CSUDH

CALIFORNIA STATE UNIVERSITY, DOMINGUEZ HILLS

Spring 2020

Senior Project Report

Computer Science Department

California State University, **Dominguez Hills**

Software Reverse Engineering

Prepared by

Jocelyn Luna Dye

In
Partial Fulfillment of the requirements
For
Senior Design – CTC 492

Department of Computer Science
California State University, Dominguez Hills

Spring 2020

Committee Members/Approval

Faculty advisor	Signature	Date
Committee member	Signature	Date
Committee member	Signature	Date
Dr. Mohsen Beheshti Department Chair	Signature	Date

Abstract

“What Is Reverse Engineering? Reverse engineering is the process of taking something apart and putting it back together again in order to see how it works, so that a developer can study the code and learn how it works. As a tool for someone learning to program, this is invaluable” [1]. This is important going forward to understand when looking at software Reverse Engineering mostly reading and figuring out CPU operation codes and their relationships in the program can be made, slowly understanding overall logic of a program. That can be used to find specific vulnerabilities or bugs that can be patched.

Acknowledgement

I would like to acknowledge Garrett Poppe for his guidance, support, and help with guiding my education and my project with the ideas on how to execute it better. And Dr. Mohsen Beheshti his guidance, support, and assistance.

Table of contents

INDEX OF FIGURES

1. Introduction

1.1 Purpose

The purpose of this paper is to try and clearly explain what reverse engineering is and how software programs can be exploited, and their relationship. It will be using visual aids of screenshots of various programs being used to reverse engineer and explain them. Programs I have developed will be used to show case various types of exploits.

1.2 Goals

The goal of this project is to give an in-depth analysis on how to read a program that's broken down in operational codes, that can be read in as assembly level language. To map out the functions, logic, etc. of the program. To spot and exploit different type of vulnerability, and how to patch it. This will be done on programs I have created and on a real program if there is time.

1.3 Motivations

I always loved programing, just creating programs, figuring out better ways to optimize my logic. To have a passion to learn the lower levels of how computers work. In this day and age, with develop methods such as agile. Bugs and problem are bound to happen, especially if developers do not have enough time to test and debug on top of that. There is going to be a vulnerability that can range to annoying for a user to a catastrophic breach in security. It is important to find such problems and help find solutions to it, have it patched and informing others to not make this mistake.

1.4 Contributions

For contributions I will be programming a suite of programs that will have a specific vulnerability for each one. There are very specific vulnerabilities and are normally called a "Crack me". The Programs will be used to; show how to debug a program, map out and look for vulnerabilities and show various types of exploits. If there is time, I might attempt to do this on a full real program. I would be using the same methods and recording the progress and findings.

2. Background

2.1 Tools, Definitions, Acronyms and Abbreviations

Library - In computer science, a library is used by computer programs. It is a collection of resources such as documentation, data, and help data. It also contains message templates and pre-written code and subroutines.

Compiler - A compiler functions as a translator. It is a computer program that takes code from one programming language to a difference coded language. The compiler process includes translating from a high level program language to a low level. This results in an executable program.

Linker – To create an executable program, a linker or binder is used to combine object modules. Modules are different code written in program languages separately. Breaking down the programming tasks into simplified modules makes the process more manageable. In order to put together all the modules you need the Linker to accomplish that task.

CPU- The CPU, central processing unit, is the most important component in your computer. It is the central component that manages all instructions giving in a logical way. The CPU will send out instructions back to other components. The CPU can perform calculations and run programs.

Registers - A processor register is a quickly accessible location available to a computer's central processing unit.

Memory - Essential information is stored and available for quick use in a computer in a memory device.

Stack – This is a region in the computer that stores temporary variables. A stack works closely with the CPU moving new variable into and out of the stack. Once a stack variable exits that portion of the memory is available.

Heap –This is a free floating region of memory. It is not managed automatically and you need to allocate and deallocate that memory when it is not needed. Memory leaks will occurs if it does not follow the deallocate procedure.

OllyDebug - OllyDebug is a reverse engineering tool, a x86 debugger that emphasized binary code analysis. It is used when the source code is unavailable. The results of using the OllyDebug tool is that it can trace registers, switches, tables, and strings. It also can locate routines from object files and libraries.

Crack me - Reverse engineering skills are tested by using a program called crackme.

Disassemblers – Disassemblers is a tool that changes binary to a readable language. It also can extract strings, libraries, and imported and exported functions.

Debuggers – A debugger is a computer program that programmers use to test and debug a targeted program. This program allows the analyst to view and edit line by line. The process can discover incorrect code.

Hex Editors – A Hex Editor views and edits binary files. A binary file contains machine readable while text file is readable. With Hex editors you can edit raw data in a file. There are three areas in the hex editor: an address area, a hexadecimal area and a character area.

PE – Windows binary

ELF – Linux binary

PE and Resource Viewer –The process to setup and initialize a program is processed using a binary code which runs on a windows machine. A portable executable supporting the DLLs is essential for all programs.

Ethical hacking – An attacker that owns or has permission over a network, program, system etc. In the express intent to test on and find vulnerabilities on.

2.2 Overview into reverse-engineering

"The main purpose of reverse engineering is to audit the security, remove the copy protection, customize the embedded systems, and include additional features without spending much and other similar activities" [2] with little or no source code or materials on the system. In order to become skilled in the field of software design and testing, it is recommended that having a strong understanding of assembly level language is essential. Software reverse engineering, a method of analyzing a binary that is readable as assembly level language. Either static, when it is not running or live when it is able to step though the logic line by line with more information about the system the program is running on. CPU operation codes is readable to humans as assembly code. Functions, methods, memory and register state, etc. and their relationships in the program can be made. Slowly constructing functions and the overall logic of a program.

2.2.1 Reverse Engineering in Software Development

Reverse Engineering involves taking reversing the machine language back to something that is readable to humans. Using this software gives developers the ability to add new features, to patch software fixing weaknesses and bugs in the program.

2.2.2 Reverse Engineering in Software Security

Programmers that battle virus attacks use reverse engineering techniques to study the virus and malware code. This gives programmers the ability to test the system for vulnerabilities, security flaws, backdoor installation, etc. that an attacker can create. The process requires experience and specialized skills to study and analyze virus code.

3. Analysis

Analysis of reverse engineering is important for examining the program, and how the machine runs it depending on the architecture. There are two major types of analysis; Static, which is when the program is not running, and Dynamic, which is when the program is in run time. Both have their differences, but it is recommended to use both, respectively.

3.1 Static

Static analysis as mentioned earlier, is the when the program is not in run time, the CPU isn’t executing instructions, values in the registers and memory are not being stored, etc. But this can be useful when trying to understand a program before executing it as it might have built in functions that make it harder to read while it is running.

3.1.1 Decompilation

Decompilation is the process of using a decompiler to recreate the source code in a higher level language with the given operational codes or bytecode. This method relies on other data to help recreate source code with different results.

3.1.2 Disassembling

Disassembly uses a disassembler to take raw machine language, normally called operational code or even bytecode. Being readable as Assembly, with the help of machine-language mnemonics. This can work with any CPU architecture but can take a bit of time, more so when the analyst is not very knowledgeable with operational codes. The Interactive Disassembler by hex-rays is a tool that is able to generate ASM code from operational codes.

3.1.3 Effect systems

Effect systems are designed to help show the relationships and effects of execution a function can have. This effect shows what will be done and with what in the program.

3.1.4 Control-flow

A control-flow analysis is a way to collect information on a program and what methods can be called at different points in a program when it is in run time. In doing this an analyst can find logical errors that a program can run into without any checks or preventive measures that can be led to huge vulnerabilities.

“•Defects that result from inconsistently following simple, mechanical design rules

•Security vulnerabilities : Buffer overruns, unvalidated input...

•Memory errors : Null dereference, uninitialized data...

•Resource leaks : Memory, OS resources...

