Interim Report

Candidate Number: 164597

November 2019

Contents

1 Introduction

In brief: The overall goal of this project is to create a toy compiler for a simple language. Which will then be used to run some code to smash the stack, and begin an exploit known as return oriented programming (ROP).

Note: There are no intended users for this project, all the work produced is meant to be private i.e. for external evaluation and my eyes only.

1.1 Brief Introduction to the Problem Area

1.1.1 Buffer Overflow Attacks

This section covers buffer overflow exploits, which will be the primary vector for beginning a ROP exploit in this project. To first understand the topic, a recap on stack frames in memory is needed.

The figure below shows a simplified view[1] of what a function with one local array variable would look like in memory. One of the key concepts to grasp is that the function that called the current one sits directly higher up in memory.

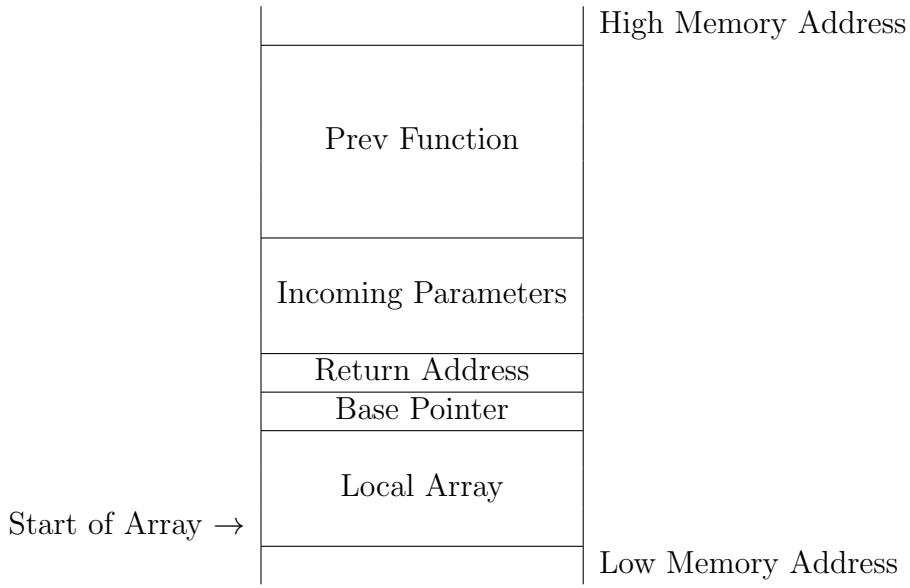

Figure 1: General Stack Frame Layout

The return address is crucial to this structure. Its job is to tell the program where to go back to in the previous function after it's done in the

current one. Therefore if it is able to be altered, an attacker can essentially to tell the program to start doing something else.

1.1.2 NX defense

1.1.3 ROP

hello

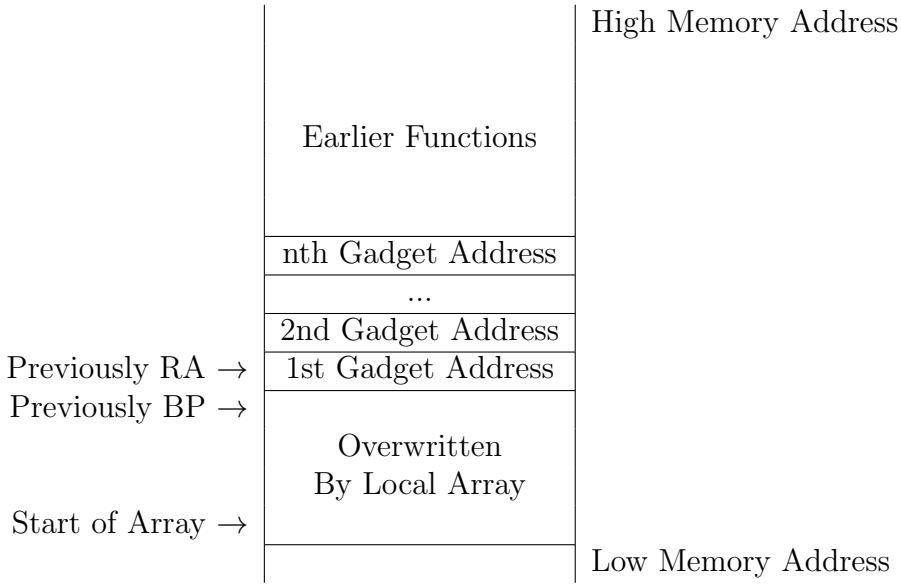

Figure 2:

1.1.4 To-do Summary

1.1.5 Software Engineering Considerations

1.2 Aims

One of the primary aims of this project will be to implement a compiler for the language given in section 5.2. This language is a modified subset of the C language, that contains only the bare bones needed to perform the buffer overflow exploit to begin ROPing.

An additional aim for this project will be to target the x86-64 assembly language for the compiler output. This architecture is chosen since it is the easiest to perform ROP exploits on. This is due to the fact that boundaries on instruction sizes aren't enforce like they are in other architectures such as

MIPS.

A final aim for this project is to target the exploit to run on a virtualised Linux distro. Initially this distro will have defenses such as ASLR disabled, but as the project progresses the aim is to ultimately run the ROP exploit with the defenses in place. However, this is relegated to the extensions phase of the project as one of the first to be attempted.

