CSC 495 — Assignment 1 — Due Thursday, February 19

1 CSC 495 — Assignment 1 — Due Thursday, February 19

1. Find a story on a recent (within the past year) software vulnerability that led to a security breach. You need to find a story that talks about how a specific software vulnerability was involved, so not just any security story will do! Write a brief summary of the incident in your own words, describing what software was involved, what the vulnerability was, and what the consequences were. Be as specific as you can - in particular, if possible provide a CVE code (see http://cve.mitre.org for reference) for the specific software vulnerability, provide one or more CWE codes (see http://cwe.mitre.org/ for reference) for the type of weakness(es) involved, and even include a CAPEC ID (see http://capec.mitre.org/) for the type of attack if possible.

2. As we discussed in class, "Calling Conventions" specify how arguments passed to a function in a high-level language are actually provided to a function when it is compiled to assembly/machine language. While we discussed this some in class, do some investigation on calling conventions for Intel architecture in a 32-bit model (sometimes called the IA-32 or i386 or x86 model) and a 64-bit model (usually called the x86-64 model). For the 32-bit model, describe in your own words the standard `cdecl` calling convention, and describe any differences between the Linux/gcc method and the Microsoft method. Do the same for the 64-bit model (both the System V AMD64 ABI method and the Microsoft method).

   Finally, notice that stack-based calling conventions typically push arguments on the stack in a right-to-left order. Work through the specifics of how such a calling convention works for the `printf` function, and use this to explain why the right-to-left ordering is important. Be very specific, down to the underlying assembly language — what would the difficulty be if arguments were pushed in a left-to right order?

3. In this problem you will explore "CWE-798: Use of Hard-coded Credentials." This is number 7 on the CWE/SANS Top 25 Most Dangerous Software Errors list, where it is described as follows (read more at the provided links):

   > Hard-coding a secret password or cryptographic key into your program is bad manners, even though it makes it extremely convenient - for skilled reverse engineers. While it might shrink your testing and support budgets, it can reduce the security of your customers to dust.

   For this problem, you will practice being a "skilled reverse engineer" on three different executables provided for you (named `findpass1`, `findpass2`, and `findpass3`), each using a different method for hard-coding a password with increasing levels of protection. Each program is a setuid executable, which runs as a different user (`hw1user1`, `hw1user2`, and `hw1user3`) and prompts for a password. If you enter the right password, a shell is executed that runs with the effective user id

(euid) set to that user, which will allow you to read a file that is protected so that it is readable only by that user.

The first two programs are required work for this assignment, but the third is more challenging and is extra credit!

What to do: You first need to locate the programs with the embedded passwords, so log in to your account on `cmpunix` and go into the directory `/csc495/hw1/findpass` and look for a directory named by your user name (each student has their own custom programs to reverse engineer, all with different passwords!). In your own personal directory you will find the executable files and protected data files that you are to work with. There are lots of standard tools you can use to gather information and try to reverse engineer the programs, including `gdb`, `strings`, and `objdump`.

What to turn in: For each program, describe what you did to break the file and find the password. You should describe this in enough detail so that someone could read your description and repeat the steps to break the code. In addition to this description, you need to say what the password was for the program and what the contents of the secret file are. (Hint: all passwords are 6 letter words, and all secret file contents are 8 digit hexadecimal numbers.)

4. In this problem you will exploit a program with stack-based buffer overflow vulnerability (CWE-120: Classic Buffer Overflow) that follows from the use of an unsafe function. While modern systems have multiple protections against buffer overflow exploits, making them very difficult to perform, I have purposefully turned off these protections to make this exploit easier. It is still quite challenging however, so don't think you can get this done at the last minute! The following things have been done to make this easier to attack: it is compiled as a 32-bit executable with an executable stack, no stack protection, and ASLR turned off.

   Here is the basic information: The `cmpunix` system is running a super-simplified web server that you can connect to from `cmpunix` on port 8888. This port is *not* open to the Internet, so you must log in to `cmpunix` via `ssh` - to see the web server running, you can either use port forwarding over ssh to connect from your own system (I'll demonstrate how to do this in class), or you can use the command "`nc localhost 8888`" to connect from the shell prompt.