•Violations of API or framework rules : e.g. Windows device drivers; real time libraries; GUI frameworks

•Exceptions : Arithmetic/library/user-defined

•Encapsulation violations : Accessing internal data, calling private functions

•Race conditions : Two threads access the same data without synchronization”[6]

The information that was obtained can be visually represented to humans as a flow chart called a control flow graph (CFG). This shows where instructions of the program are represented by nodes and edges represented the flow of control of the program. When analyzing, code blocks and loops, a CFG can clarify how a program will behave, and help lead an analyst to a specific vulnerability or bug or where to look. The Graph in Figure 1 is based on jumps for branching logic. Commonly used in loops and conditional statements.

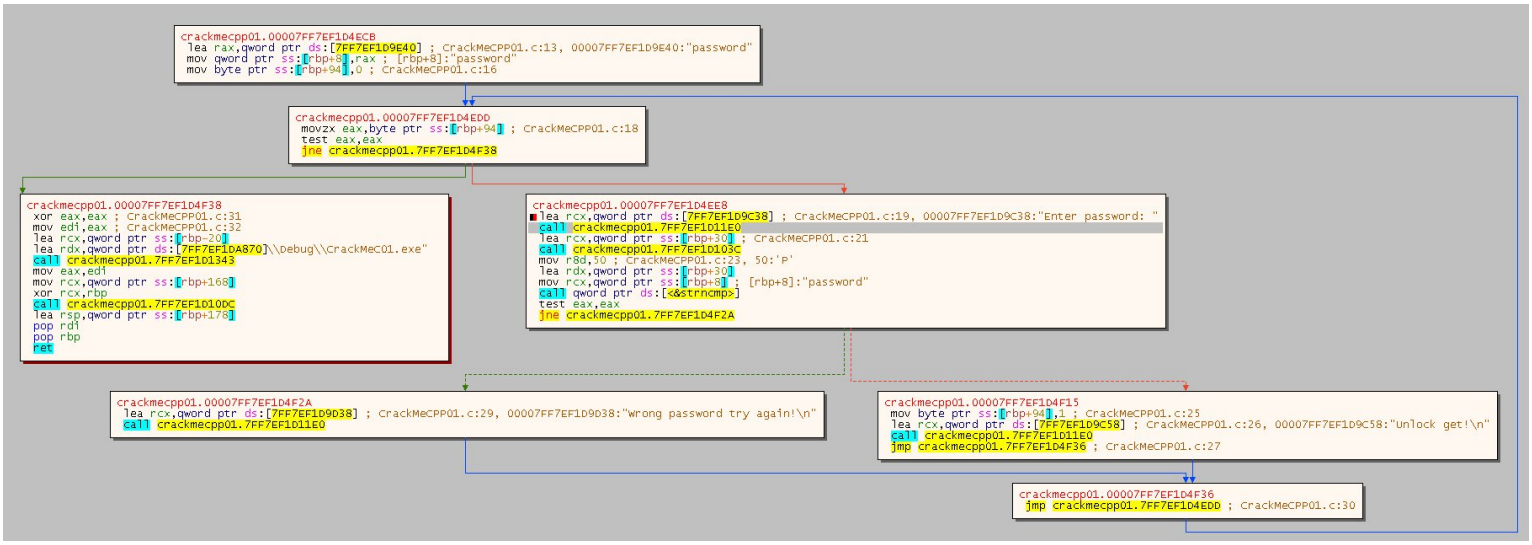


Figure 1: CrackMeC01 Control-flow

3.1.5 Model checking

When referring to model checking, it is the process automated ways of checking if a model is correctly behaving with it is given specification. The specification and the logic of the program in the form of a formula. From this it is possible use algorithmic methods to check if the logic of the program matches up to the given specifications.

3.1.6 Static Data-flow analysis

Data flow in Static analysis shows how data in a program flow. The type of data in a program can very depended on what operational code will be executed. When going through the process of analysis, it can be done by reading the logic of the program from the syntax/symbols tables “(e.g., "variable X in scope Y is modified at control flow node N")” [5] and logic on syntax constructs based on the logic in other parts of the program (e.g., "X is modified at M, and M postdominates N")” [5]. All to make a prediction how the program will run and produce what data in run time.

3.1.7 Advantages and Disadvantages

Static code analysis advantages:

- When enough code is available, is can be used to find weaknesses in a program at an exact location.
- Can be read by anyone with experience in programing
- Can be a lot quicker to fix a program
- Can be patched later in the development life cycle.

- Specific defects that can be found that dynamic analysis might find little or none of.
    - Unreachable code
    - Variable use (undeclared, unused)
    - Uncalled functions
    - Boundary value violations

Static code analysis limitations:

- It takes up a lot of time when done manually, usually longer then dynamic analysis.
- Tools that automate the process can have false positives and negatives.
- Automated of tools can provide false negatives and positives.
- Automated of tools rely on a set of rules when scanning and can be misused.
- It cannot find any vulnerabilities that come up only when in runtime.

3.2 Dynamic

Dynamic analysis is letting code run its normal course or stepping through it line by line or with set break points on a real system all while monitoring system values in real time.

3.2.1 Debuggers

A debugger is program application that is used to run on a target computer under controlled conditions, that allow the analyst of the debugger step though a programs operation while it is running. This allows the analyst to monitor computer resources like flash and disk drives, and be able to modify memory, CPU registries in order to try and cause the program to behave in ways the developer had not intended to find exploits or bugs that can be used without the debugger. It can run and halt the program at specific points with break points. “Tools commonly used in dynamic analysis include debuggers of all kinds. GDB or WinDbg would be pure debuggers allowing for this. IDA Pro is somewhat of a swiss army knife for the reverse code engineer when it comes to dynamic-analysis as it allows one to use various kinds of debugger back ends, but also Bochs to emulate through bits and pieces of code ad hoc” [3]. The program in Figure 2 called x64dbg is a debugger on a compiled C program.

