Assignment 4

Due Date: December 10th, 2022

1. Buffer Overflow attacks:

Even after 30 years of discovery, buffer overflow attacks (originally discovered in 1988 was used in the design of *morris* worm), continue to remain the number 1 cause of security attacks. Some major buffer overflow attacks seen in the last decade are:

1. ***Heartbleed bug in openSSL cryptography library***. This was exploited in April 2014 to steal private crypto keys of major banking servers across the globe. It greatly affected Indian banks and on account of which RBI initiated emergency procedures wherein banks were asked to replace their digital certificates.
2. Buffer overflow vulnerability in Microsoft word (CVE-2012-0158) was exploited through ***carbanak*** malware. Hackers stole ***1 Billion USD*** (from a bank in eastern Europe) in a span of three years (2011 and 2014) before it was discovered.
3. On Feb 19th, 2013 hackers stole 45 million USD (banks from middle east were affected) from ATMs in a span of 8 hours. Yes, you can force an ATM to randomly disburse any amount of cash (with or without debit card inserted). ***FACT: There are little over 2 lac ATMs in India and almost all of them still run-on age-old windows XP OS and is likely to have unpatched vulnerabilities as Microsoft has stopped supporting windows XP several years ago.***
4. In 2010, US govt deployed a buffer overflow attack through stuxnet virus (was planted in Siemens built PLCs) on an Iranian nuclear research reactor which resulted in delaying Iran's nuclear program by a few years.
5. Finally stealing money from mobile wallets. Buffer overflow vulnerabilities exist in both Android and IoS mobile operation system and widely exploited trigger fraudulent transactions.

The attack strategy of buffer overflow attack is to find a vulnerable pointer in the user program and overwrite the address held in the pointer. The new address now points to attack code. The attack code itself can be placed either in the program stack or heap or as part of a corrupted dll (or shared object in case of linux OS).

Problem Statement:

In real world, ability to exploit buffer overflow vulnerabilities requires ability to reverse engineer the binary program, as you may not have access to the source code of the program you intend to hijack.

However, in this assignment, to simplify your job you are provided with the source code of a vulnerable program. The program prompts the user for a password and your task is to provide an input such that, on further execution:

1. Design input to the program i.e. command line argument such that the program will print "access granted" even when you give wrong password as input. Use source file ***hackthisprogram1.c***. **[This is easiest to do!!!]**
2. The program launches a unix shell. Use source file ***hackthisprogram1.c***
3. **RAM Scrapping using buffer overflow vulnerability**: Format an input that allows you to print the entire contents of RAM memory or to obtain values stored at a desired memory location on the RAM. In real world this target address that you plan to read might hold valuable information such as crypto keys. Use source file ***hackthisprogram2.c.***
4. **Return-to-libc attacks (HARD PROBLEM but I will give an A grade for the course if you can demonstrate this).** Successful execution requires placing the address of *system* call into the location where %eip is stored on the stack. In real world after you hijack a remote machine, you want to have that machine to periodically POST valuable information to your machine. So return-to-libc attacks are the preferred way to do this reporting. Use source file ***hackthisprogram1.c***
5. You can use the program HackTestProgram.c for basic study. [use reference 5(a) for more info to practice]

GOAL:

- The purpose of this assignment is to familiarize you with gdb (necessary to reverse engineer a target program) and layout of program memory on RAM.
- Understand the importance of security testing and how it is different from traditional approaches used in software testing. For example, traditional testing techniques when applied to the vulnerable program in this assignment may not identify any bugs.
- The source code of the program that you need to attack is given in the file *hackthisprogram2.c*

Outline of Solution:

Your task is to determine the location of %eip stored on the stack and rewrite the contents with the destination address of the attack code. For this you will use *gdb* to study the layout of stack and determine the format string that need to be passed as input to the program.

Output:

You can either submit a report with detailed screenshots showing results of hacking or come and demonstrate the attacks in front of me.

TEST BED:

- You can perform the above attacks either on KaliLinux or Ubuntu 18.0 or later version.
- Modern operating systems come with several protection mechanisms to defend against buffer overflow attacks. You will have to disable these defences before you can execute your attack. Use the following commands to disable the protection mechanisms:
  - Disable ASLR:
    - *su root*
    - *sysctl -w kernel.randomize_va_space=0*

    or by using

    echo 0 > /proc/sys/kernel/randomize_va_space
  - Compile the .c file with an additional gcc option *-fno-stack-protector* to enable execution from stack segment i.e. compile the code using

NOTE: The commands to disable the protection mechanisms are sometimes kernel dependent, so refer to your kernel documentation for correct commands in case the above do not work.

gcc -o hackthisprogram -z execstack -fno-stack-protector hackthisprogram.c

The above commands are for Ubuntu, so in case you are using any other OS you will have to refer to user manual of your OS. **Other resources to set up your work environment**

1. Disable protection mechanisms https://unix.stackexchange.com/questions/66802/disable-stack-protection-on-ubuntu-for-buffer-overflow-without-c-compiler-flags
2. Setting up your virtual machine: https://github.com/npapernot/buffer-overflow-attack

References:

1. GDB (to examine memory contents)
    a. https://www.sourceware.org/gdb/current/onlinedocs/gdb.html#Data
    b. https://web.mit.edu/gnu/doc/html/gdb_10.html
    c. https://visualgdb.com/gdbreference/commands/disassemble
2. Compiling C Program: https://cs-fundamentals.com/c-programming/how-to-compile-c-program-using-gcc.php
3. Step by step guide to format string vulnerabilities with sample code
    a. [ret-to-libc] https://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html
    b. [basic format string] https://www.tallan.com/blog/2019/04/04/exploring-buffer-overflows-in-c-part-two-the-exploit/
    c. https://cheese-hub.github.io/secure-coding/04-formatstring/index.html
4. Determine address of various functions in libc: https://security.stackexchange.com/questions/121922/return-to-libc-cant-get-the-system-function-address
5. Examining Memory contents (Hands on):
    a. https://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html
    b. https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/
    c. http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html
    d. https://jvns.ca/blog/2021/05/17/how-to-look-at-the-stack-in-gdb/
8. **The code to launch shell is: [One of the many ways]**
   \x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80
   To give the above line as command line input, copy paste the above code in a text file and use the following command.
   a.out $(cat inputfile)
   inputfile is the file holding the shell code (give above) and ***a.out*** is the program that you want to hack.
   To do the same from gdb prompt use the following command at gdb prompt
   run $(cat inputfile)
9. Refer to shared pdf documents (onedrive folder) for more info.

Other Useful Info:

- The stack layout will be different for each of you and so while attack technique is same you will have to explore on your own to detect the correct destination address (to place in the location of %eip on the stack) for each case. So the exact solution will have to be tailored specifically for your environment.