Interim Report

Candidate Number: 164597

November 2019

Contents

1 Introduction

The base goal of this project is to create a toy compiler for a simple language. This will then be used to compile a program written in the language which smashes the stack. The program will be a jumping off point to begin an exploit known as return oriented programming (ROP).

1.1 Aims

One of the primary aims of this project will be to implement a compiler for the language given in section 8.1. 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 [6].

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 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.

The rational behind disabling ASLR in the early phases of this project is to ensure that gadgets consistently sit in the same spot in memory. Address Space Layout Randomisation (ASLR) essentially randomises the entire layout of the program each time it is run, so if the buffer overflow exploit is to be run consistently the gadgets that are in memory must be consistently referable.

1.2 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 8.1. 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 [6].
2. Automating the process of making Gadget chains by implementing a Gadget compiler. See related work section 6.
3. Defeating the ASLR defense to get the exploit to run consistently without it disabled [4].
4. Implementing a run-time monitor using Intel program statistics to detect when ROP attacks are occurring.

2 Problem Area

2.1 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. Please note that this section is modeled with a standardized stack frame layout in mind, buffer overflows can vary depending on the layout a compiler chooses.

The figure below shows a simplified view of what a function with one local array variable would look like in memory [3]. 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.

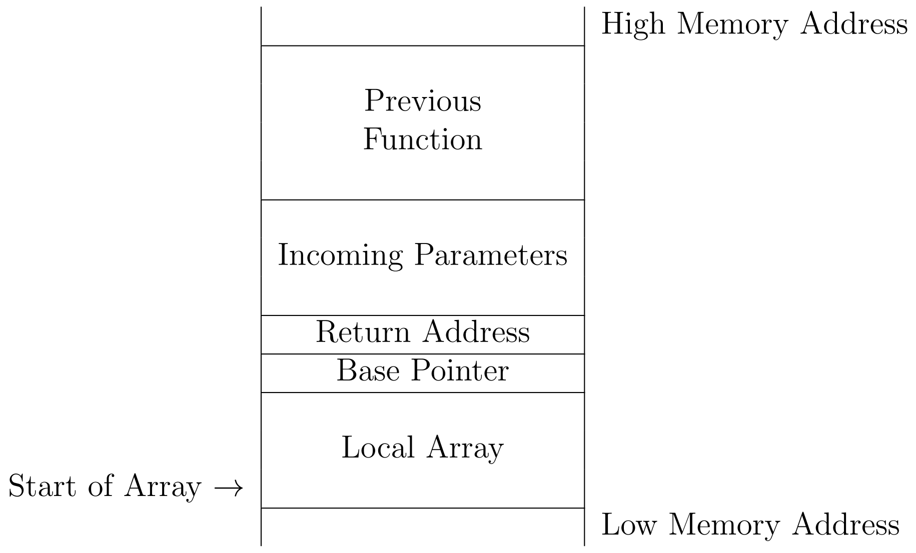


Figure 1: General Stack Frame Layout

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.

2.2 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) [5].

2.3 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.