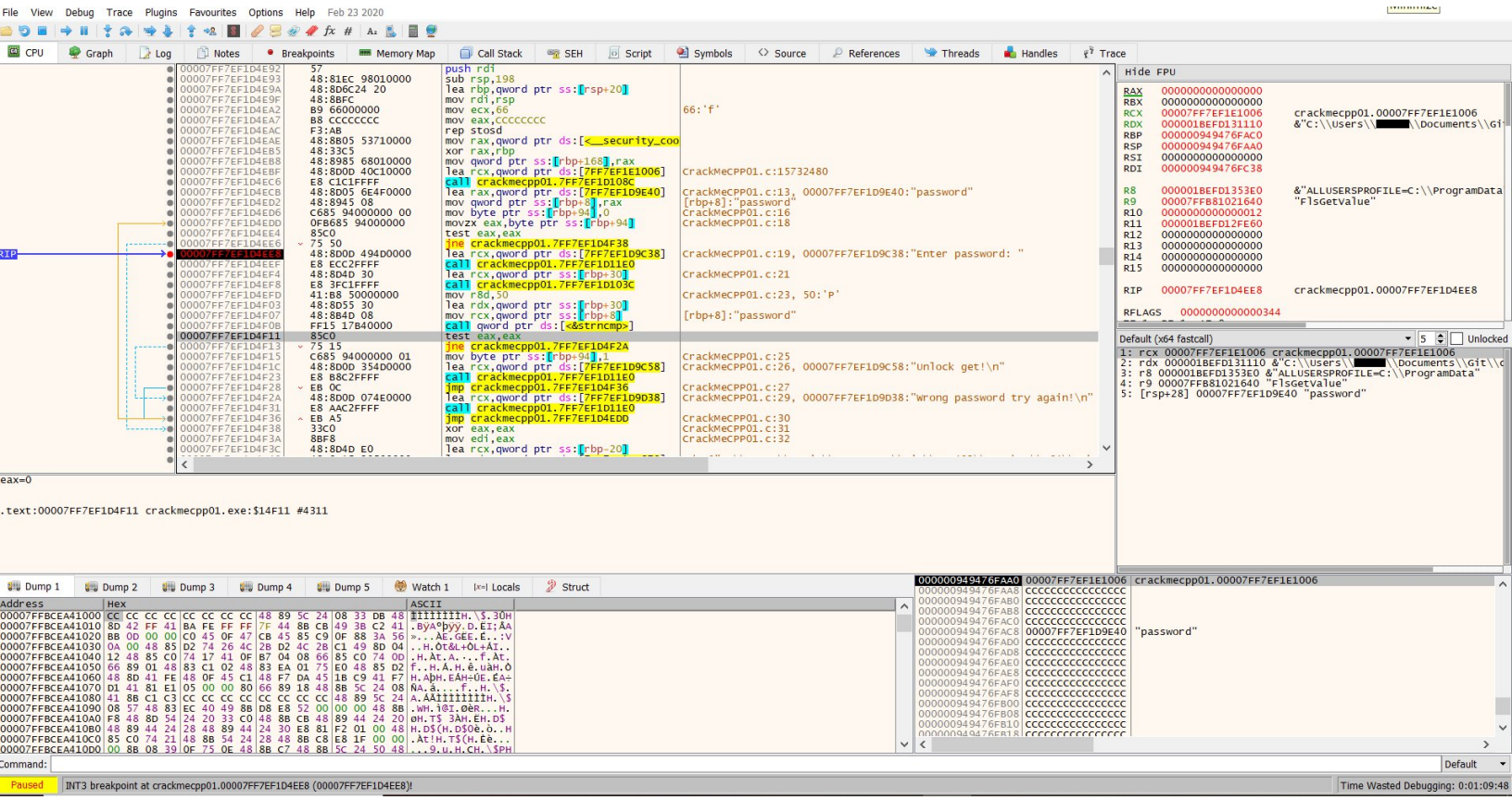


Figure 2: x64dbg Debugger CrackMeC01

3.2.2 Dynamic Data-flow analysis

Dynamic Data-flow analysis is a method for someone to monitor values at different parts of the program and how they change when the program is running. “Existing approaches for performing dynamic data flow analysis for object oriented programs have tended to be data focused and procedural in nature. Dynamic data flow analysis approaches consist of two primary aspects; a model of the data flow information, and a method for collecting action information from a running program” [4].

3.2.3 Advantages and Disadvantages

Dynamic code analysis advantages:

- It can help expose vulnerabilities in a runtime environment.
- Allows for the analysis of programs when you don't have access to source code.
- It can help expose vulnerabilities that could have been flagged as false negatives with static analysis.
- It will give you the information to validate the findings in static analysis
- It can be used against any program that runs CPU operational instructions.

Dynamic code analysis limitations:

- Automated of tools can mislead an analysis that everything can be addressed.
- Cannot always cover a full test coverage like it would with source code.
- Automated of tools can provide false negatives and positives.
- Automated of tools rely on a set of rules when scanning and can be missed used.
- It can take a long time to trace a vulnerability to its exact location.

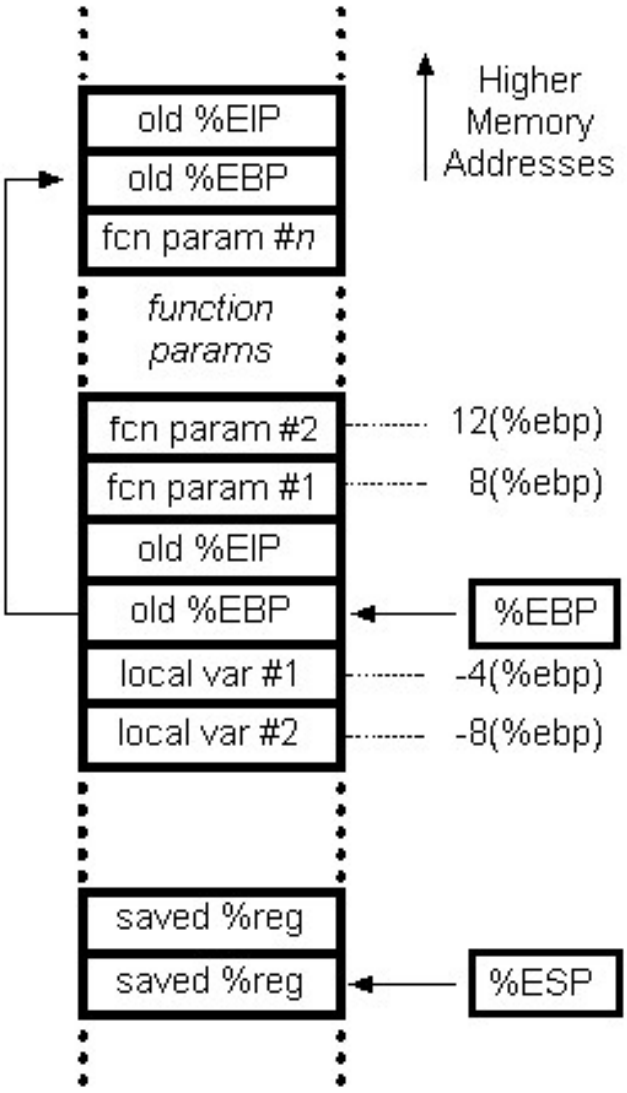


Figure 3 : Diagram of a buffer overflow

4. Methods of exploitation

4.1 Reading Memory

One method of exploitation is simply reading data in memory at runtime. To explain how to read memory. In memory there is addresses that bytes of data can be held per address. To access this random-access memory normally, there is the stack frame, Stack frames help programming languages in supporting recursive functionality for subroutines. Which is between the pointers ESP, the stack pointer and EBP the fame pointer. A pointer carries a value of the address that is being used. So, if a function is called and pushes more values on the stack, it grows the frame. While in runtime This helps nested to recursive calls to the same function , while each call will obtain their own separate frames.

