be called in the command line with a single argument `<program>.json` and produce the output referred below and no other to a file `<program>.output.json`**.

```
./bo-analyser program.json
python ./bo-analyser.py program.json
java bo-analyser program.json
```

The json **input file** `test13.json` for the code presented in our example is the following:

```
        {
            "main": {
                "Ninstructions": 15,
                "variables": [
                    {
                        "bytes": 4,
                        "type": "int",
                        "name": "control",
                        "address": "rbp-0x4"
                    },
                    {
                        "bytes": 64,
                        "type": "buffer",
                        "name": "buf",
                        "address": "rbp-0x50"
                    }
                ],
                "instructions": [
                    {
                        "op": "push",
                        "pos": 0,
                        "args": {
                            "value": "rbp"
                        },
                        "address": "400589"
                    },
                    {
                        "op": "mov",
                        "pos": 1,
                        "args": {
                            "dest": "rbp",
                            "value": "rsp"
                        },
                        "address": "40058a"
                    },
                    {
                        "op": "sub",
        "pos": 2,
        "args": {
            "dest": "rsp",
            "value": "0x50"
        },
        "address": "40058d"
    },
    {
        "op": "mov",
        "pos": 3,
        "args": {
            "dest": "DWORD PTR [rbp-0x4]",
            "value": "0xd"
        },
        "address": "400591"
    },
    {
        "op": "mov",
        "pos": 4,
        "args": {
            "dest": "rdx",
            "value": "QWORD PTR [rip+0x200aa1]",
            "obs": "# 601040 <stdin@@GLIBC_2.2.5>"
        },
        "address": "400598"
    },
    {
        "op": "lea",
        "pos": 5,
        "args": {
            "dest": "rax",
            "value": "[rbp-0x50]"
        },
        "address": "40059f"
    },
    {
        "op": "mov",
                "pos": 6,
                "args": {
                    "dest": "esi",
                    "value": "0x14"
                },
                "address": "4005a3"
            },
            {
                "op": "mov",
                "pos": 7,
                "args": {
                    "dest": "rdi",
                    "value": "rax"
                },
                "address": "4005a8"
            },
            {
                "op": "call",
                "pos": 8,
                "args": {
                    "fnname": "<fgets@plt>",
                    "address": "400480"
                },
                "address": "4005ab"
            },
            {
                "op": "lea",
                "pos": 9,
                "args": {
                    "dest": "rax",
                    "value": "[rbp-0x50]"
                },
                "address": "4005b0"
            },
            {
                "op": "mov",
                "pos": 10,
                "args": {
                    "dest": "rdi",
                    "value": "rax"
                },
                "address": "4005b4"
            },
            {
                "op": "call",
                "pos": 11,
                "args": {
                    "fnname": "<fun1>",
                    "address": "400567"
                },
                "address": "4005b7"
            },
            {
                "op": "mov",
                "pos": 12,
                "args": {
                    "dest": "eax",
                    "value": "0x0"
                },
                "address": "4005bc"
            },
            {
                "op": "leave",
                "pos": 13,
                "address": "4005c1"
            },
            {
                "op": "ret",
                "pos": 14,
                "address": "4005c2"
            }
        ]
    },
    "fun1": {
        "Ninstructions": 12,
        "variables": [
            {
                "bytes": 16,
                "type": "buffer",
                "name": "buf2",
                "address": "rbp-0x10"
            }
        ],
        "instructions": [
            {
                "op": "push",
                "pos": 0,
                "args": {
                    "value": "rbp"
                },
                "address": "400567"
            },
            {
                "op": "mov",
                "pos": 1,
                "args": {
                    "dest": "rbp",
                    "value": "rsp"
                },
                "address": "400568"
            },
            {
                "op": "sub",
                "pos": 2,
                "args": {
                    "dest": "rsp",
                    "value": "0x20"
                },
                "address": "40056b"
            },
            {
```

```
        "op": "mov",
        "pos": 3,
        "args": {
            "dest": "QWORD PTR [rbp-0x18]",
            "value": "rdi"
        },
        "address": "40056f"
    },
    {
        "op": "mov",
        "pos": 4,
        "args": {
            "dest": "rdx",
            "value": "QWORD PTR [rbp-0x18]"
        },
        "address": "400573"
    },
    {
        "op": "lea",
        "pos": 5,
        "args": {
            "dest": "rax",
            "value": "[rbp-0x10]"
        },
        "address": "400577"
    },
    {
        "op": "mov",
        "pos": 6,
        "args": {
            "dest": "rsi",
            "value": "rdx"
        },
        "address": "40057b"
    },
    {
        "op": "mov",
                    "pos": 7,
                    "args": {
                        "dest": "rdi",
                        "value": "rax"
                    },
                    "address": "40057e"
                },
                {
                    "op": "call",
                    "pos": 8,
                    "args": {
                        "fnname": "<strcpy@plt>",
                        "address": "400470"
                    },
                    "address": "400581"
                },
                {
                    "op": "nop",
                    "pos": 9,
                    "address": "400586"
                },
                {
                    "op": "leave",
                    "pos": 10,
                    "address": "400587"
                },
                {
                    "op": "ret",
                    "pos": 11,
                    "address": "400588"
                }
            ]
        }
    }
```

