CS412 Term Project (2019 Spring)
GRASS is buggier on the other side

Revision 1, 15/2/2019

Due dates:

Team decision:	1/3/2019
Development:	28/4/2019
Hacking:	12/5/2019
Patching:	19/5/2019

1 Introduction

In this programming project, you have to deal with a develop, hack, patch, repeat challenge. In these type of challenges, you are given a program description, and you have to develop that program (development phase). Then you inspect the code written from the other teams searching for flaws and vulnerabilities (hacking phase). Finally, you collect feedback about your code from other teams, and you try to fix the bugs (patching phase).

In the course's project, you will take the role of a team of 3 developers and experience the different stages (or cycles) of secure software development. Figure 1 shows a simplified life-cycle of secure development where continuously iterate from development to bug reports to updating the software in the wild.

For the sake of this project, we will consider development as a single black box. You will have to develop a server and client program according to a given protocol specification in the first phase. In practice, software development itself is not just a black box but consists of multiple per-feature-dependent phases and milestones. In the second phase, we will share the implementations among all teams, and you have the opportunity to find exploitable bugs in the code of all other students. In the third phase, you have the opportunity to fix any discovered bugs. While for this class project, we only pass through the life-cycle once, in practice it is repeated indefinitely due to new features being developed or deeper bugs being found in the existing code.

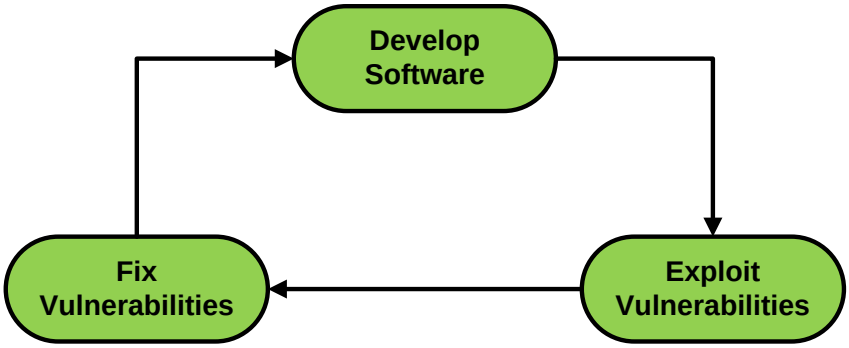

Figure 1: The (security) life-cycle of a program

2 Phase 1: Program Development

You are asked to develop an online search and file management service called "GRASS" – GRep AS a Service, according to a well-defined protocol. Your implementation must follow the defined protocol/configuration format and must be able to interact with other implementations. We encourage you to develop a library that can be used both in the client and in the server as you will have to reuse several functionalities.

In the spirit of ssh and ftp, you will have to implement a clear-text service that allows remote command execution, file transfer, and search. For compartmentalization, our protocol separates command and data channels. To simplify the implementation, we assume that the network is trusted. In practice, we would transfer both control information and data through an encrypted channel.

Overall, you have to implement the following commands: login, pass, ls, cd, mkdir, rm, get, put, search, date, whoami, w, ping, logout, and exit. The command channel is implemented as a simple bidirectional TCP connection. The data channel is implemented as a second TCP channel to a dynamic port.

2.1 Commands

Some commands may be available even without login, yet a set of commands is only available to logged in users, e.g., listing, accessing, or changing files. After authentication, the user can then execute any of the commands. All commands must be followed by a Unix newline ($\backslash n$). The server signals any error by starting the response with `ERROR`, followed by a human-readable error message and a newline.

login The login command starts authentication. The format is `login $USERNAME`, followed by a newline. The username must be one of the allowed usernames in

the configuration file.

pass The pass command must *directly* follow the login command. The format is `pass $PASSWORD`, followed by a newline. The password must match the password for the earlier specified user. If the password matches, the user is successfully authenticated.

ping The ping may always be executed even if the user is not authenticated. The ping command takes one parameter, the host of the machine that is about to be pinged (`ping $HOST`). The server will respond with the output of the Unix command `ping $HOST -c 1`.

