UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Winter 2015**
Urs Hengartner

ASSIGNMENT 1

Blog Task signup due date: **Thursday, January 15th, 2015** 3:00pm (no extension)
Milestone due date: **Thursday, January 22th, 2015 3:00pm**
Assignment due date: **Monday, February 2nd, 2015 3:00 pm**

Total marks: 64
Written Response Questions TA: Nabiha Asghar
Programming Question TA: Yihang (Frank) Song

Please use Piazza for all communication. Ask a private question if necessary. The TAs' office hours are also posted to Piazza.

Blog Task

0. [0 marks, but -2 if you do not sign up by the due date] Sign up for a blog task timeslot by the due date above. The 48 hour late policy, as described in the course syllabus, does not apply to this signup due date. Look at the blog task in the Course Materials, Content section of the course website to learn how to sign up.

Written Response Question [26 marks]

Note: For written questions, please be sure to use complete, grammatically-correct sentences. You will be marked on the presentation and clarity of your answers as well as the content.

1. In this question, you are asked to compare fingerprint-based access control to badge-based access control.

   In a fingerprint-based access control system for a building, all the doors have a fingerprint scanner. To open the door, users place their finger on the scanner. The scanner forms an image of the fingerprint pattern and sends it to a central server. The server then matches the image with templates for fingerprints that are allowed access to that door. If there is a match, the central server will open the door and log the successful unlocking of the door by that user.

In a badge-based access control system for a building, every authorised user is issued an access card with a unique barcode. Every door has a barcode scanner. The scanner reads the barcode on a user's access card and sends it to a central server. The server then looks up for users associated with that barcode. If the user is on the list of allowed users for that door, the central server will open the door and log the successful unlocking of the door by that user.

All logs are timestamped.

For the following questions, be sure to state any assumptions that you make.

- (8 marks) Explain and discuss whether the above fingerprint-based access control system is susceptible to interception, interruption, modification, and fabrication attacks. If you believe the system is susceptible to a given attack, provide a specific, realistic attack scenario. If necessary, state any additional assumptions made about the system.

  Interception: The attacker could compromise the server and copy the logfile, compromising user privacy. Another possible interception attack could be the interception of user fingerprints through apparently innocous means.

  Interruption: The attacker may damage the fingerprint scanner. The attacker may also use DDoS attacks against the server or DoS attacks against the reader. The attacker may steal a reader or physically hurt a user's fingers (so that the fingerprint does not match or even cut the finger off).

  Modification: The attacker may modify the logfile to conceal his entry or alter time of entry through a door.

  Fabrication: The attacker may use fake fingerprints.

- (8 marks) Explain and discuss whether the above badge-based access control system is susceptible to interception, interruption, modification, and fabrication attacks. If you believe the system is susceptible to a given attack, provide a specific, realistic attack scenario.

  Interception: The attacker could compromise the server and copy the logfile, compromising user privacy. An attacker may also use a mobile device to capture the barcode on an access card.

  Interruption: The attacker may damage the barcode scanner. The attacker may also use DoS attacks against the server or DoS attacks against the reader. The attacker may also physically steal a user's access card or a reader.

  Modification: The attacker may modify the logfile to conceal his entry or alter time of entry through a door.

  Fabrication: The attacker may use a duplicate access card that has the same bar code as a user.

  1 mark each for providing a valid attack in an incorrect category. 2 marks for a valid attack in the correct category.

  I anticipate some confusion over the physical theft of an access card being interception or interruption. Obtaining a copy of data, like the barcode, is interception. Stealing an access card is interruption. Also, note that modifying the logfile or access list with new entries is fabrication, not modification. Similarly, altering someones permissions is modification, whereas removing them from the access list is interruption. The distinction between reading, deleting, modifying, and adding is important, but not necessarily intuitive from the English definitions of the attacks.

- (2 marks) Based on your discussion, which system is more secure?