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.

1. *(Basic Analysis)* what is the vulnerability, either `VAROVERFLOW`, `RBPOVERFLOW`, `RETOVERFLOW`;
2. *(Advanced Analysis)* what is the vulnerability, either `INVALIDACCS`, `SCORRUPTION`;
3. the dangerous function/instruction that causes the vulnerability;
4. which function of your program is vulnerable,
5. what is the address where the dangerous function/instruction is called;
6. *(Basic Analysis)* the overflowing buffer. In the case of `VAROVERFLOW` it should also state the overflown variable.
7. *(Advanced Analysis)* the overflowing buffer and the first address of the block(s) that is(are) overflown.

More details about the output format in Section Input/Output format.

In our example there was a single vulnerability

```
[
    {
        "vulnerability": "RBPOVERFLOW",
        "vuln_function": "fun1",
        "address": "400581",
        "fnname": "strcpy",
        "overflow_var": "buf2"
    }
]
```

The tool should be tested with all the programs provided, but be generic and configurable to analyse other programs. Also, and in spite of the non-existence of a general way to describe the dangerous functions, you should try to deal with them in an abstract way.

Given the intrinsic limitations of the static analysis problem, the developed tool is necessarily imprecise. It can be unsound (produce false negatives), incomplete (produce false positives) or both. Define and discuss what your tool is able to achieve, while answering the following questions:

1. Explain what are the imprecisions that are built into the proposed mechanism. Have in mind that they can originate at different levels, for instance:
    - imprecise tracking of information flows -- Are all illegal information flows captured by the adopted technique? Are there flows that are unduly reported?
    - imprecise endorsement of input validation -- Are there sanitization functions that could be ill-used and not properly validate the input? Are all possible validation procedures

detected by the tool?

2. For each of the identified imprecisions that lead to:
    - undetected vulnerabilities (false negatives) -- Can these vulnerabilities be exploited? If yes, how (give concrete examples)?
    - reporting non-vulnerabilities (false positives) -- Can you think of how they could be avoided?

3. Propose one way of making the tool more precise, and predict what would be the tradeoffs (efficiency, precision) involved in this change.

Experimental Part (80% of the project grade - 16 out of 20)

Grading of the experimental part will reflect the level of complexity of the developed tool. The *Basic analysis* is worth 13 (out of 16) of the total grade for the experimental part. Extra credit will be given to solutions that include parts of the *Advanced analysis*.