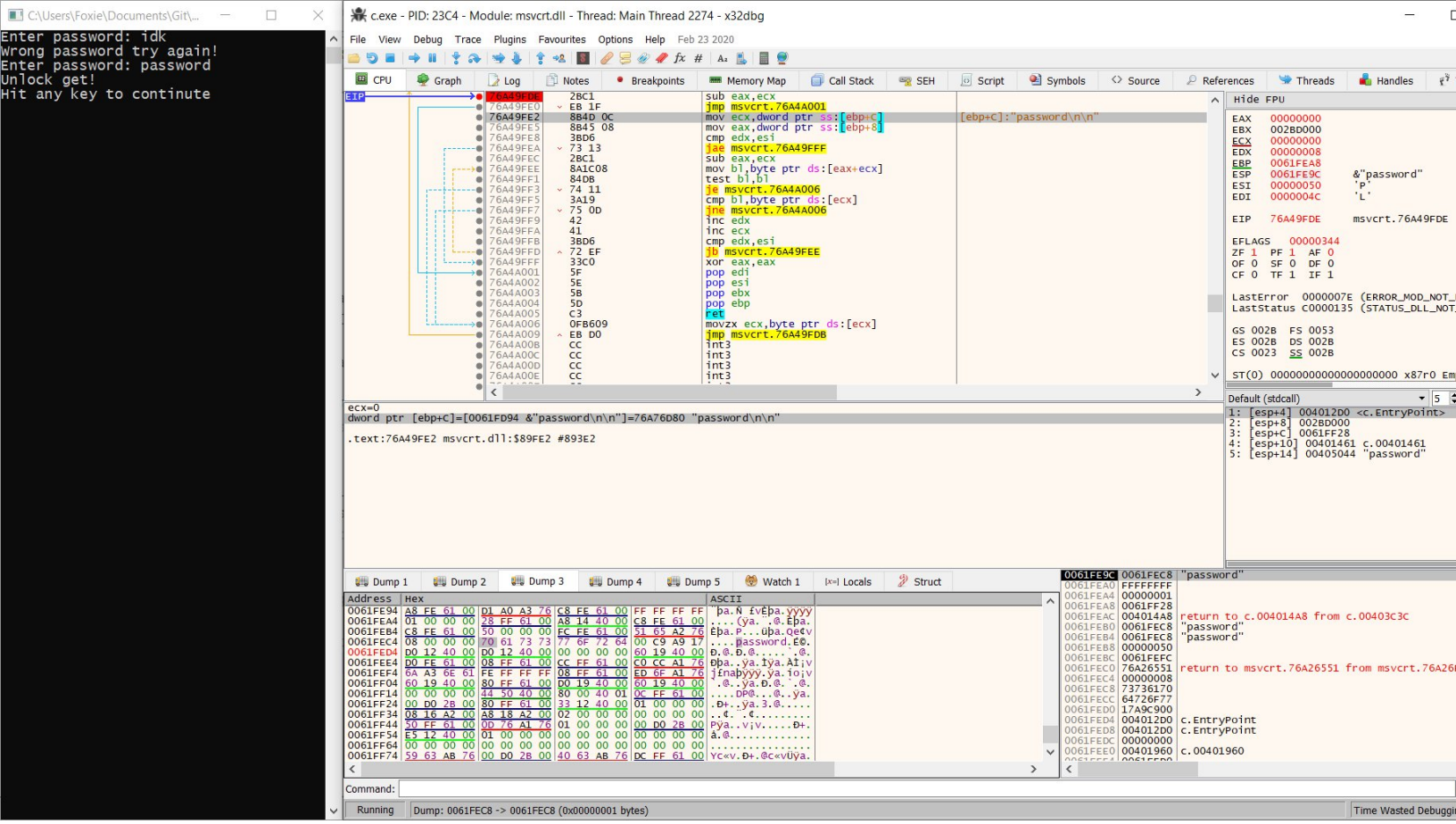


Figure 4 : x64dbg Debugger CrackMeC01

In this simple program, there is an obvious flaw in its security that some simple memory sleuthing can uncover. The debugger pictured in figure 4, shows an instruction at address 76A49FE2 loading a pointer 0061FEC8 into CPU register ECX. This command at 76A49FE5 is repeated for the user input being stored at 0061FEB0 stored into EAX. So, it is easy to see the password in plain text at 0061FEC8. This shows the importance to never have password in plain text, and thus better to use hashing methods when doing mathematical compares between two salted hashes.

Buffer overflow

A classic flaw in security a classic security is when the programmer assumes the user will correctly input what is needed into the input field. With this false assumption an attacker can input more bytes then the allocated space will hold. This holding space is called an input buffer, and this can overflow into the rest of the stack. The software structure, like an array is not being checked on the hardware level as it will be too expensive in resources as it relies on page-level checks. When an attacker puts more bytes then what the input buffer was allocated for. This can happen when this is left up to a software environment (e.g. Java), or a programmer (e.g. C/C++) makes this mistake when a programmer omits this kind of software check because they trust the user and think they fully understand the index within the limits.

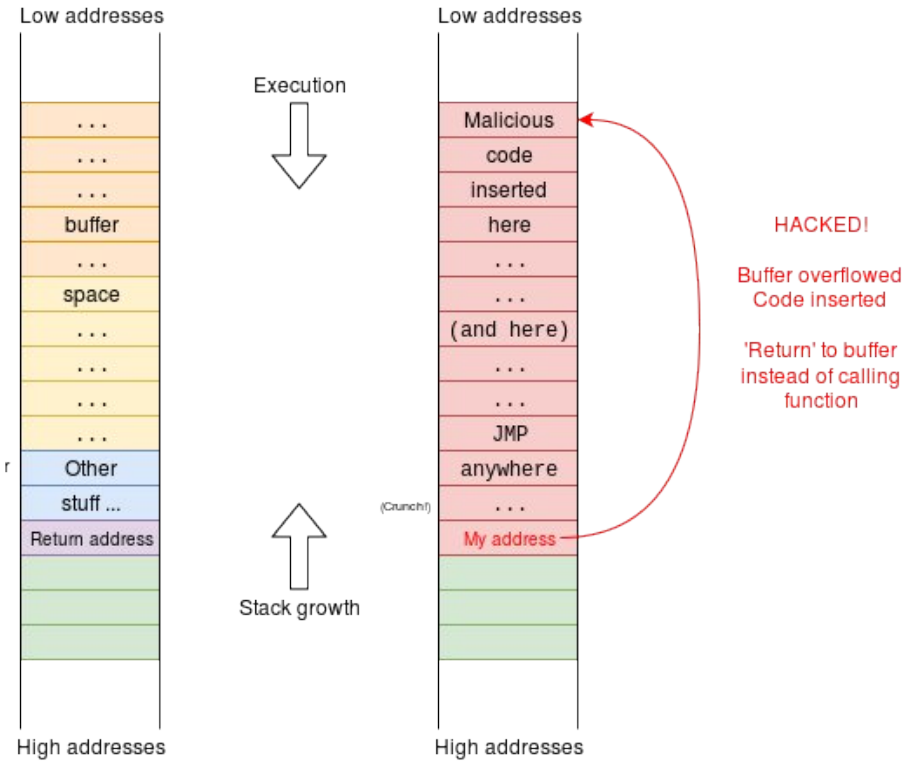


Figure 5 : Diagram of a buffer overflow

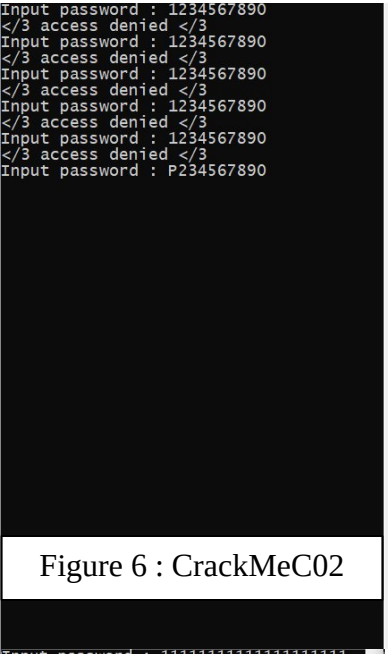


Figure 6 : CrackMeC02

Figure 7 : CrackMeC02

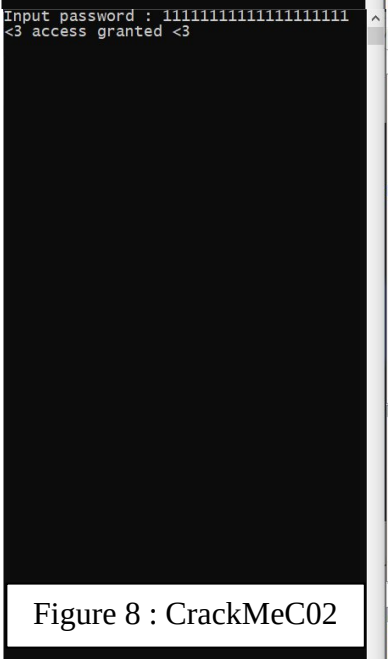


Figure 8 : CrackMeC02

In this crackme example pictured in figure 6-8 has a buffer overflow vulnerability. To note, the compiler normally catches problems like this and had to be modified to have this vulnerability easily explained. But it can still manifest in ways modern compilers do not catch. When a program initializes at runtime, memory is allocated in a stack going in ascending order. So, if in this example, the variable that holds the password gets initialized first then the user input gets initialized on top of the stack. With this, if a max number of bytes that the frame buffer should hold is not set in the program, the user input can go passed the allocated twelve bytes into the next frame buffer, where the password is being held. Even if both memory spaces were encrypted with a hash, the overflow would still overwrite the password information being held.

Remote Code Execution

"A more complex but much more damaging abuse case is using the vulnerability to execute arbitrary code on the server" [7]. The A PHP exploit pictured in figure 9. Is a type of attack where the user inputs a command. Server executes the unsensitized code. This is a more complex attack where the attacker is abusing the ability to execute arbitrary code server side. This is an example of a PHP exploit where the user inputs a command with the query parameter –d. The string is passed though the unsensitized input where the server executes it.