  They are both equally vulnerable, though some of the attacks on the fingerprint scanner may require a little more effort or physical intervention. Answers that make sense based on the answers to the previous questions are acceptable.

2. For each of the two following scenarios, describe how the police officer/Alice's phone identifies and authenticates Alice. Clearly separate the identification and the authentication step. You should wait with answering this question until we have discussed identification and authentication in Module 3.

   - (4 marks) Alice from Waterloo gets stopped by a police officer while driving somewhere in Ontario.

     Identification: The police officer asks Alice for her driver's license and reads her name from it (2 marks).

     Authentification: The police officer looks at the picture on Alice's license and ensures that Alice looks like the person in the picture (1 mark). (Optionally, the officer could take the gender and height into account, which are also printed on the license.) The officer also ensures that the license is a valid Ontario driver's license (e.g., no tampering, not expired) (1 mark).

   - (4 marks) Alice has configured the Smart Lock feature of her Android 5.0 Lollipop smartphone. You can find more information about Smart Lock on the Internet.

     Identification: Alice starts interacting with her phone. The phone assumes that the phone's owner (i.e., Alice) wants to access the device (2 marks). (If there are multiple user accounts on the device, Alice needs to activate her account.)

     Authentication: The smartphone checks whether it can sense one of Alice's trusted Bluetooth devices or NFC tags in its proximity. (Instead of a "trusted device", the phone can also be configured to look for a "trusted face" or a "trusted place".) If so, the phone lets Alice access the device (1 mark). If none of the devices/tags is nearby, Alice's backup authentication mechanism (e.g., a passcode-based scheme) takes over and asks Alice to enter her secret value (1 mark).

Programming Question [40 marks]

Background

You are tasked with testing the security of a custom-developed *backup application* for your organization. It is known that the application was *very poorly written*, and that in the past, this application had been exploited by some users with the malicious intent of *gaining root privileges*. There is some talk of the application having *four or more vulnerabilities*! As you are the only person in your organization to have a background in computer security, only you can *demonstrate how these vulnerabilities can be exploited* and *document/describe your exploits* so a fix can be made in the future.

Application Description

The application is a very simple program with the purpose of backing up and restoring files. There are at least two ways to invoke it:

- `backup backup foo` : this will copy file *foo* from the current working directory into the backup space.
- `backup restore foo` : this will copy file *foo* from the backup space into the current working directory.

There may be other ways to invoke the program that you are unaware of. Luckily, you have been provided with the source code of the application, `backup.c`, for further analysis.

The executable `backup` is *setuid root*, meaning that whenever `backup` is executed (even by a normal user), it will have the full privileges of *root* instead of the privileges of the normal user. Therefore, if a normal user can exploit a vulnerability in a setuid root target, he or she can cause the target to execute arbitrary code (such as shellcode) with the full permissions of root. If you are successful, running your exploit program will execute the setuid backup, which will perform some privileged operations, which will result in a shell with root privileges. You can verify that the resulting shell has root privileges by running the `whoami` command within that shell. The shell can be exited with `exit` command.

Testing Environment

To help with your testing, you have been provided with a virtual *user-mode linux* (uml) environment where you can log in and test your exploits. These are located on one of the *ugster* machines. You will receive an email with your account credentials for your designated ugster machine.

Once you have logged into your ugster account, you can run `uml` to start your virtual linux environment. The following logins are useful to keep handy as reference:

- `user` (no password): main login for virtual environment
- `halt` (no password): halts the virtual environment, and returns you to the ugster prompt

The executable `backup` application has been installed to `/usr/local/bin` in the virtual environment, while `/usr/local/src` on the same environment contains `backup.c`. Conveniently, someone seems to have left some shellcode in `shellcode.h` in the same directory.

It is important to note all changes made to the virtual environment will be lost when you halt it. Thus it is important to remember to keep your working files in `/share` on the virtual environment, which maps to `~/uml/share` on the ugster environment.