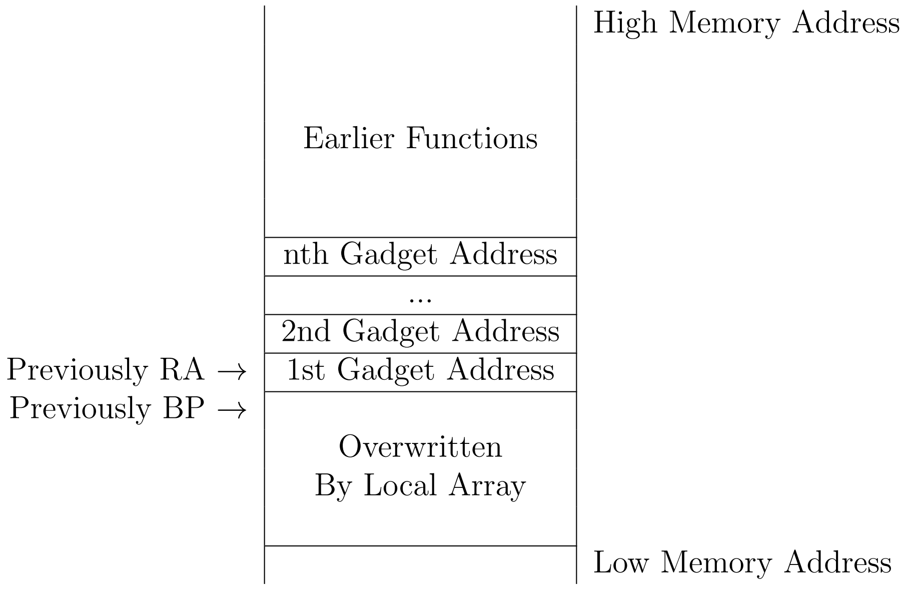


Figure 2: Example Prepped ROP Stack

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.

2.4 Gadgets

To further understand how ROP works, it is worth talking about how the gadgets perform this arbitrary code execution. However, before this the stack pointer (SP) needs to be introduced into the picture. The stack pointer marks the boundary between what memory is in use i.e. the higher addresses and what is garbage i.e the lower addresses. In figures 1 and 2 the stack pointer would be at the same position as the start of the local array.

In actuality what the return instruction is doing is taking the address at the SP and then moving it up by one. Just to note, the compiler will always ensure that the SP sits at the RA when the function ends, so it won't try and use the start of the array as an address.

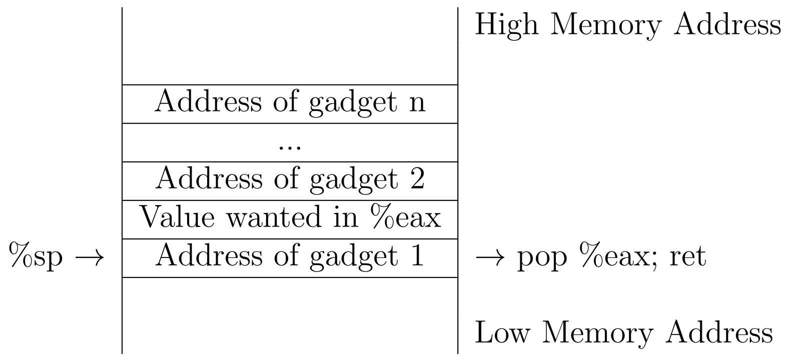


Figure 3: Gadget to load constant into register

A typical operation that code might want to perform is to load some constant into a register, this value could then be added to, subbed, etc. The example above from [6] will highlight how such an operation is done and give a flavor as to how gadgets have to make use of the SP.

In this example the primary gadget consists of the sequence: "`pop %eax; ret`", when this gadget starts executing the stack pointer will be pointing at the value above. The pop instruction does a similar job to return, however instead of jumping to the value, it stores it in a register. So with the execution of the pop instruction the original goal was accomplished, and as always the ending return starts the next gadget. To perform more complex operations more intricate designs are needed, and what can be done is limited by what gadgets are available to be addressed. For example, it is relatively easy to add to a register but to store that result in memory might be tricky, since fine grain control of where values are put might not be available. Although storage at an annoying offset from the SP might be.

2.5 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 8.1.

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 8.1 and automate this process. In this project the Java CUP parser (`http://www2.cs.tum.edu/projects/cup/`) 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.

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.

2.6 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.

2.7 Project Relevance

Essentially this project not only builds upon the Compilers course from last year, but also directly ties into the current Computer Security module. Both of these modules represent major sections of Computer Science and work within them is highly relevant to this course and my degree. Additionally, this project will also demonstrate my proficiency in areas such as compiler design and knowledge of exploits. Such areas are of direct interest to myself and are sought after in the areas in which I hope to work.

3 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 live testing of the buffer overflow and ROP exploit, as previously mentioned, this 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 weigh heavily. Finally, whether extensions on the project are completed will also be taken into consideration in my evaluation.

4 Professional and Ethical Considerations

Although this project will provide explanations of how buffer overflow and ROP exploits work, it 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 it will be about demonstrating knowledge of how these attacks work and how they would be carried out in practice.