Attention will be given to:

- the precision of the tool;
- the subset of the language that the tool handles (minimum requirements are the *assembly instructions* and *dangerous functions* that appear in the examples provided; up to 1 value of extra credit will be given to solutions that go beyond those);
- the internal representation of the memory, modularity, quality of code and documentation.

Report (20% of the project grade - 4 out of 20)

The maximum grade of the report does not depend on the complexity of the tool. It will of course reflect whether the analysis of the imprecisions matches the precision of the tool that was developed (which in turn depends on the complexity of the tool). The components of the grading are:

- Quality of writing (1 out of 20)
- Identification of imprecisions (1 out of 20) - See questions 1.a) and 1.b) in Section Project Report above.
- Understanding of imprecisions (1 out of 20) - See questions 2.a) and 2.b) in Section Project Report above.
- Improving precision (1 out of 20) - See question 3 in Section Project Report above.
- Have in mind the grading criteria specified in the project presentation.

Input

In order to simplify the parsing of the inputs, you will receive `json` objects that are the result of parsing the corresponing assembly file of the programs. The structure of the `json` object is the following:

```
{
    "f1": {                             // a function named f1
        "Ninstructions": 10,            // number of assembly instructions of f1
        "variables": [                  // list of the variables of f1
            {
                "bytes": 4,             // size of this variable
                "type": "int",          // type of this variable
                "name": "var11",        // name of this variable
                "address": "rbp-0x4"    // location of this variable wrt rbp of
f1
            },
            {
                "bytes": 64,            // another variable of f1
                "type": "buffer",
                "name": "var12",
                "address": "rbp-0x50"
            }
        ],
        "instructions": [               // list of the instructions of f1
            {                           // 1st instruction of f1
                "op": "push",           // assembly operation
                "pos": 0,               // position of the instruction within f1
                "args": {               // arguments of the instruction
                    "value": "rbp"
                },
                "address": "400589"     // memory address of the instruction
            },
            ...
            {                           // 9th instruction of f1
                "op": "call",
                "pos": 8,
                "args": {
                    "fnname": "<fgets@plt>",
                    "address": "400480"
                },
                "address": "4005ab"
            }
            ...
        ]
    },
    "f2": {                             // another function named f2
        "Ninstructions": 12,
        "variables": [                  // list of the variables of f2
            {
                "bytes": 16,
                "type": "buffer",
                "name": "var2",
                "address": "rbp-0x10"
            }
        ],
        "instructions": [               // list of the instructions of f2
            {                           // 1st instruction of f2
                "op": "push",
                "pos": 0,
                "args": {
                    "value": "rbp"
                },
                "address": "400567"
            },
            ...
            {                           // 12th instruction of f2
                "op": "ret",
                "pos": 11,
                "address": "400588"
            }
        ]
    }
}
```

Each individual instruction is encoded as follows. Notice that these examples were obtained from random programs:

```
// 400588:    ret
// (similarly for leave, nop)
{
    "op": "ret",
    "pos": 11,
    "address": "400588"             // address where the ret instruction is
}


// 400589:    push   rbp
// (similarly for pop)
{
    "op": "push",
    "pos": 0,
    "args": {
        "value": "rbp"              // value that is being pushed
    },
    "address": "400589"             // address where the push instruction is
}


// 4005ab:    call   400480 <fgets@plt>
{
    "op": "call",
    "pos": 8,
    "args": {
        "fnname": "<fgets@plt>",    // name of the function being called
        "address": "400480"         // address of the called function
    },
    "address": "4005ab"             // address where the functon is called
}


// 400598:    mov    rdx,QWORD PTR [rip+0x200aa1]        # 601040
<stdin@@GLIBC_2.2.5>
// (similarly for lea, sub, add)
{
    "op": "mov",
    "pos": 4,
    "args": {
        "dest": "rdx",
        "obs": "# 601040 <stdin@@GLIBC_2.2.5>",     // this one is optional
        "value": "QWORD PTR [rip+0x200aa1]"
    },
    "address": "400598"
}



// 400565:    jmp    400573 <main+0x4c>
// (similarly for je, jne)
{
    "op": "jmp",
    "pos": 14,
    "args": {
        "address": "400573"
    },
    "address": "400565"
}



// 400553:    cmp    DWORD PTR [rbp-0x4],0x0
// (similarly for test)
{
    "op": "cmp",
    "pos": 10,
    "args": {
        "arg0": "DWORD PTR [rbp-0x4]",
        "arg1": "0x0"
    },
    "address": "400553"
}
```