Rules for exploit execution

- You have to submit four exploit programs to be considered for full credit. Two of your submitted exploit programs must exploit specific vulnerabilities. Namely, one must target a buffer overflow vulnerability, and another must target a format string vulnerability. Your other submitted exploit programs can address other vulnerabilities.
- Each vulnerability can be exploited only in a single exploit program. A single exploit program can exploit more than one vulnerability. If unsure whether two vulnerabilities are different, please ask a private question on Piazza.
- There is a specific execution procedure for your exploit programs ("*sploits*") when they are tested (i.e. graded) in the virtual environment:
  - Sploits will be run in a **pristine** virtual environment, i.e. you should not expect the presence of any additional files that are not already available
  - Execution will be from a clean `/share` directory on the virtual environment as follows: `./sploitX` (where X=1..4)
  - Sploits must not require any command line parameters
  - Sploits must not expect any user input
  - If your sploit requires additional files, it has to create them itself
- For marking, we will compile your exploit programs in the /share directory in a virtual machine in the following way: `gcc -Wall -ggdb sploitX.c -o sploitX`. You can assume that shellcode.h is available in the /share directory.

- Be polite. After ending up in a root shell, the user invoking your exploit program must still be able to exit the shell, log out, and terminate the virtual machine by logging in as user `halt`. Also, please do not run any cpu-intensive processes for a long time on the ugster machines (see below). None of the exploits should take more than about a minute to finish.
- Give feedback. In case your exploit program might not succeed instantly, keep the user informed of what is going on.

Deliverables

Each sploit is worth 10 marks, divided up as follows:

- 6 marks for a successfully running exploit that gains root
- 4 marks for the description of the identified vulnerability/vulnerabilities, saying how your exploit program exploits it/them, and describing how it/they could be repaired.

A total of four exploits must be submitted to be considered for full credit. Marks will be docked if you submit no *buffer overflow* sploit or no *format string* sploit. **Note:** sploit1.c and sploit2.c are due by the milestone due date given above.

What to hand in

It is very important that you follow the rules outlined in the Assignments section of the LEARN course site for submitting your assignment. Otherwise we may not be able to mark your assignment and you may lose partial or all marks.

By the **milestone due date**, you are required to hand in:

sploit1.c, sploit2.c Two completed exploit programs for the programming question. Note that we will build your sploit programs **on the uml virtual machine**

a1-milestone.pdf: A PDF file containing exploit descriptions for sploit1 and 2

Note: You will not be able to submit sploit1.c, sploit2.c or a1-milestone.pdf after the milestone due date (plus 48 hours).

By the **assignment due date**, you are required to hand in:

sploit3.c, sploit4.c: The two remaining exploit programs for the programming question.

a1.pdf: A PDF file containing your answers for the written-response questions, and the exploit descriptions for sploit3 and 4. Do not put written answers pertaining to sploit1 and 2 into this file; they will be ignored.

The 48 hour late policy, as described in the course syllabus, applies to the milestone due date and the assignment due date. It does not apply to the blog task signup due date.

- 4 marks for the sploit executable
  - Full marks if the sploit opens a fully functional shell and exits normally.
  - 3 marks if the sploit does not work but needs minimal effort to get working.
  - 1 mark for good documentation and feedback if the sploit does not work.
- 3 marks for the sploit description
  - 1 mark for identifying the type of vulnerability and where it is located in the program.
  - 1 mark for **clearly** and **concisely** explaining how their sploit works.
  - 1 mark for describing a valid repair for the vulnerability.