If an actual live exploit were to be carried out, it would require one of two things in order to be ethical. Either, proper fore-disclosure to the affected company some time before the final release of my report would have to be made in order so that they can patch it. Alternatively, an older version of some software that already has the vulnerability patched would have to be used. However, even the former is still questionable as there may be some users that are still using the version. To re-iterate, the purpose of this project is not about either. Additionally, searching out for such vulnerabilities would be violating point 2.a/2.b of the BCS code of conduct as it is beyond my current capabilities [2].

A final point to note about this project is that there are no intended users, neither are there any human participants. Therefore an ethical review before such interactions and subsequent interviews are not applicable. Even the language developed will not have any other programmer testing it.

5 Requirements Analysis

It is worth saying that an ideal solution for a compiler is one that preserves the meaning of the program it is trying to translate. Additionally, the ideal function that an exploit aims for is to open a root shell. This is because a root shell provides system level privileges and a platform on which the attacker can easily do anything on the system. Therefore the former points should be reflected in the requirements below.

Just to note, what must be achieved in the project time frame is given by the primary objectives in 1.2. As a recap, it comprises the actual compiler implementation, a program for performing buffer overflows, and a manually compiled gadget chain that's to be inserted using the former.

5.1 Functional Requirements

F1	The compiler shall use a simple register allocation scheme, in which it first targets an unlimited register machine and then switches to an accumulator strategy when only a few registers are left.
F2	The correctness of compilation output shall be rigorously tested and hold in almost every case.
F3	The compiler should operate directly on the AST. No Intermediate Representation needs to be used in the initial phase of the project.
F4	The type checker should catch most if not all semantic errors made within any given program. This shall be tested on many correct and incorrect example programs.
F5	The ROP exploit should open a root shell within the virtualised operating system chosen.
F6	The language developed shall allow for incoming arguments on the command line to be used within the program.
F7	The compiler shall be targeting its output for the x86-64 assembly language. This shall require an assembler to be used.

5.2 Non-Functional Requirements

NF1	The implementation for the compiler shall be written in Scala.
NF2	The development of the compiler shall be done using the Intellij IDE.
NF3	The compilation output should be correct with regard to the computation that is specified by the programmer.
NF4	The front end of the compiler shall utilise the JFlex scanner and the Java CUP parser for its implementation.
NF5	The exploit shall be targeted to run on a Linux distro such as Ubuntu which is virtualised.