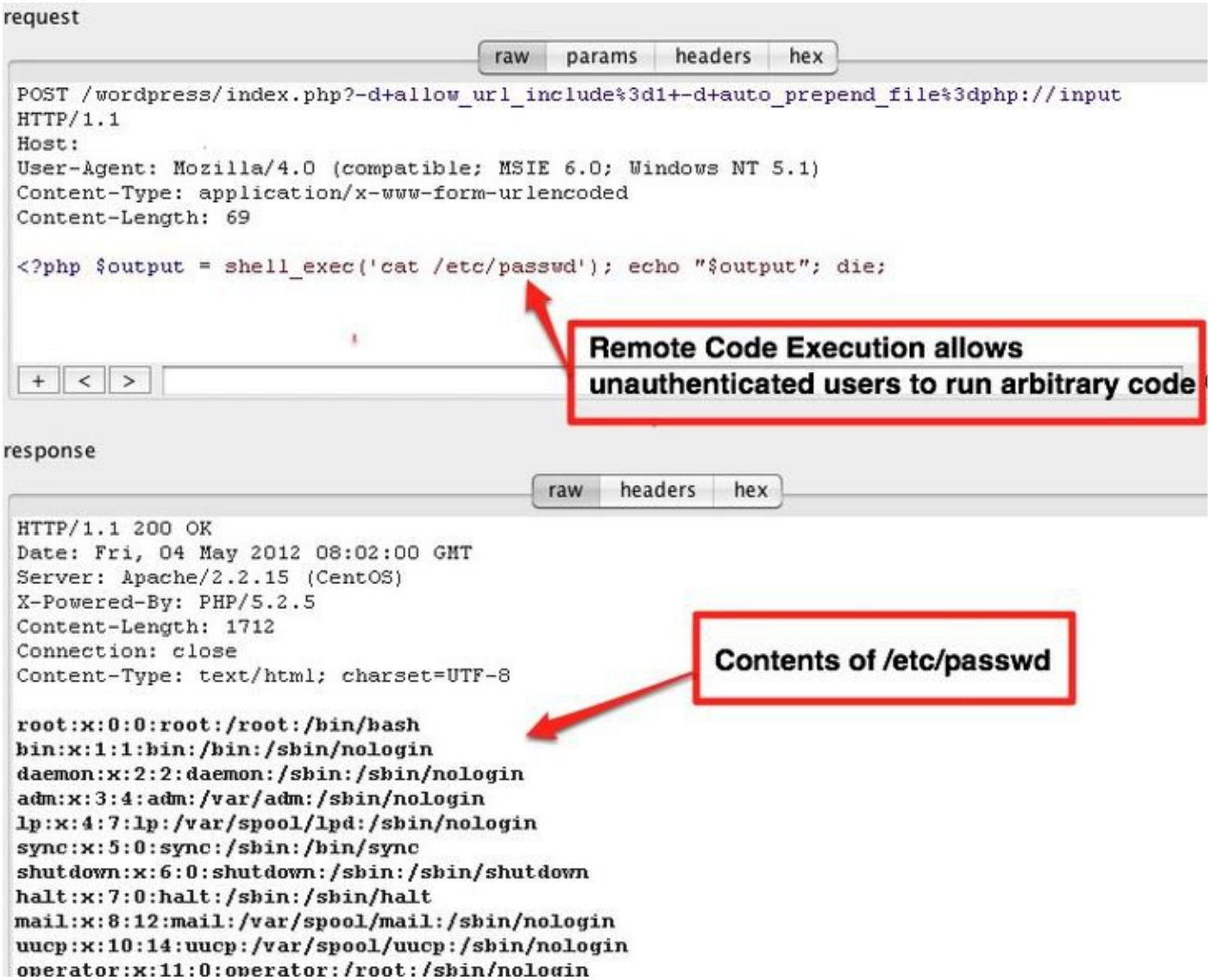


Figure 9: PHP server code injection

5 Conclusion & Future work

5.1 Future plans and goals met

Briefly go over what you plan to do possibly program a few more flawed applications demonstrating more exploits, how to patch a flawed closed sourced program accomplished and was able to explain main ideas of reverse engineering and the impotence of proper dev ops. Was able to program a few crackmes

5.2 Trouble shooting

Issues that that I faced while doing this project and how it affected the result. After a while I switched from Visual studio to just notepad++ and GCC compiler. This was a lot easier to pull and push from my git. Also, to compile the code with parameters to turn off unsafe programing problems the compiler would pick up. To be able to develop an application where I can show case the kind of flaws that come up, just in different ways. I just do not have time to find zero days for this project. Had to research a lot of concepts in detail to be able to purposely recreate them.

5.3 What was learned

What I learned from this project Learned more about reverse engineering , debugging, and programing. If I had more time, I would attempt more types of crackmes showing off vulnerabilities. Used more current examples that are not already checked my modern compiled and interpreted languages. Explained a more granular process of debugging a program.

6. References

[1] B. Engard, 'The Power of Reverse Engineering', 2016. [Online]. Available: https://www.thesoftwareguild.com/blog/what-is-reverse-engineering/. [Accessed: 20- Feb- 2020].

[2] S. Kelly, ‘Reverse Engineering Tutorial: How to Reverse Engineer Any Software’, 2020. [Online]. Available: https://blog.udemy.com/reverse-engineering-tutorial/. [Accessed: 20- Feb- 2020].

[3] Stackexchange.com, ‘About dynamic-analysis’, 2013. [Online]. Available: https://reverseengineering.stackexchange.com/tags/ dynamic-analysis/info. [Accessed: 21- Mar- 2020].

[4] A. Cain, ‘Dynamic data flow analysis for object oriented programs’, 2003. [Online]. Available: https://www.researchgate.net/publication/228554341_Dynamic_data_flow_analysis_for_object_oriented_programs/. [Accessed: 22- Feb- 2020].

[5] Semanticdesigns.com, ‘Control and Data Flow Analysis’, 2018. [Online]. Available: www.semanticdesigns.com/Purchase/Justifications.html/. [Accessed: 23- Mar- 2020].

[6] J. Aldrich ‘Analysis of Software Artifacts’, 2006. [Online]. Available: https://www.cs.cmu.edu/~aldrich/courses/654-sp08/slides/11-dataflow.pdf/. [Accessed: 24- Mar- 2020].

[7] Claudius , Jonathan. “PHP-CGI Exploitation by Example.” May 08, 2012. [Online]. Available: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/php-cgi-exploitation-by-example/. [Accessed 4 April. 2020].

Figure 3: Friedl, Steve. “Intel x86 Function-call Conventions - Assembly View.” [*Unixwiz.net*, http://unixwiz.net/techtips/win32-callconv-asm.html. Accessed 4 April. 2020.]

Figure 5: *The University of Manchester*. “Buffer Overflow.” [*The University of Manchester*, https://xerxes.cs.manchester.ac.uk/comp251/kb/Buffer_Overflow/. Accessed 4 April. 2020.]

Figure 9: Claudius , Jonathan. “PHP-CGI Exploitation by Example.” [Trustwave, May 08, 2012. https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/php-cgi-exploitation-by-example/. Accessed 4 April. 2020.]

