UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 Computer Security and Privacy Ian McKillop and Cecylia Bocovich

ASSIGNMENT 1
Milestone due date: **Monday, May 25, 2015, 3:00 pm**
Assignment due date: **Tuesday, June 2, 2015, 3:00 pm**

Total Marks: 55
Written Response Questions TA: Nabiha Asghar
Programming Questions TA: Andrew Tinits

Please direct all communication to the "*Ask the TAs a question about Assignment 1*" discussion forum in Learn. For questions that might reveal part of a solution, use the separate Assignment 1 forum that allows you to ask a *private* question. The TAs' office hours will be posted in Learn.

Written Response Questions [25 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. The School of Computer Science is currently replacing a number of physical locks on doors (to some office spaces, labs, and lounges) with electronic locks. Each person who was previously issued a (metal) key is now instead issued with a "fob" that attaches to a keychain. The fob is a passive device with no computational ability. The way it works is that a reader next to the door sends out a query radio signal, and if there is a fob nearby (a few inches in the default configuration), the fob will respond with its unique ID number. The reader then sends that ID number to a central server, which logs the attempted opening of the door. If the ID number is on a list of allowed IDs for that door, the central server will unlock the door, and log the successful unlocking of the door by that ID. All logs are timestamped as well.

   (a) (3 marks) The textbook describes 3 components of a computer system: *Hardware*, *software*, and *data*. Give an example of each of these components with respect to the described electronic lock system.

   - The hardware portions of the system are: The server, the lock, and the key fobs.
   - The software portion of the system is the server/logging software.
   - The data portion of the system is the ID number on the key and the logfile and access list on the server.

1 mark each for naming at least a single component in a given category.

(b) (8 marks) We have discussed 4 types of attacks against computer systems: *Interception*, *interruption*, *modification*, and *fabrication*. Explain and discuss whether the above electronic lock system is susceptible to each of these 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. The attacker may also install a device to capture the ID transmitted by a fob when opening a door.
- Interruption: The attacker may flood an area with wireless interference that prevents a fob from communicating, or simply damage the reader so a lock cannot be opened. An attacker may also DDoS the server to prevent the lock from checking the access list. Physical theft of a key or reader would also be considered an interruption attack.
- Modification: The attacker may modify the logfile to conceal his entry into a door, or to alter his time of entry.
- Fabrication: The attacker may fabricate a duplicate of another user's fob, thus allowing him to enter doors he may otherwise be denied access to.

1 mark each for providing a valid attack an incorrect category. 2 marks for a valid attack in the correct category. 0-2 marks for no attack depending on explanation (only 1 mark unless it's really convincing.)

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

(c) (4 marks) The CIA (*Confidentiality*, *Integrity*, *Availability*) characteristics of a secure system are closely related to the 4 types of attacks listed in Part (b). For each of the 4 types of attacks, briefly describe which aspect of CIA the attack targets.

- Interception: Interception attacks target data, and therefore may provide the attacker with confidential data, thus breaching confidentiality.
- Interruption: An interruption attack, such as a DoS attack, is a direct attack on availability.
- Modification: Modifying data, such as altering an access log in the key fob system, is an attack on integrity.
- Fabrication: Creation of false data, such as adding entries to a log file or fabricating an unauthorized key, is also an attack on integrity.

1 mark each for reasonable justification.

2. (10 marks) Laura is a computer game developer who is looking to sell her games on the Internet. She realizes that once she sells these games, her customers might share them with others. Laura naturally wants to ensure that anybody who has a copy of her games has actually paid for it.

   For each of the five classes of methods to defend against Laura's games being freely shared (prevent, deter, deflect, detect, recover), give an example of a defence in that class or claim that none exists. If you provide an example, briefly (one sentence) explain why it is an example of that class. If you claim that none exists, then explain your reasoning.

Prevent: Cannot be absolutely prevented. The game is a piece of digital information that, once in the hands of others, can freely be distributed (though maybe not used).

Deter: Require the user to register their product with Laura online using a unique product key and some hardware ID. If Laura sends confirmation back, then the software stores the license information in an undocumented location. The software checks for this license whenever it is run. This deters sharing the software because each copy of the product can only receive one confirmation and it is not clear how to circumvent the hardware ID check.

Deflect: Include a copyright notice with the game that threatens legal action if it is shared without permission. This may make freely sharing Laura's games less attractive because the sharers may face lawsuits.

Detect: Look for copies on the Internet. If Laura finds her games being freely shared on the Internet, then she has detected that her games are being free shared.

Recover: Pursue legal action against sharers. Laura may be able to recover some of the money that was not received for the freely shared games.

- For **prevent**:
  - 1 mark for recognizing sharing cannot be absolutely prevented.
  - 1 mark for explaining why this is.
- For each other method of defense (**deter, deflect, detect, recover**):
  - 1 mark for identifying a **realistic** example
  - 1 mark for **clearly** identifying why the example fits in this method of defense.
- 1 mark if they incorrect judge whether or not an example exists for a method of defense, but give strong supporting explanation.
- For reference, the methods of attack are defined as:

Prevent: (absolutely) prevent the attack

Deter: make the attack harder or more expensive

Deflect: make yourself less attractive to attacker

Detect: notice that attack is occurring (or has occurred)

Recover: mitigate the effects of the attack

Programming Question [30 marks]

Background

You are tasked with testing the security of a custom-developed password-generation 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 *three 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 generating a random password and optionally writing it to `/etc/shadow`. The usage of `pwgen` is as follows:

```
Usage:  pwgen [options]
Randomly generates a password, optionally writes it to /etc/shadow
Options:
 -s, --salt <salt>  Specify custom salt, default is random
 -e, --seed [file]  Specify custom seed from file, default is from stdin
 -t, --type <type>  Specify different encryption method
 -w, --write        Write the password to /etc/shadow.
 -v, --version      Show version
 -h, --help         Show this usage message
Encryption types:
0 - DES (default)
1 - MD5
2 - Blowfish
3 - SHA-256
4 - SHA-512
```

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, `pwgen.c`, for further analysis.