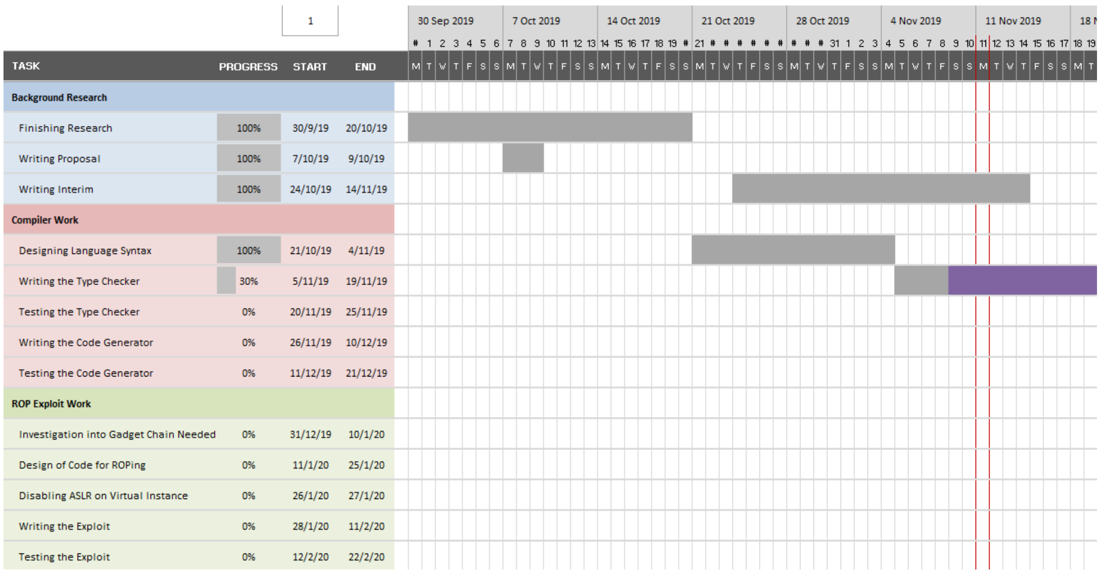
6 Related Work

With regard to related works there are a couple. First off, credit goes to the original team that invented/ discovered the exploit and their paper on the subject [6]. The second work to be discussed is a smaller project found on GitHub at: `https://github.com/Speedi13/ROP-COMPILER`. It essentially is a ROP assembler, i.e. it takes in assembly language and spits out the gadget chain that implements the assembly written. In addition it also analyzes which instructions are implementable given the gadgets available, and allows the user to bring their own into memory. This automation of the compilation of gadget chains is something that hopefully will be explored in the extensions to this project.

Project Contributions:

Generally the language developed here will contribute to the overall field of programming languages, as well as further my own understanding and skills in compiler and language design. Additionally, further contributions could come from the development of a ROP monitor highlighted in the secondary objectives section 1.2. As this could be extended upon and used in systems such as anti-cheat engines. Furthermore, extensions on the ROP assembler again mentioned in section 1.2 could contribute by adding new instructions to the assembly given and allow for more efficient gadget chain translation.

7 Project Plan



TASK	PROGRESS	START	END
Background Research			
Finishing Research	100%	30/9/19	20/10/19
Writing Proposal	100%	7/10/19	9/10/19
Writing Interim	100%	24/10/19	14/11/19
Compiler Work			
Designing Language Syntax	100%	21/10/19	4/11/19
Writing the Type Checker	30%	5/11/19	19/11/19
Testing the Type Checker	0%	20/11/19	25/11/19
Writing the Code Generator	0%	26/11/19	10/12/19
Testing the Code Generator	0%	11/12/19	21/12/19
ROP Exploit Work			
Investigation into Gadget Chain Needed	0%	31/12/19	10/1/20
Design of Code for ROPing	0%	11/1/20	25/1/20
Disabling ASLR on Virtual Instance	0%	26/1/20	27/1/20
Writing the Exploit	0%	28/1/20	11/2/20
Testing the Exploit	0%	12/2/20	22/2/20

Figure 4: Project GANTT Chart

The GANTT chart shown above is a smaller view of the full plan, further details are in my own larger Excel workbook. Although all dates are shown, please note that this plan is subject to change and delays.

Essentially the plan given above is comprised of 3 main stages. First of all is the background research, this was mainly research into compiler implementations [1] and how the ROP exploit works. This first stage was cut short in the second meeting and would of otherwise been longer. The next and current phase is the implementation phase, here this is subdivided into several sections according to section 1.2 and is expected to be completed by late December. The third and final phase is about the exploit, here design of the gadgets/ buffer overflow will take place and will be tested. This final phase is expected to be completed in February of next term. An additional buffer is given on-top of these phases to allow for overruns and unexpected delays.

Note: The first stage also includes the time taken to write the proposal and this report.

8 Appendix

8.1 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 follows those in standard 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⟩ ‘)’
```

8.2 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;
}
```

8.3 Interim Logs

Meeting #1 (15/08/19): This meeting was over the summer holidays and served as an initial discussion about the project. I first proposed to do a project relating to honey potting, however this idea was advised against. The rational being it relies too heavily on luck of what you trap and subsequently must analyse. Recommended the current project relating to ROP

Meeting #2 (2/10/19): This was the first meeting in term it mainly about my initial progress on the project. We discussed my research into the topic, and I mentioned was looking into Liveness Analysis and Register Allocation. He mentioned that those topics were more complex than what is needed in the implementation for my compiler. Suggested that I end the research here and begin implementation of the compiler using a simpler code generation scheme for dealing with temporary registers.

Meeting #3 (14/10/19): This meeting was a brief one as it was mainly to confirm all the relevant info was contained within my proposal. Not much else.

Meeting #4 (29/10/19): This meeting was mainly about the progress I had made towards my project and to confirm a few things. By this stage I had completed the grammar shown previously in 8.1 and was moving on to making the type checker. We discussed if any changes were needed for the grammar of my language. However, none could come to mind and ultimately he recommended I continue on with the project.

