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. This will then be used to run a program to smash the stack, and begin an exploit known as return oriented programming (ROP).

1.1 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 the function that called the current one sits directly higher up in memory.

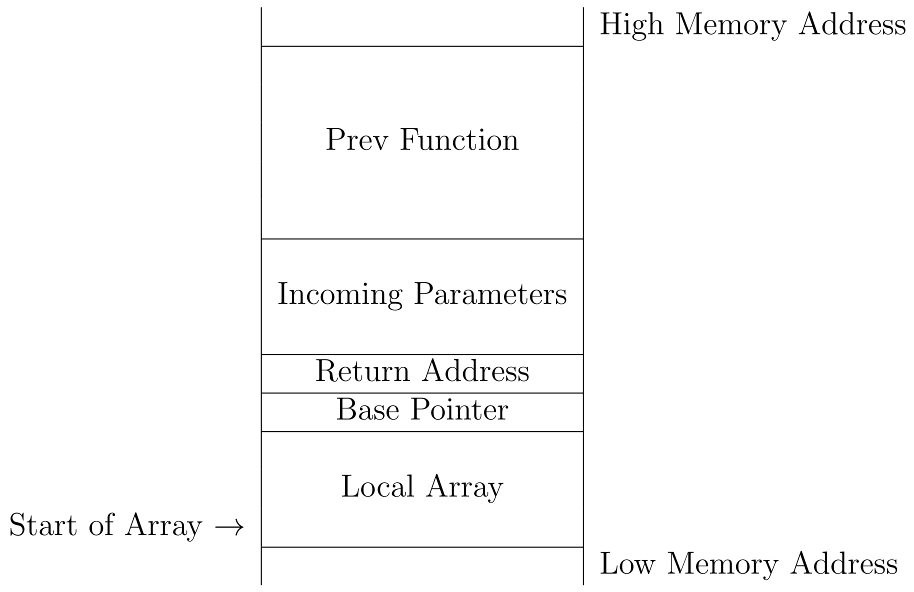

Figure 1: General Stack Frame Layout

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 with 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 they would target the return address back at the sequence that's just been written.

However, this attack type of attack was identified and defended against some time ago. 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 and 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.

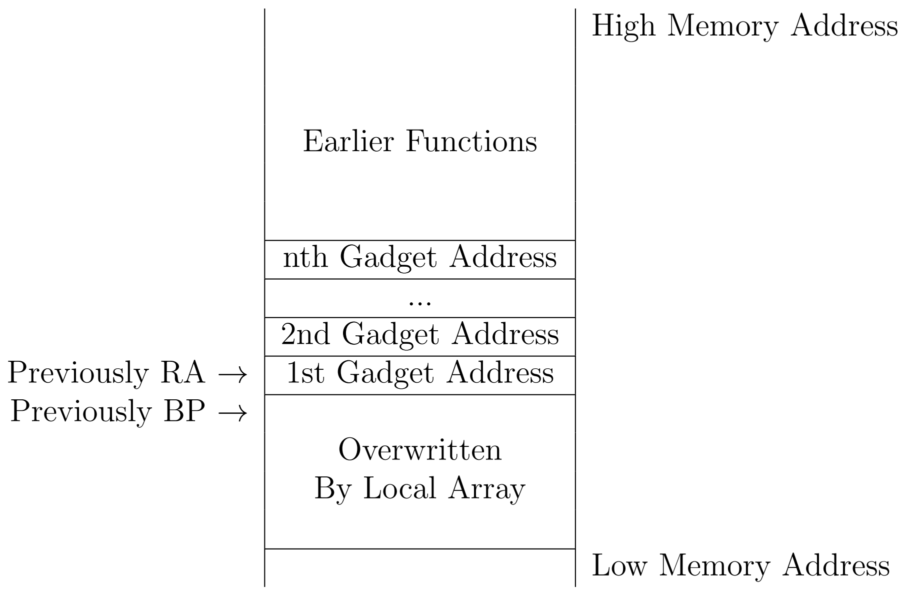

Figure 2: Example Prepped ROP Stack

The return instruction pulls an address off the stack and starts executing other instructions from there. Therefore, if several addresses are placed consecutively in memory as shown above, and 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 6.1.

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 a.k.a. words with the useless information removed i.e. whitespace and comments. The tokenised version of above is therefore [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 6.1 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 any semantic errors in the program structure that was given by the parser. For example the programmer might try to assign True to a variable that holds an integer. Obviously for safety reasons the 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 can operate 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 6.1. This language is a modified subset of the C language that contains only the bare bones needed to perform the buffer overflow exploit that begins 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 enforced like they are in other architectures such as MIPS [ref needed].