7. Appendix

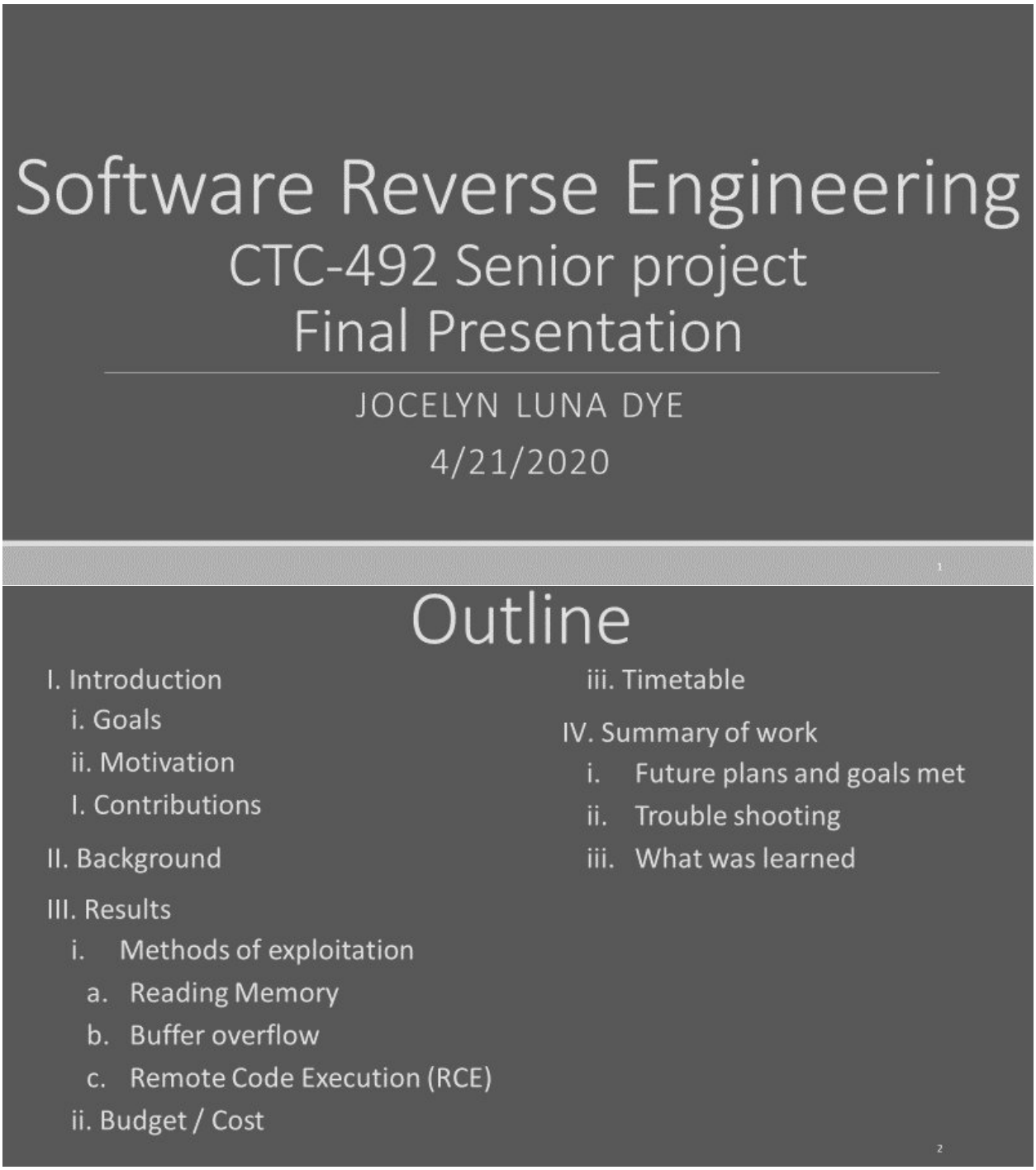
7.1 CrackMeC01.c

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define MAX_LEN 80

int main(){


	char *WowLookAtThisPasswordArray = "password";
	char UserInput[MAX_LEN];

	bool reallyObviousFlag = false;

	while (reallyObviousFlag == false){
		printf("Enter password: ");
		gets(UserInput);

		if (!strncmp(WowLookAtThisPasswordArray, UserInput, MAX_LEN)) {
			reallyObviousFlag = true;
			printf("Unlock get!\n");
		}
		else
			printf("Wrong password try again!\n");
	}
	return 0;
```

7.2 CrackMeC02.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_LEN 8

int main(int argc, char** argv)
{
   volatile int harmlessflag;
        char charPasswordBuffer[12] = "PassW0rd!";
   char charBuffer[12];
   harmlessflag = 1;
        while(harmlessflag == 1){
        printf("Input password : ");
        gets(charBuffer);
                if(!strncmp(charPasswordBuffer,charBuffer, MAX_LEN)){
                        printf("<3 access granted <3\n");
                        harmlessflag = 0;
                }
                else {
                        printf("</3 access denied </3\n");
                }
        }
        printf("Hit any key to continute");
        gets(charBuffer);
}
```

7.3 Project Presentation Slides



Introduction
Goals

Give an in-depth analysis of

- How to read a program in ASM
- Map out Control-flow of program
- Look for and exploit different vulnerabilities
- Reverse engineer crackmes / programs
- How to patch it with outsource code

3

Introduction
Motivation

- Loved programing
- Passion for learning how computers work
- Enjoys breaking the Kobayashi Maru
- Fixing rather then buy new when possible
- Spread awareness

4

Introduction
Contributions

- ✓ Show how to debug a program
- ✓ Show how to look for vulnerabilities
- ✓ Show various types of exploits
- ✓ Recording the progress and findings
- × Patch with out access to source code

Background

- Software Development
  - Software testing
  - Enables the developer to add new features with no source
  - Incorporate new features to existing software

Background

- Software Security
  - Test system does not have any major vulnerabilities
  - To make the system robust to protect it from malwares and attackers.
  - Helps testers to study the virus and other malware code

7

Results : Methods of exploitation

- Not fully completed, was able to explain
- Background
- Analysis
- Develop some flawed application
- Explain some types of exploits

8

Methods of Exploitation: Reading memory

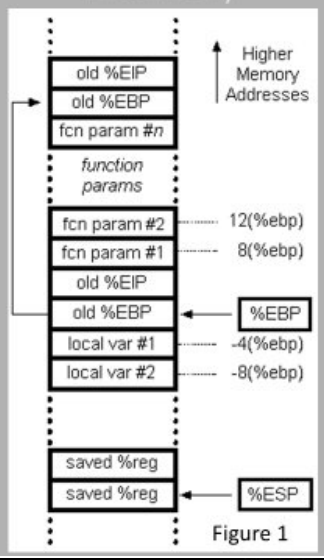


Figure 1

- The stack frame is an allocated section of memory
- Instructions are stored between esp (stack pointer) and ebp (frame pointer)

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define MAX_LEN 80

int main() {


    char* WowLookAtThisPasswordArray = "password";
    char UserInput[MAX_LEN];

    bool reallyObviousFlag = false;

    while (reallyObviousFlag == false) {
        printf("Enter password: ");
        gets(UserInput);