Meeting #5 (08/11/19): Again, this meeting was another brief one as it was mainly for seeking some advice about formatting. Not much work towards the type checker had been made at the time.

8.4 Proposal

The proposal was written up using Microsoft Word and has been appended to the end of this document.

References

[1] Andrew Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 2nd edition, 2002.

[2] BCS. Code of conduct for bcs members. `https://www.bcs.org/upload/pdf/conduct.pdf`, June 2015. [Online; accessed 11-November-2019].

[3] 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].

[4] S. Hovav, P. Matthew, P. Ben, G. Eu-Jin, M. Nagendra, and B. Dan. On the effectiveness of address-space randomization. In *Proceedings of ACM CCS*, 2004.

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

[6] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of ACM CCS*, 2007.

Word Count: Approx. 3056

Working Title: Compiler for Return Oriented Programming

Student Name: ***Peter Samuel***

Supervisor Name: ***Martin Berger***

Aims and Objectives:

In brief, the overall aim of this project is to create a toy compiler for a simple language which can be used to smash the stack and begin an exploit known as Return Oriented Programming (ROP) [1].

Aims:

One of the main aims will be to build a compiler for most likely a stripped-down version of C. This version will contain only the bare bones needed to write the ROP exploit. That'll involve removing features such as: Typedefs and Macro pre-processors as they aren't overall necessary.

For the implementation of the compiler, Scala will be the language of choice. This is choice is primarily so that pre-existing work in Java (compiler project in [2]) can be reused easily, and new tools such as parser generators won't have to be learnt.

Another aim will be to target the compiler to output code for the x86 architecture. This is choice is made since x86 is the easiest to exploit for ROP. This is because fixed boundaries on instructions aren't enforced like they are in other architectures such as MIPS [1].

Additionally, the codegen strategy will most likely targeting a register machine that handles spilling by switching to an accumulator machine. This will be done since the efficiency gained by a proper register allocator and liveness analysis is not necessary and hence will be relegated to the extensions section.

A final aim is to then target this exploit to run on a virtualised Linux distro. This distro will have to lack ASLR, so the exploit can execute with the guarantee that the gadgets remain in the same position in memory. This will require a short investigation into potential candidate distros.

Primary Objectives:

So, to summarise from the aims, the objectives then are to:

1. Formulate a stripped-down version of C, with enough functionality to enable the exploit to be written.
2. Develop the main components of the compiler targeting the x86 architecture:
    - Lexer
    - Parser
    - Type Checker
    - Code generator
3. Write the exploit in the newly created language, this will involve:
    - Initially manually inserting the code to begin ROPing.
    - Designing a short gadget chain which will open a root shell.
4. Demonstrating the Exploit works on a Linux distro, presumably with no ASLR.

Extensions:

Possible extensions to this project include the following:

- Implementing the Galileo algorithm, and no longer relying on manually inserted gadgets.
    - Possibly implementing a gadget compiler.
- Investigations into defeating ASLR [3]
- Implementing a proper register allocator to the language and introducing some optimisations.
- Implementing run-time systems such as:
    - Garbage collector
    - Return count monitor for detecting ROPing

Relevance:

- Essentially this project is an extension on the compilers course in my second year 2, building a working albeit toy compiler will demonstrate my abilities in such areas.
- Challenge me to learn a slightly different language: Scala
- Ties into what I want to go into: pen testing and to some extent the computer security module.

Resources Required:

Minimal:

- A copy of the Tiger book [2]
- Access to the mentioned papers in the bibliography
- A running version of an Linux distro w/o ASLR [4]

All the above I currently possess or could easily get access to w/o funding.

Bibliography

[1] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and communications security*, Alexandria, Virginia, 2007.

[2] A. Appel and J. Palsberg, Modern Compiler Implementation in Java (2nd Edition), Cambridge University Press, 2002.

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu and D. Boneh, “On the Effectiveness of Address-Space Randomization,” in *Proceedings of the 11th ACM conference on Computer and communications security*, Washington DC, 2004.

[4] Adrián, “How Effective is ASLR on Linux Systems?,” 03 February 2013. [Online]. Available: https://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/. [Accessed 14 October 2019].