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.

1.3 Objectives

A primary objective of this project is to first develop a front end that is a Lexer and Parser for the language given in 6.1. This will be done using the lexer generator JFlex coupled with the Java CUP parser generator.

Along a similar vein another primary objective of this project will then be to implement the type checking and code generation phases. The language of choice for this will be Scala.

Given the implemented compiler, the next primary objective is to then write a program in the language that will perform the buffer overflow exploit. With this initial vector the objective is to then get the ROP exploit running by first manually inserting the gadget code into the program at the low level.

If there's time left within the project a few secondary objectives will be explored. For example:

1. Removing the crutch of manually inserting gadgets and instead finding them by searching through libraries using the Galileo algorithm [ref needed].

2. Automating the process of making Gadget chains by implementing a Gadget compiler. See related work [ref needed].

3. Defeating the ASLR defense to get the exploit to run consistently without it disabled. [ref needed]

4. Implementing a run-time monitor using Intel program statistics to detect when ROP attacks are occurring.

1.4 Motivations

The motivation for choosing Scala is down to a couple reasons. First and foremost, it will be a challenge to learn and build a project in a new language. Secondly, Scala interfaces with Java allowing use of tools previously mentioned such as the Java CUP parser.

The motive for building a compiler from scratch rather than performing the exploit in a pre-existing language is likewise for two reasons. Firstly, in creating my own compiler it allows a greater deal of control at the low level than what is normally available. Additionally, building a compiler adds complexity to what would otherwise be a shorter and less complex project.

1.5 Project Relevance

Essentially this project not only builds upon the Compilers course from last year, but also somewhat ties into the current Computer Security module. This project will also demonstrate my proficiency in areas such as compiler design and knowledge of exploits.

2 Software Engineering Considerations

It goes without saying that a private GitHub repository will be used to track changes and commits through out this project. That aside, in terms of testing the compilation output for correctness, this will be done in an emulated x86-64 environment such as: `https://www.unicorn-engine.org/` . Essentially an oracle style of testing will be used, wherein pre-computed results of equivalent programs written in other languages will be compared with the actual output of my language in the emulator.

For the actual live testing of the buffer overflow and ROP exploits, as previously mentioned that will be done on a virtual Linux image using any of the major virtualization providers e.g. `https://www.virtualbox.org/`

With regard to self evaluation of the project's success/ quality, there are a few key metrics that will be used. First of all, the correctness of the compiled output and the type checker matters a great deal. Secondly, the amount of crutches needed to execute the ROP exploit will also weigh heavy. Finally, whether extensions on the project are completed will also be taken into consideration in my evaluation.

3 Professional and Ethical Considerations

This project although will provide explanations of how buffer overflow and ROP exploits work, will not be about finding a live vulnerability in a real program. The exploit will be performed within my own privileged program on a virtualised instance, therefore about demonstrating knowledge of how these attacks work and how they would be carried out in practice.

With the use of an actual live exploit, I believe to remain ethical would require one of two things. Either, proper fore-disclosure to the affected company some time before the final release of my report in order so that they can patch it. Alternatively, purposely using an older version of some software that already has the vulnerability patched. However, even the former in my belief is still questionable as there may be some users that are still using the version. To re-iterate this project is not about either. Additionally, searching out for such vulnerabilities would be in my opinion violating point 2.1/2.2 of the BCS code of conduct as I believe it is beyond my current capabilities [ref needed].

A final point to note about this project is that there are no intended users, neither are there any human participants. Therefore ethical review before such interactions and subsequently interviews is not necessary as there will be none within this project. Even the language developed will not have any other programmer testing other than myself.

4 Requirements Analysis

4.1 Non-Functional Requirements

4.2 Functional Requirements

5 Related Work

6 Appendix

6.1 EBNF Grammar

Main things to note:

1. This language may be subject to slight changes as the project progresses, but the version given here symbolises the core of the language.
2. This language explicitly differentiates between arrays on the stack and ones on the heap.
3. The new and free keywords break from C convention so as not to mislead about malloc usage.
4. This language has the idiom declarations before definitions baked in. This is done for ease of type checking.
5. 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⟩ ‘)’
```

6.2 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;
}
```

6.3 Project Plan

6.4 Interim Logs

6.5 Proposal

6.6 Word Count:

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.