1.3 Objectives

1.4 Motivation

1.5 Project Relevance

2 Professional and Ethical Considerations

3 Related Work

4 Requirements Analysis

5 Appendix

5.1 Project Plan

5.2 EBNF Grammar

Main things to note:

1. This language explicitly differentiates between arrays on the stack and ones on the heap.
2. The new and free keywords break from C convention so as not to mislead about malloc usage.
3. This language has the idiom declarations before definitions baked in. This is done for ease of type checking.
4. The precedence of operators is follows those in regular C.

Language:

⟨*Goal*⟩ ::= ⟨*FunctionDeclaration*⟩* ⟨*MainFunction*⟩ ⟨*FunctionDefinition*⟩*

⟨*FunctionSignature*⟩ ::= ⟨*SignatureType*⟩ ⟨*Identifier*⟩ '`(`' (⟨*SignatureType*⟩ ⟨*Identifier*⟩ ('`,`' ⟨*SignatureType*⟩ ⟨*Identifier*⟩)*)? '`)`'

⟨*FunctionDeclaration*⟩ ::= ⟨*FunctionSignature*⟩ '`;`'

⟨*FunctionDefinition*⟩ ::= ⟨*FunctionSignature*⟩ '{' ⟨*FunctionBody*⟩ '}'

⟨*FunctionBody*⟩ ::= ⟨*VarDeclaration*⟩* ⟨*Statement*⟩* '`return`' ⟨*Expression*⟩ '`;`'

⟨*MainFunction*⟩ ::= '`int`' '`main`' '`(`' '`int`' ⟨*Identifier*⟩ '`,`' '`char`' '`[`' '`]`' '`[`' '`]`' ⟨*Identifier*⟩ '`)`' '{' ⟨*FunctionBody*⟩ '}'

⟨*VarDeclaration*⟩ ::= ⟨*Type*⟩ ⟨*Identifier*⟩ '`;`'

⟨*Type*⟩ ::= ⟨*StackArray*⟩
| ⟨*HeapArray*⟩
| ⟨*PrimitiveType*⟩

⟨*SignatureType*⟩ ::= ⟨*PrimitiveType*⟩
| ⟨*HeapArray*⟩

⟨*PrimitiveType*⟩ ::= '`int`'
| '`char`'
| '`boolean`'

⟨*StackArray*⟩ ::= ‘`int`’ ‘`[`’ ⟨*IntegerLiteral*⟩ ‘`]`’
  | ‘`char`’ ‘`[`’ ⟨*IntegerLiteral*⟩ ‘`]`’

⟨*HeapArray*⟩ ::= ‘`int`’ ‘`[`’ ‘`]`’
  | ‘`char`’ ‘`[`’ ‘`]`’

⟨*Statement*⟩ ::= ‘`{`’ ⟨*Statement*⟩* ‘`}`’
  | ‘`if`’ ‘`(`’ ⟨*Expression*⟩ ‘`)`’ ⟨*Statement*⟩ ‘`else`’ ⟨*Statement*⟩
  | ‘`while`’ ‘`(`’ ⟨*Expression*⟩ ‘`)`’ ⟨*Statement*⟩
  | ⟨*Identifier*⟩ ‘`=`’ ⟨*Expression*⟩ ‘`;`’
  | ⟨*Identifier*⟩ ‘`[`’ ⟨*Expression*⟩ ‘`]`’ ‘`=`’ ⟨*Expression*⟩ ‘`;`’
  | ‘`free`’ ⟨*Identifier*⟩ ‘`;`’
  | ‘`print`’ ‘`(`’ ⟨*Expression*⟩ ‘`)`’ ‘`;`’

⟨*Expression*⟩ ::= ⟨*Expression*⟩ (‘`&&`’ | ‘`||`’ | ‘`!=`’ | ‘`==`’ | ‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’ |
    ‘`+`’ | ‘`-`’ | ‘`*`’) ⟨*Expression*⟩
  | ⟨*Expression*⟩ ‘`[`’ ⟨*Expression*⟩ ‘`]`’
  | ⟨*Identifier*⟩ ‘`(`’ (⟨*Expression*⟩ (‘`,`’ ⟨*Expression*⟩)*)? ‘`)`’
  | ⟨*IntegerLiteral*⟩
  | ⟨*CharacterLiteral*⟩
  | ⟨*BooleanLiteral*⟩
  | ⟨*Identifier*⟩
  | ‘`new`’ ‘`int`’ ‘`[`’ ⟨*Expression*⟩ ‘`]`’
  | ‘`new`’ ‘`char`’ ‘`[`’ ⟨*Expression*⟩ ‘`]`’
  | ‘`!`’ ⟨*Expression*⟩
  | ‘`(`’ ⟨*Expression*⟩ ‘`)`’

5.3 Example Program

```
int factorial(int num);

int main(int argc, char[][] argv) {
    print(factorial(5));
    return 0;
}

int factorial(int num) {
    int aux;

    if (num < 1)
        aux = 1;
    else
        aux = num * (factorial(num - 1));

    return aux;
}
```

5.4 Proposal

References

[1] Eli Bendersky. Stack frame layout on x86-64. `https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64`, September 2011. [Online; accessed 1-November-2019].