Useful Information For Programming Sploits

Most of the exploit programs do not require much code to be written. Nonetheless, we advise you to start early since you will likely have to read additional information to acquire the necessary knowledge for finding and exploiting a vulnerability. Namely, we suggest that you take a closer look at the following items:

- Module 2
- Smashing the Stack for Fun and Profit (http://insecure.org/stf/smashstack.html)
- Exploiting Format String Vulnerabilities (v1.2) (http://julianor.tripod.com/bc/formatstring-1.2.pdf) (Sections 1-3 only)
- The manpages for passwd (man 5 passwd), execve (man 2 execve), su (man su), mkdir (man mkdir), and chdir (man chdir)
- SSH public key authentication (e.g., http://www.cs.uwaterloo.ca/cscf/howto/ssh/public_key/, ignore the PuTTY part for this assignment)

Note that in the virtual machine you cannot create files that are owned by root in the `/share` directory. Similarly, you cannot run chown on files in this directory. (Think about why these limitations exist.)

GDB

The gdb debugger will be useful for writing some of the exploit programs. It is available in the virtual machine. In case you have never used gdb, you are encouraged to look at a tutorial (e.g.,http://www.unknownroad.com/rtfm/gdbtut/).

Assuming your exploit program invokes the `backup` application using the `execve()` (or a similar) function, the following statements will allow you to debug the `backup` application:

1. `gdb sploitX` (X=1..4)
2. `catch exec` (This will make the debugger stop as soon as the `execve()` function is reached)
3. `run` (Run the exploit program)
4. `symbol-file /usr/local/bin/backup` (We are now in the `backup` application, so we need to load its symbol table)
5. `break main` (Set a break-point in the `backup` application)
6. `cont` (Run to break-point)

You can store commands 2-6 in a file and use the "`source`" command to execute them. Some other useful gdb commands are:

- "`info frame`" displays information about the current stackframe. Namely, "saved eip" gives you the current return address, as stored on the stack. Under saved registers, eip tells you where on the stack the return address is stored.
- "`info reg esp`" gives you the current value of the stack pointer.
- "`x <address>`" can be used to examine a memory location.
- "`print <variable>`" and "`print &<variable>`" will give you the value and address of a variable, respectively.

- See one of the various gdb cheat sheets (e.g., http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf) for the various formatting options for the print and x command and for other commands.

Note that `backup` will not run with root privileges while you are debugging it with gdb. (Think about why this limitation exists.)

The Ugster Course Computing Environment

In order to responsibly let students learn about security flaws that can be exploited in order to become "root", we have set up a virtual "user-mode linux" (uml) environment where you can log in and mount your attacks. The gcc version for this environment is the same as described in the article "Smashing the Stack for Fun and Profit"; we have also disabled the stack randomization feature of the 2.6 Linux kernel so as to make your life easier. (But if you'd like an extra challenge, ask us how to turn it back on!)

To access this system, you will need to use ssh to log into your account on one of the `ugster` environment: `ugsterXX.student.cs.uwaterloo.ca`. There are a number of ugster machines, and each student will have an account for one of these machines. You will get an e-mail with your password and telling you which ugster you are to use. If you do not receive a password please check your spam folder. When logged into your ugster account, you can run "`uml`" to start the user-mode linux to boot up a virtual machine.

The gcc compiler installed in the uml environment may be very old and may not fully implement the ANSI C99 standard. You might need to declare variables at the beginning of a function, before any other code. You may also be unable to use single-line comments ("//"). If you encounter compile errors, check for these cases before asking on Piazza.

Any changes that you make in the uml environment are lost when you exit (or upon a crash of user-mode linux). **Lost Forever**. Anything you want to keep must be put in `/share` in the virtual machine. This directory maps to `~/uml/share` on the ugster machines, which is how you can copy files in and out of the virtual machine. It is wisest to ssh twice into ugster. In one shell, start user-mode linux, and compile and execute your exploits. In the other account, log into ugster and edit your files directly in `~/uml/share/`, so as to ensure you do not lose any work. The ugster machines are not backed up. You should copy all your work over to your student.cs account regularly.

When you want to exit the virtual machine, use exit. Then at the login prompt, login as user "halt" and no password to halt the machine.

Any questions about your ugster environment should be asked on Piazza.