Lab07

Advanced Buffer Overflow and Shell Code

[You must not attack any network without authorization! There are also severe legal consequences for unauthorized interception of network data.]

Introduction

This project will introduce you to control-flow hijacking vulnerabilities in application software, including buffer overflows. You will be working through this in a virtual machine environment starting with some practice programs for you to get familiar with the tools you need. We will then provide a series of vulnerable programs which you will develop exploits.

Objectives

Be able to identify and avoid buffer overflow vulnerabilities in native code.

• Understand the severity of buffer overflows and the necessity of standard defenses.

• Gain familiarity with buffer overflow defense bypass techniques.

Setup

Buffer-overflow exploitation depends on specific details of the target system, so we are providing an Ubuntu VM in which you should develop and test your attacks. We've also slightly tweaked the configuration to disable security features that would complicate your work. We'll use this precise configuration to grade your submissions, so you MUST NOT use your own VM.

1. Download VirtualBox from https://www.virtualbox.org/ and install it on your computer. VirtualBox runs on Windows, Linux, and Mac OS.

2. Get the VM file that we gave you at the lab. This file is 1.3 GB, so we recommend taking this on your pen drive.

3. Launch VirtualBox and select Import Appliance to add the VM.

4. Start the VM. There is a user named ubuntu with password ubuntu.

5. You will also need **this file** to complete your lab work.
6. untar the files and change to that directory. Run ./setcookie firstname
7. make

Resources and Guidelines

No Attack Tools! You MUST NOT use special-purpose tools meant for testing security or exploiting vulnerabilities. You MUST complete the project using only general purpose tools, such as gdb. You will make extensive use of the GDB debugger. Useful commands that you may not know are “disassemble”, “info reg”, “x”, and setting breakpoints. See the GDB help for details, and don’t be afraid to experiment! This quick reference may also be useful:

http://www.tenouk.com/Module000linuxgcc1.html

x86 Assembly These are many good references for Intel assembly language, but note that this project targets the 32-bit x86 ISA. The stack is organized differently in x86 and x86_64. If you are reading any online documentation, ensure that it is based on the x86 architecture, not x86_64.

7.2.4 Overwriting the return address indirectly (9 points) (Difficulty: Medium)

In this target, the programmer is using a safer function (strncpy) to copy the input string to a buffer. Therefore, the buffer overflow exploit is restricted and cannot directly overwrite the return address. However, this programmer has miscalculated the length of the buffer. Hopefully this will help you to find another way to gain control. Your input should cause the provided shellcode to execute and open a root shell.

What to submit
Create a Python program named 4.2.4.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.4 $(python 4.2.4.py)

7.2.5 Beyond strings (9 points) (Difficulty: Medium)

This target takes as its command-line argument the name of a data file it will read. The file format is a 32-bit count followed by that many 32-bit integers. Create a data file that causes the provided shellcode to execute and opens a root shell.

What to submit
Create a Python program named 4.2.5.py that outputs the contents of a data file to be read by the target. Test your program with the command line:
python 4.2.5.py > tmp; ./4.2.5 tmp

7.2.6 Bypassing DEP (9 points) (Difficulty: Medium)

This program resembles 4.2.3, but it has been compiled with data execution prevention (DEP) enabled. DEP means that the processor will refuse to execute instructions stored on the stack. You can overflow the stack and modify values like the return address, but you can't jump to any shellcode you inject. You need to find another way to run the command /bin/sh and open a root shell.

What to submit
Create a Python program named 4.2.6.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.6 $(python 4.2.6.py)
For this target, it's acceptable if the program segfaults after the root shell is closed.

7.2.7 Variable stack position (9 points) (Difficulty: Medium)

When we constructed the previous targets, we ensured that the stack would be in the same position every time the vulnerable function was called, but this is often not the case in real targets. In fact, a defense called ASLR (address-space layout randomization) makes buffer overflows harder to exploit by changing the position of the stack and other memory areas on each execution. This target resembles 4.2.3, but the stack position is randomly offset by 0–0x110 bytes each time it runs. You need to construct an input that always opens a root shell despite this randomization.

What to submit
Create a Python program named 4.2.7.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.7 $(python 4.2.7.py)

7.2.8 Linked list exploitation (10 points) (Difficulty: Hard)

This program implements a doubly linked list on the heap. It takes three command-line arguments. Figure out a way to exploit it to open a root shell. You may need to modify the provided shellcode slightly.

What to submit
Create a Python program named 4.2.8.py that print lines to be used for each of the command-line arguments to the target. Test your program with the command line:
./4.2.8 $(python 4.2.8.py)

7.2.9 Returned-oriented Programming (10 points) (Difficulty: Hard)

This target uses the same code as 4.2.3, but it is compiled with DEP enabled. Your job is to construct a ROP attack to open a root shell.

Tips:
1. You can use objdump to search for useful gadgets:
objdump -d ./4.2.9 > 4.2.9.txt
2. Reading Havoc's paper may help: https://cseweb.ucsd.edu/~hovav/dist/geometry.
Pdf

What to submit
Create a Python program named 4.2.9.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.9 $(python 4.2.9.py)

7.2.10 Callback shell (10 points) (Difficulty: Hard)

This target uses the same code as 4.2.4, but you have a different objective. Instead of opening a root shell, implement your own shellcode to implement a callback shell. Your shellcode should open a TCP connection to 127.0.0.1 on port 31337. Commands received over this connection should be executed at a root shell, and the output should be sent back to the remote machine.

Tips:
1. Lecture slide is a good starting point.
2. Network byte order for s_addr and sin_addr are in big endian.
3. You can write your shellcode in x86 assembly, then use objdump to translate the shellcode to hex.

What to submit
Create a Python program named 4.2.10.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.10 $(python 4.2.10.py)
For the remote end of the connection, use netcat:
nc -l 31337

To receive credit, you must include (as an extended comment in your Python file) a fully annotated disassembly on your shellcode that explains in detail how it works.

7.2.11 Format String Attack (10 points) (Difficulty: Hard)

Did you know that printf is actually vulnerable to an attack called format string attack? Your job is to exploit this and open a root shell. You should think about what the format specifier %n does.

What to submit
Create a Python program named 4.2.11.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.11 $(python 4.2.11.py)