        if (!strncmp(WowLookAtThisPasswordArray, UserInput, MAX_LEN)) {
            reallyObviousFlag = true;
            printf("Unlock get!\n");
        }
        else
            printf("Wrong password try again!\n");
    }
    printf("Hit any key to continute");
    gets(UserInput);
    return 0;
}
```

Figure 2

9

Methods of exploitation: Reading memory

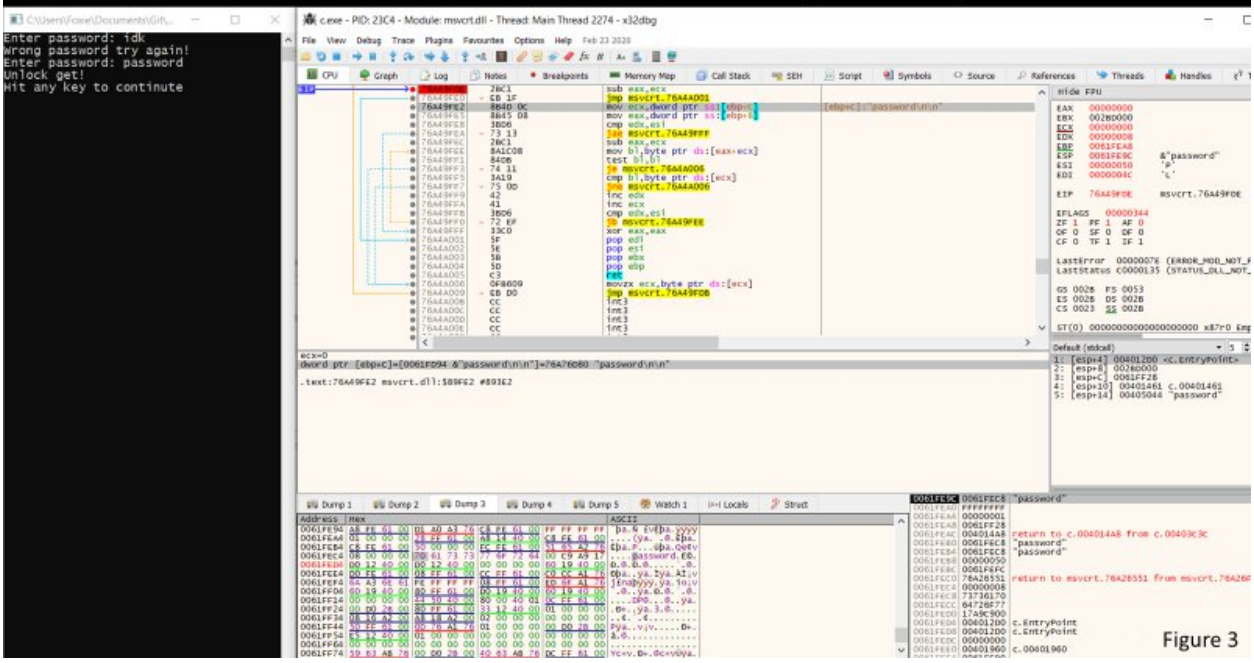


Figure 3

Methods of exploitation: Buffer overflow

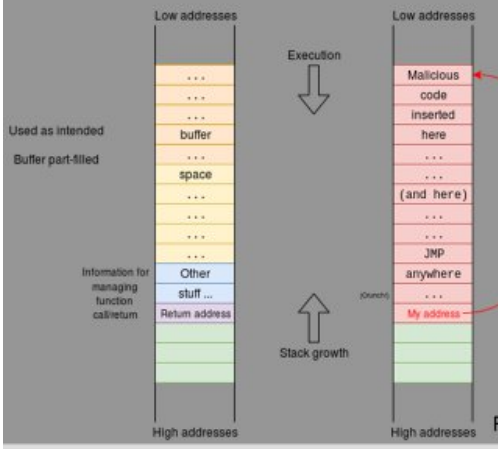


Figure 4

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_LEN 8

int main(int argc, char** argv)
{
    volatile int harmlessflag;
    char charPasswordBuffer[12] = "PassWOrd!";
    char charBuffer[12];
    harmlessflag = 1;
    while(harmlessflag == 1){
    printf("Input password : ");
    gets(charBuffer);
        if(!strncmp(charPasswordBuffer,charBuffer, MAX_LEN)){
            printf("<3 access granted <3\n");
            harmlessflag = 0;
        }
        else {
            printf("</3 access denied </3\n");
        }
    }
    printf("Hit any key to continute");
    gets(charBuffer);
}
```

Figure 5

- When an attacker puts more bytes then what the input buffer was allocated for.
- This can happen when this is left up to a software environnent (e.g. Java), or a programmer (e.g. C/C++) makes this mistake.

11

Methods of exploitation: Buffer overflow

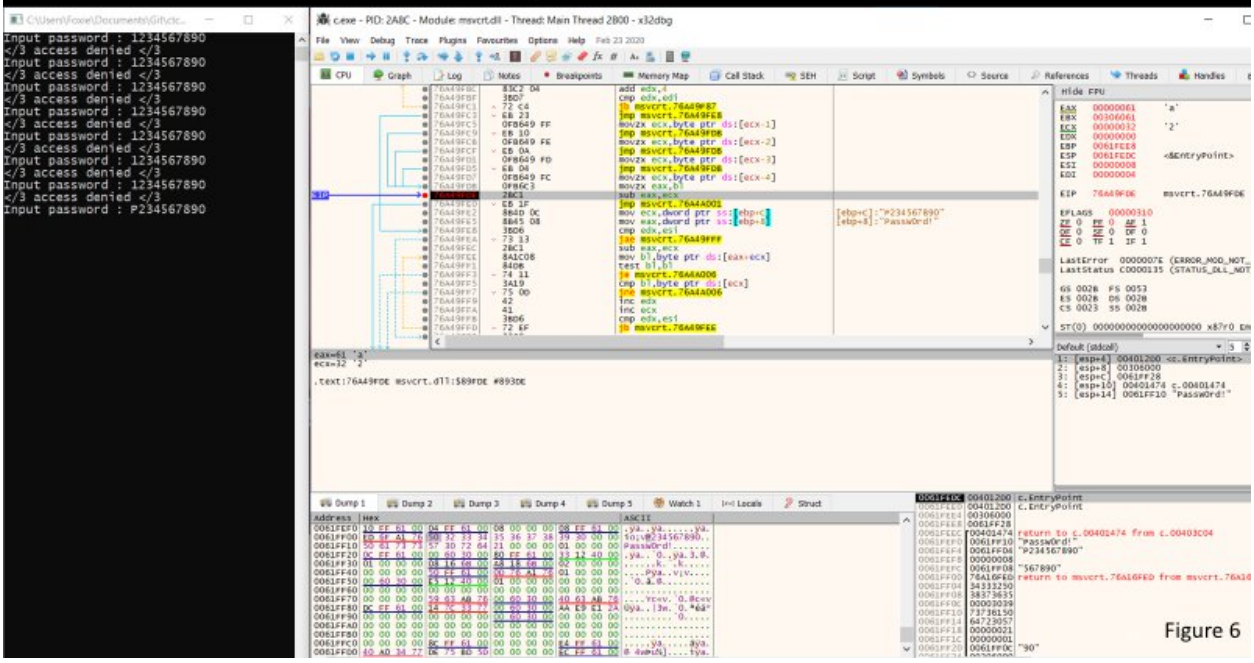
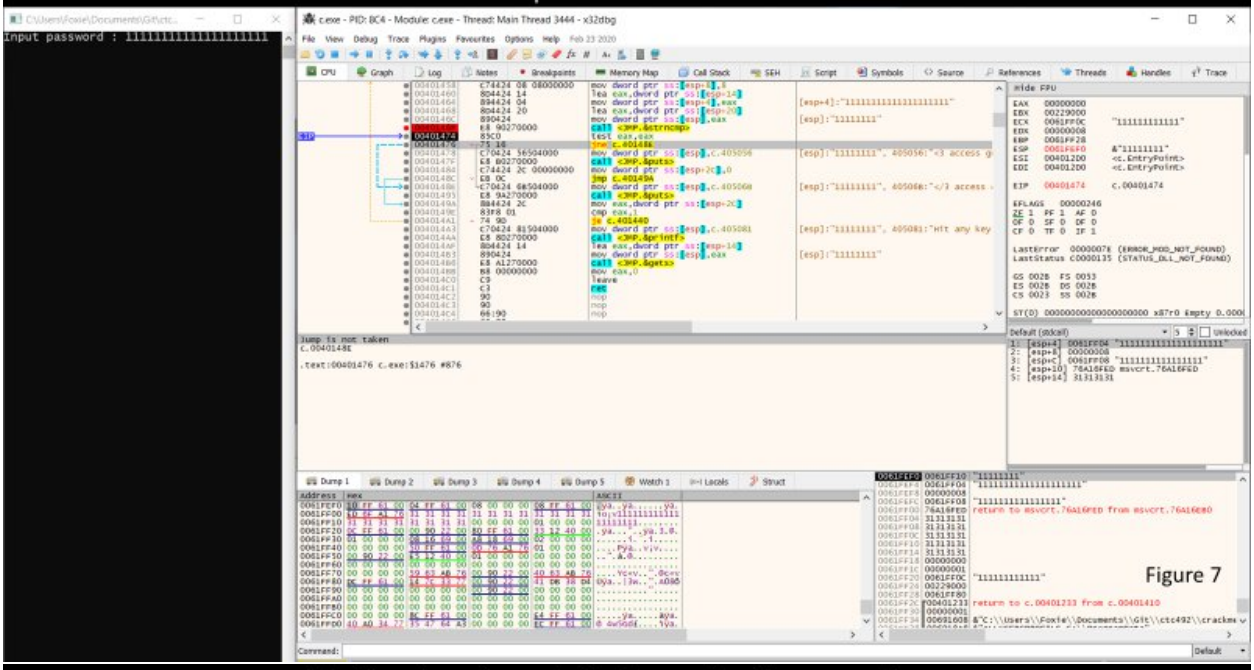
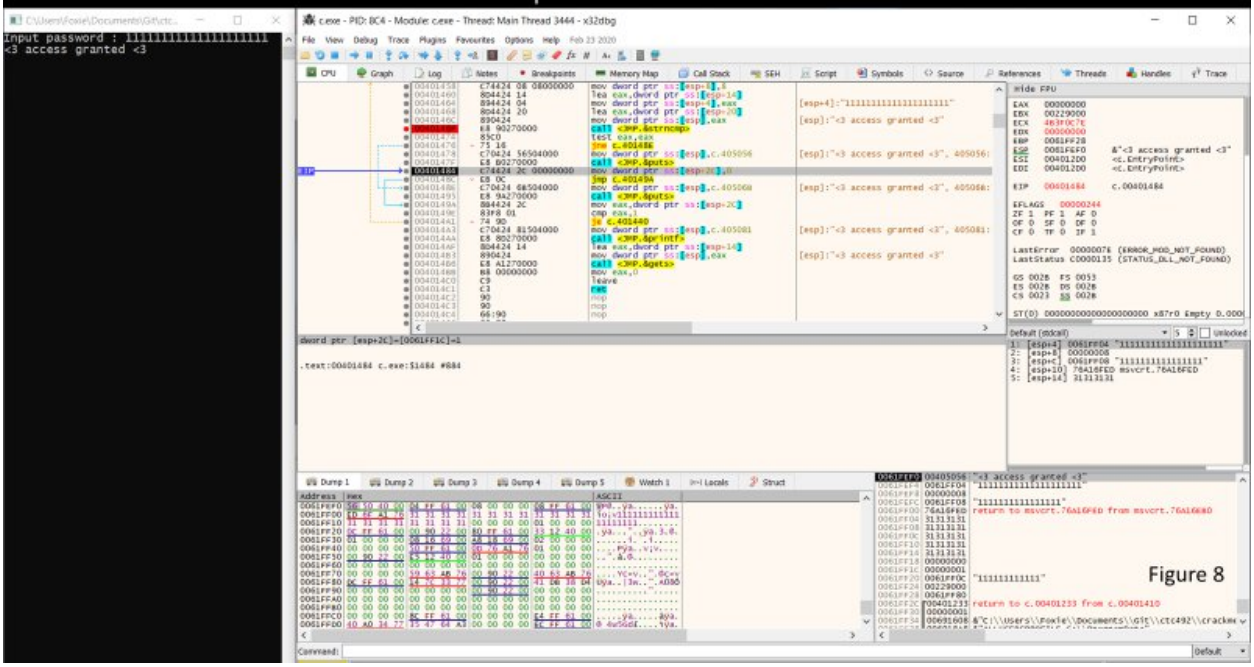