   The source code that is running on the web server is available on `cmpunix` at `/csc495/hw1/buffoverflow/badhttpd.c`, so your first task is to read through this code, understand how it works, and find the buffer overflow vulnerability.

   a. This code provides a handy "helpful to the hacker" function: when there is a segmentation fault, caused by the program attempting to access an invalid memory location, it will report that fact *and* give the illegal memory address that the program was trying to access. Try connecting to the program using the 'nc` command above, and type about 80 `A' characters and see what happens. Then try 40 A's and 40 B's. Once you understand how that works, figure out how to modify parts of that input so that you can figure out where the return address is stored relative to the start of the input buffer. In other words, is it 12 bytes from the beginning? 40 bytes from the beginning? 80 bytes from the beginning? *To turn in*: Give the value you discovered (where the function return address is relative to the start of the beginning of the input buffer), and describe the process you used to determine this.

b. For this part, you are going to go for a full exploit, not just crashing the server like you did in part a. This is very tricky if you've never done it before, so you will have to be persistent! If you can get a shell through the server, you can easily complete this exercise. This is a stack-smashing attack, and there are many references on-line explaining how to do this, including the "Smashing the Stack for Fun and Profit" article that is linked under the Readings section of the class web site. You can use any resources that you can locate on-line, including code, but be sure to cite your sources in your solution write-up. Also note that "any resources that you can locate on-line" does not include solutions of other students!

   What nefarious thing do you need to accomplish? The server can only read files from the system, and has no capability for writing or saving files, so your goal is to exploit the system in a way that allow you to write to a file in the account of the user that is running the web server (user `hw1user4`). In particular, you should create a file named by your user name, with some text contents. For example, if your user name were `stsnape` you could create a file named `stsnape.txt` with the contents "The Half-Blood Prince was here." For this part, you should write up what you did to exploit the server to turn in with your homework, in addition to leaving that file on the system. If you don't get the full exploit accomplished, you should at least write up a description of what you tried and describe how close you felt you were to completing the exploit.

   Extra incentive for not putting this off: 5 points extra credit will be given to the *first* student with a successful exploit.

5. One way to try to avoid buffer overflow problems is to keep track of the size of every buffer you allocate so that you can always check to make sure there's enough room for an operation before performing it. You could define a "sized buffer" type like this:

```
struct sizedbuff_s {
    unsigned int size;
    char data[0];
};
```

   This structure adds a size field, so needs slightly more memory, but can potentially result in safer code. Here's an example of how an allocation might work, where this code gets a buffer size from the (untrusted!) user and then allocates and zeros out the new block of memory:

```
int i;
unsigned int user_size = get_size_from_user();
unsigned int struct_size = sizeof(unsigned int) + user_size;
struct sizedbuff_s *mybuffer =
    (struct sizedbuff_s *)malloc(struct_size);

for (i=0; i<user_size; i++)
    mybuffer->data[i] = 0;
```

   Unfortunately, while this helps us avoid buffer overflows, it actually has a different vulnerability. What precisely is this vulnerability, how can the user trigger it, and what are the consequences? When you

answer how the vulnerability is triggered, be very specific: what specific value would be supplied on a 32-bit machine, and what *precisely* would happen. (Hint: CWE 190)

6. Now that you have seen programs with security vulnerabilities in the previous two problems, you are going to see how well you can do at writing a secure program. First create a directory named `securefiles` under your home directory, and in that directory create at least 3 different text files. Set permissions so that you can read those files, but no other non-root user on the system can. Also create a file in your home directory named `secretstuff.txt` that no one else can read. Finally, write a program that will be compiled to an executable named `getfile` in your home directory that does the following: prompt the user for a password and a file name, and output the contents of that file from the `securefiles` directory (if it exists). Here are the basic security requirements: It should be impossible for an attacker who does not know the password to reverse engineer it from the executable, and it should be impossible for an attacker to get the contents of any file outside of the `securefiles` directory (so in particular it should be impossible to read `secretstuff.txt` from your home directory, even for a user that knows the password). Your executable program `getfile` should be a setuid executable that any user on the system can access and run. Have another student test your program from their account to make sure they can access the files that they should be able to! In addition to the stated high-level security goals, make sure you follow good coding practices: don't use unsafe functions, always check return values and handle exceptions properly, etc.