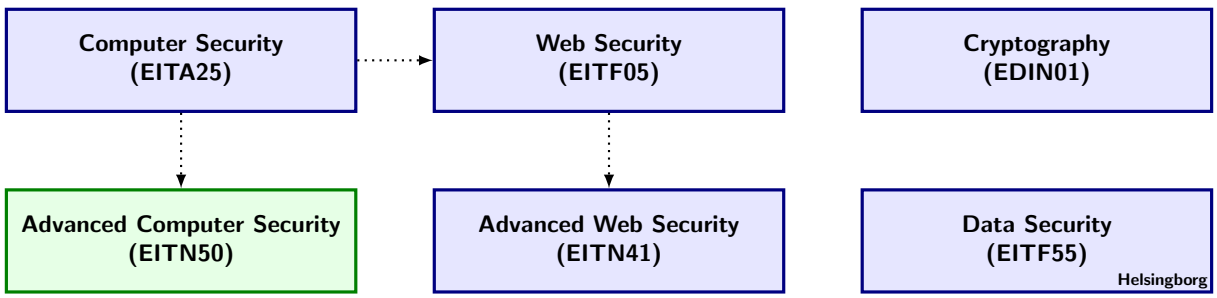
Advanced Computer Security 2019

Department of Electrical and Information Technology
Lund University

Project: Anatomy of an Exploit

Learning goals:

- Introduce the steps of reverse engineering.
- Get familiar with the x86 assembler language.
- Get an understanding for how ROP exploitation works.



Preparations

- Read the master's thesis, `http://www.eit.lth.se/index.php?gpuid=288&L=0&eauid=829`, and study the following chapters: Buffer Overflows (2.2), Return-Oriented Programming (2.4) and Address Space Layout Randomization (2.6).
- Read the entire project description (this document) before you begin.

Instructions for Project Approval

The project consists of a number of assignments that guide your work and you should use the assignment numbers to structure your report.

- Indicate on the front page your group number in addition to your name(s).
- In the report, give clear indications with reference to the related assignment and question numbers where you put your answers.
- Document your work with logs and printouts.
- You should submit the report electronically in pdf format. Give the file the following name: "adsecxy_project1.pdf", where xy is the number of your group and use the subject "EITN50" in the email that contains the report. Send it to `ben.smeets.lu@analys.urkund.se`.

1 Introduction

In this assignment you will study a real-world exploit. The objective of this project is for you to explain its behaviour through static and dynamic analysis. This particular exploit utilizes a technique called Return-Oriented Programming, which grants attackers new vectors of exploitation.

The exploit is written in Python and is available here: `http://www.exploit-db.com/exploits/33610`. The installation program for the vulnerable software is also available at that link. **Note that this application is vulnerable and may be exploited by a third party if you allow connections through the port of your choice (default 80) in your firewall. The safest bet would be to only work on this project on the lab computers.**

You will be given a VMware virtual machine with the above software included. The VM is located at `S:\Courses\eit\EITN50\Project - Exploit`.

2 Prerequisites

In order to execute the exploit you need Python 2.7.x 64-bit[1].

In order to analyze the exploit dynamically you need a debugger. A common Windows based debugger is OllyDbg[2], which gives you fine-grained control over the program execution.

The above software is already installed on the virtual machine.

> **Preparatory assignment 1**
>
> If you have no experience with debuggers, begin by reading the following tutorials on OllyDbg:
>
> `http://octopuslabs.io/legend/blog/archives/31.html`
>
> `http://octopuslabs.io/legend/blog/archives/115.html`

You should be able to:

- Start a program / Attach a running program
- Step through the program
- Enable/disable breakpoints
- Follow data in the dump and stack windows

A cheat sheet for OllyDbg is available here: `http://ollydbg.de/quickst.htm`.

> **Preparatory assignment 2**
>
> To get familiarized with the x86 assembly language, read
>
> **Mandatory**: `http://www.eit.lth.se/fileadmin/eit/courses/eitn50/Project_Exploit/assembly_intro.pdf`
>
> Optional: `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

[1] `https://www.python.org/download`
[2] `http://www.ollydbg.de/`

3 Assignments

3.1 Part A

The exploit's payload is sent as a UserID string in an HTTP GET request, as is clearly seen in the exploit code.

> **Assignment 1**
>
> Try to run the exploit and report, **in detail**, what happens.

> **Assignment 2**
>
> Study the exploit code and extract the gadgets and parameters used for the ROP-part of the attack. Illustrate what the stack looks like. Ignore the NOPs and shellcode.

It can be rather difficult to see the meaning behind each gadget by just looking at static code, even if it is relatively well documented. It is much easier to see what is going on by actually executing the code and observing what each instruction in the exploit achieves in the end.

> **Assignment 3**
>
> You will now perform dynamic analysis on the exploit and document **as much information** about it as possible. The execution of the exploit begins at address `0x00468702` in module `fmws`. The program should be in "running" mode in Olly before executing the exploit. Executing the exploit should make Olly break at the address above. You might have to run through the program a couple of times before the exploit's code is executed. The following information is required:
>
> - A complete listing of all instructions executed by the exploit from the instruction at the above mention address, up until the shellcode. Redundant instructions like the NOPs can be simplified in the listing.
> - Highlight the instructions that you find essential for the exploit.
> - Clearly motivate what purpose each essential instruction has on the outcome of the exploit, e.g. for the instruction `mov eax, ebx`, **do not** write "moves ebx to eax", rather explain *why* the instruction is present and what purpose it serves. Make sure you number your instruction listing so that you can refer to them in your documentation. **Hint**: *analyze the gadgets from bottom up. It is easier to understand what needs to happen in order to run the shellcode if you begin analyze from the last gadget.*
> - What about the rest of the instructions? Discuss.
> - What is the purpose of the filler code in the exploit? When performing the dynamic analysis, try to figure out why the fillers are needed. Add images when discussing to make it easier to follow.

ASLR (Address Space Layout Randomization) makes sure that executable modules in memory has different base addresses each time it is executed in order to hinder code-reuse attacks such as ROP. It is obvious that the vulnerable application has at least some parts of memory unprotected with this defense.

> **Assignment 4**
>
> Which parts of the memory in the vulnerable application are not protected by ASLR? How can you tell?

Assignment 5

How does the exploit use the non-protected parts to be able to execute arbitrary code?

There are many defenses that are aimed at hindering both code injection attacks and code reuse attacks. The W^X (Write xor eXecute) defense hinders memory pages from being writeable and executeable at the same time. In Windows this defense is labeled DEP for Data Execution Prevention.

Assignment 6

Is DEP enabled for the vulnerable application? How can you tell?

Assignment 7

Propose a way to extend the exploit to work when DEP is enabled for the application. Research other ROP attacks on google and `http://exploit-db.com`.

3.2 Part B

Assignment 8

Implement a working exploit under DEP. You may use the Immunity Debugger plugin "Mona" `https://github.com/corelan/mona`, `https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/`.