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

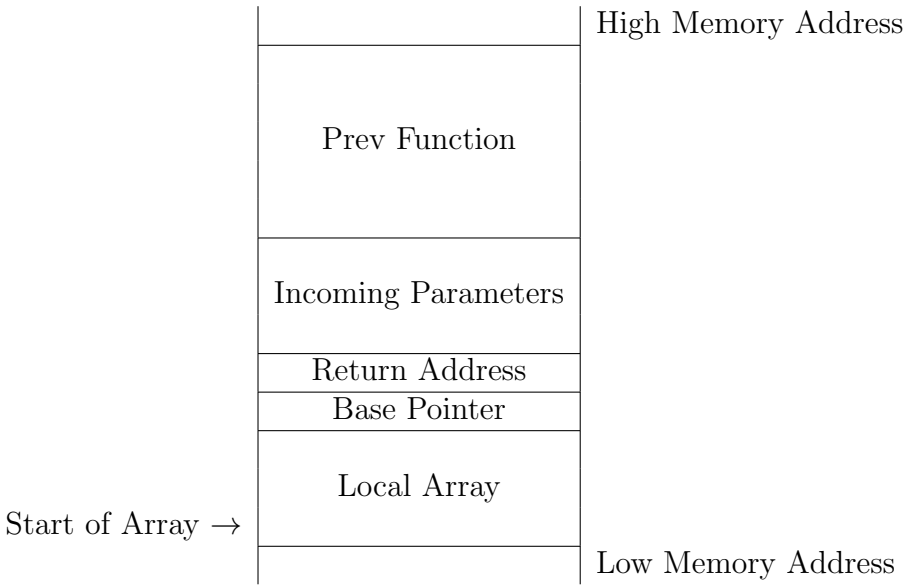

Figure 1: General Stack Frame Layout

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 tell the program to start doing something else.

Fortunately the return address is accessible with the use of the local array. Since array accesses are encoded by adding to the start of the array, an array access outside the bounds goes higher up in memory than what is usually safe. Therefore by using calculated unsafe array accesses and assignments, the return address can be altered to whatever the attacker wants.

1.1.2 The NX/XD defense

Typically due to the basic fact that instructions are data and vice-versa. The way that simple attacks would work is that before placing the return address, an attacker would write in the code they want to run. Then target the return address back at this sequence that's just been written.

However, this attack type of attack was realised and fairly easily defended against. The defense put in place relies upon the idea that the stack in memory should only be used to store data, not code to be executed. Therefore by putting a flag which disallows execution on certain regions of memory, you can effectively shutdown the attacker being able to write what they want then execute it. They can only write what they want, not execute it, hence the name No Execute (NX) or Execute Disable (XD) [2].

1.1.3 ROP

ROP is among one of the methods of effectively sidestepping this defense. The core idea behind it and many other such attacks is to execute pre-existing sequences without this NX/XD flag, rather than find a way to turn it off.

However, one of the main challenges in this approach is to find such desirable sequences that an attacker would want to execute. In terms of ROP these sequences are called gadgets and use of them requires a little arranging beforehand. The structure of a gadget is simply a sequence of code that is guaranteed to execute its final instruction, that is the return ("ret") instruction.

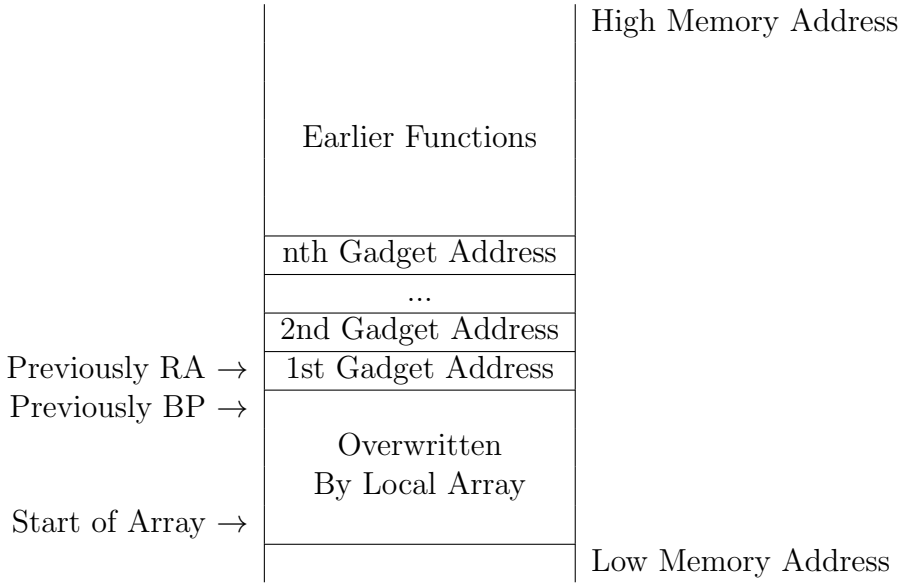

Figure 2: Example Prepped ROP Stack

What "ret" does, is it pulls an address off the stack and starts executing the instructions from there. Therefore, if several addresses are placed consecutively in memory like what is shown above. Since all the gadgets end in "ret", what will end up happening is the body of the gadgets will run one after the other with the "ret" chaining them together. Each "ret" at the end of the gadget will pull the next address of the stack, and start the whole process again from there. Because the gadget bodies are also executed, through careful arrangement this allows arbitrary code execution to be performed.

1.1.4 Components of a Toy Compiler

This section will be giving a short overview of the components necessary to implement the compiler for the language given in 6.2.

The Lexer:

The job of the lexer is simply to tokenise the input. The computer sees code as a long list of characters e.g. ['i', 'n', 't', ' ', 'x', '=']. As one would imagine this is rather inconvenient to work with. So to tokenise is to take this list of characters and produce a list of corresponding tokens with the useless info removed i.e. whitespace and comments. A tokenise example from above is as such [INT, x, EQ].

The Parser:

The job of the parser is to take the tokens given from the lexer, and produce an easy to work with structure that captures the essence of the program that was written. There are several parser tools available that take a grammar not too dissimilar from 6.2 and automate this process. In this project the Java CUP parser [ref needed] is one such tool that will be used.

The Type Checker:

Essentially the job of the type checker is to catch all semantic errors in the program structure (given by the parser) that may have been made. For example the programmer might try to assign True to a variable that holds an integer. Obviously for safety reasons this code should not move past this phase. The way this will be accomplished is through the use of a common design pattern known as the Visitor pattern [ref needed].

The Code Generator:

Finally and perhaps the most self explanatory phase is code generation. The code generator operates on the same structure after type checking is successful, and produces correct machine code that is able to be run directly on the computer. A point to note is that as this is a toy compiler no optimisations of such code will be attempted in this project.

1.2 Aims

One of the primary aims of this project will be to implement a compiler for the language given in section 6.2. 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 other defenses such as ASLR (not mentioned in this report) 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 Software Engineering Considerations

3 Professional and Ethical Considerations

4 Requirements Analysis

4.0.1 Non-Functional Requirements

4.0.2 Functional Requirements

4.0.3 Domain Requirements

5 Related Work

6 Appendix

6.1 Project Plan

6.2 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*⟩ ‘`)`’

6.3 Example Program

```
int factorial(int n);

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

int factorial(int n) {
    int x;

    if (n < 1)
        x = 1;
    else
        x = n * factorial(n - 1);

    return x;
}
```

6.4 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].

[2] Hewell Packard. Data execution prevention. http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf, May 2005.