Figure 6

Methods of exploitation: Buffer overflow

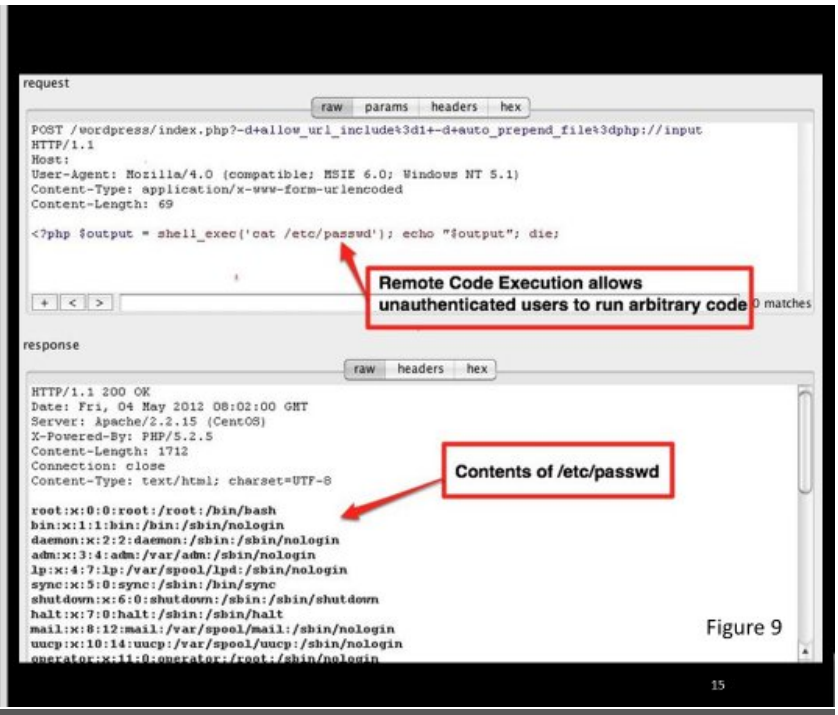


Methods of exploitation: Buffer overflow



Methods of exploitation Remote Code Execution (RCE)

- "A more complex but much more damaging abuse case is using the vulnerability to execute arbitrary code on the server" [1].
- A PHP exploit where the user inputs a command.
- Server executes the unsanitized code.



Results Budget/Cost

- Total estimated cost used in programs, hardware etc.
- Software Free
  - Git
  - Notepad++
  - gcc
  - C
  - X64dbg
- Hardware used around 700$
  - I5 2.30ghz with 8 hyper threads 24gb ddr3 laptop

16

Results
Time Table

- Talk about estimated investment total time 30 hours ish
  - Research 8
  - Dev setup 4
  - Programing 8
  - Debugging 10
- Had merge mapping program milestone to analysis
- Had to cut how to patch a program with outsource

Summary of work
Future plans and goals met

- Program a few more flawed applications
- Patch program with vulnerabilities

- Explained main ideas of reverse engineering
- Developed a few crack-mes

Summary of work
Trouble shooting

- Switched from visual studio to Notepad++ and gcc
- Was able to easily compile with parameters to help explain flaws in run time.
- Had to research a lot of concepts in detail to be able to purposely recreate them.

19

Summary of work
What was learned

- Learned more about reverse engineering, debugging and programing.
- A seconded attempt at this would include crackmes that covered more types of vulnerabilities.
- Used more current examples that aren't already checked my modern compiled and interpreted languages.
- Explained a more granular prosses of debugging a program.

20

Refences

Figure 1: Friedl, Steve. "Intel x86 Function-call Conventions - Assembly View." [*Unixwiz.net*, http://unixwiz.net/techtips/win32-callconv-asm.html. Accessed 4 April. 2020.]

Figure 4: *The University of Manchester*. "Buffer Overflow." [*The University of Manchester*, https://xerxes.cs.manchester.ac.uk/comp251/kb/Buffer_Overflow/. Accessed 4 April. 2020.]

Figure 9: Claudius , Jonathan. "PHP-CGI Exploitation by Example." [*Trustwave*, May 08, 2012. https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/php-cgi-exploitation-by-example/. Accessed 4 April. 2020.]

[1] Claudius , Jonathan. "PHP-CGI Exploitation by Example." May 08, 2012. [Online]. Available: https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/php-cgi-exploitation-by-example/. [Accessed 4 April. 2020].