Output

The ouptut of the program is also a `json` object that should be written to a file `test.output.json` when the program under analysis is `test.json`.

```
[
    {   // overflow of var control of function main, by variable buf
        "vulnerability": "VAROVERFLOW",
        "vuln_function": "main",
        "address": "4005ab",
        "fnname": "fgets",
        "overflow_var": "buf",
        "overflown_var": "control"
    },
    {   // overflow of rbp of main
        "vulnerability": "RBPOVERFLOW",
        "vuln_function": "main",
        "address": "4005ab",
        "fnname": "fgets",
        "overflow_var": "buf"
    },
    {   // overflow of ret address of main
        "vulnerability": "RETOVERFLOW",
        "vuln_function": "main",
        "address": "4005ab",
        "fnname": "fgets",
        "overflow_var": "buf"
    },
    {   // (Advanced) access to non-reserved memory in the frame of function main
        "vulnerability": "INVALIDACCS",
        "vuln_function": "main",
        "address": "4005ab",
        "fnname": "fgets",
        "overflow_var": "buf",
        "overflown_address": "rbp-0x10"
    },
    {   // (Advanced) writing in memory out of the stack frame of main
        "vulnerability": "SCORRUPTION",
        "vuln_function": "main",
        "address": "4005ab",
        "fnname": "fgets",
        "overflow_var": "buf",
        "overflown_address": "rbp0x10"
    }
]
```

Assembly instructions to consider

- **(updated) Basic:** `ret`, `leave`, `nop`, `push`, ~~`pop`~~, `call` (of dangerous functions), `mov`, `lea`, `sub`, `add`
- **(updated) Advanced:** `call` (of generic functions), `cmp`, `test`, `je`, `jmp`, `jne`

Dangerous functions to consider

- **(updated) Basic:** `gets`, `strcpy`, `strcat`, `fgets`, `strncpy`, `strncat`
- **(updated) Advanced:** `sprintf`, `scanf`, `fscanf`, `snprintf`, `read`

Tips

- *(06Nov2018)* Start with the *Basic solution*. Define a memory and register model and keep track of the tainted addresses.
- *(06Nov2018)* Notice that there are different kinds of operations: `register-to-register`, `register-to-pointer`, `pointer-to-register`, `number-to-register`, `number-to-pointer`.
- *(04Nov2018)* Differences between `lea` and `mov` (link).
- Try to analyse how adversarial controlled input propagates throughout the code. Be consevative when in doubt.

References

- x86 Assembly Guide

FAQS

Logs of changes

- *(06Nov2018)* - Trimmed down the requisites for the Basic solution.
  - See *Section 3, Simplified Model* and *Section 7, Assembly and Dangerous Functions*.
- *(06Nov2018)* - Updated Tips. Clarified what errors should be returned and in which situations.
- *(04Nov2018)* - Fixed broken links.
- *(04Nov2018)* - Changed the order of keys in json objects in the vulnerabilities' output in order to make them uniform. Notice that this is just to improve readability of the document and should have no impact in your solution as there is no order in json objects. You should use (*eg*, Python)

function `json.loads` to transform a string into `json` object, and `json.dumps` to transform a `json` object into a string.

Attachments

- public_advanced_tests.zip (updated 14Nov2018)
- public_basic_tests.zip (updated12Nov2018)
- discussoes.pdf