The executable `pwgen` is *setuid root*, meaning that whenever `pwgen` 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 program, he or she can cause the program 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 `pwgen`, which will perform some privileged operations that 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 be e-mailed by the programming TA with your account credentials for your designated ugster machine.

Once you have logged into your ugster account with SSH, you can use the `uml` command to start your virtual Linux environment. The following logins can be used:

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

The executable `pwgen` application has been installed to `/usr/local/bin` in the virtual environment, while `/usr/local/src` in the same environment contains `pwgen.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

- 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 Learn.
- 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..3)
  - 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`. 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 a description of the vulnerability used, an explanation of how your sploit program exploits the vulnerability, and a description of how the vulnerability could be fixed

A total of three exploits must be submitted to be considered for full credit. At least one of these **must** be a *buffer overflow*. Marks may be docked if you do not submit a buffer overflow exploit.

There are at lease five vulnerabilities in the program:

1. There is a buffer overflow vulnerability on line 268, where `strcpy()` is used to copy the command line argument into `args.salt`, a char buffer of length 32. This can be fixed by using `strncpy()` instead and null-terminating the buffer.
2. There is another buffer overflow vulnerability on line 295, where `sprintf()` is used to interpolate `argv[0]` into the usage string. The resulting string is stored in a char buffer of length 640. This can be fixed by using `snprintf()` instead and null-terminating the buffer. Another option would be to print the beginning of the usage string, then print `argv[0]`, then print the rest of the usage string.
3. There is a format string vulnerability on line 312, where `printf()` is used to print the buffer described above. Additional format specifiers such as `%n` can be placed in `argv[0]`, which will then be interpreted by the `printf()` statement. This can be fixed by using the form `printf("%s", buffer)`.
4. There is a TOCTTOU vulnerability between the `check_perms()` and `fill_entropy()` functions as called by `parse_args()`. `check_perms()` ensures that `/tmp/pwgen` is a

regular file owned by the user, and `fill_entropy()` writes the user-provided seed to this file. The user can start `pwgen` with the `--seed` option, at which point `pwgen` will perform the check and wait for user input. Before providing input, the user can create a symlink at `/tmp/pwgen` pointing to `/etc/passwd`. The user can then supply the input, whose contents will be written to `/etc/passwd`. This can be used to overwrite or clear the root password. The user can then log in as root using the `su` utility (this can be automated using `expect`). This can be fixed by using `mkstemp()` to create the temporary file. Another option would be to use an in-memory buffer rather than a temporary file, and use OpenSSL's `RAND_seed()` rather than `RAND_load_file()` in this case.

5. There is an incomplete mediation vulnerability in the `get_uid()` and `get_gid()` functions. These functions trust that the `USER` environment variable contains the name of the current user. In fact, the user can set this value to whatever they like. If they set this value to `root` or any non-existent user name, these functions will return the id 0. This can be used to trick `pwgen` into setting root's password rather than the current user's. The user can then log in as root using the `su` utility and the password printed out by `pwgen` (this can be automated using `expect`). This can be fixed by getting the uid and gid with the `getuid()` and `getgid()` functions.

- 6 marks for the sploit executable:
  - 6 marks if the sploit opens a fully functional shell and exits normally
  - 5 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
- 4 marks for the sploit description:
  - 1 mark for identifying the type of vulnerability and where it is located in the program
  - 2 mark for **clearly** and **concisely** explaining how the sploit works
  - 1 mark for describing a valid repair for the vulnerability

What to hand in

All assignment submission takes place on the `student.cs` machines (not ugster or the virtual environments), using the `submit` utility.

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

sploit1.c One completed exploit program for the programming question.

a1-milestone.pdf: A PDF file containing the exploit description for sploit1.

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

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

sploit2.c, sploit3.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 sploit{2, 3}.

Note: The 48 hour no-penalty late policy, as described in the course syllabus, applies to the milestone due date and the assignment due date.

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 execve (man execve), pipe (man pipe), popen (man popen), getenv (man getenv), setenv (man setenv), passwd (man 5 passwd), shadow (man 5 shadow), symlink (man symlink), expect (man expect).

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 `pwgen` application using the `execve()` (or a similar) function, the following statements will allow you to debug the `pwgen` application:

1. `gdb sploitX` (X=1..3)
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/pwgen` (We are now in the `pwgen` application, so we need to load its symbol table)
5. `break main` (Set a break-point in the `pwgen` 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 `pwgen` 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` machines: `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.

The ugster machines are located behind the university's firewall. While on campus you should be able to ssh directly to your ugster machine. When off campus, you have the option of using the university's VPN (see these instructions), or you can first ssh into `linux.student.cs.uwaterloo.ca` and then ssh into your ugster machine from there.

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 Learn.

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 can be helpful to ssh twice into the ugster. In one shell, start user-mode linux, and compile and execute your exploits. In the other shell, edit your files directly in `~/uml/share/`, 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 the ugster environment should be directed to the Programming Question TA.