ls The ls command may only be executed after a successful authentication. The ls command (`ls`) takes no parameters and lists the available files in the current working directory in the format as reported by `ls -l`.

cd The cd command may only be executed after a successful authentication. The cd command takes exactly one parameter (`cd $DIRECTORY`) and changes the current working directory to the specified one.

mkdir The mkdir command may only be executed after a successful authentication. The mkdir command takes exactly one parameter (`mkdir $DIRECTORY`) and creates a new directory with the specified name in the current working directory. If a file or directory with the specified name already exists this commands returns an error.

rm The rm command may only be executed after a successful authentication. The rm command takes exactly one parameter (`rm $NAME`) and deletes the file or directory with the specified name in the current working directory.

get The get command may only be executed after a successful authentication. The get command takes exactly one parameter (`get $FILENAME`) and retrieves a file from the *current* working directory. The server responds to this command with a TCP port and the file size (in ASCII decimal) in the following format: `get port: $PORT size: $FILESIZE` (followed by a newline) where the client can connect to retrieve the file. In this instance, the server will spawn a thread to send the file to the clients receiving thread as seen in Figure 2.

The server may only leave one port open per client. Note that client and server must handle failure conditions, e.g., if the client issues another get or put request, the server will only handle the new request and ignore (or drop) any stale ones.

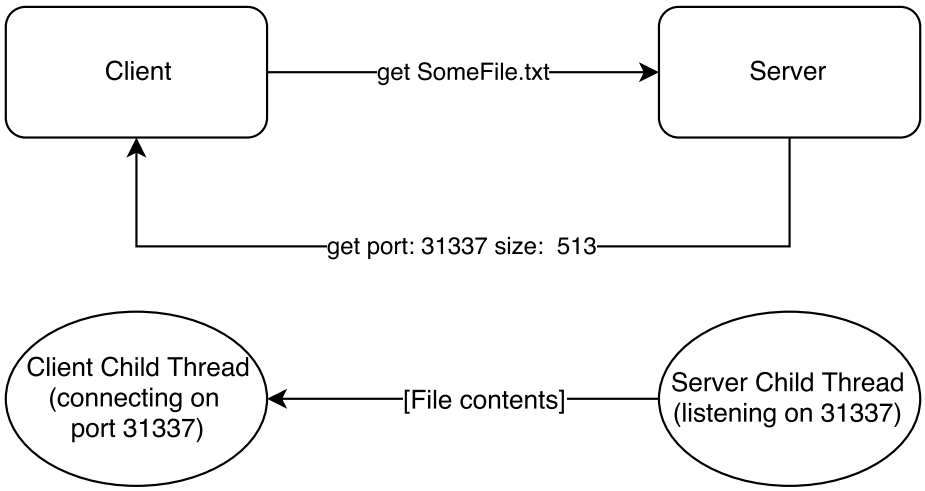

Figure 2: Get Command

put The put command may only be executed after a successful authentication. The put command takes exactly two parameters (`put $FILENAME $SIZE`) and sends the specified file from the current local working directory (i.e., where the client was started) to the server.

The server responds to this command with a TCP port (in ASCII decimal) in the following format: `put port: $PORT`. In this instance, the server will spawn a thread to receive the file from the clients sending thread as seen in Figure 3.

grep The grep command may only be executed after a successful authentication. The grep command takes exactly one parameter (`grep $PATTERN`) and searches every file in the current directory and its subdirectory for the requested pattern. The pattern follows the Extended Regular Expressions rules[1].

The server responds to this command with a line separated list of addresses for matching files in the following format: `$FILEADDRESS $ENDLINE`.

date The date command may only be executed after a successful authentication. The `date` command takes no parameters and returns the output from the Unix date command.

whoami The whoami command may only be executed after a successful authentication. The `whoami` command takes no parameters and returns the name of the currently logged in user.

[1] `https://www.gnu.org/software/grep/manual/html_node/Regular-Expressions.html`

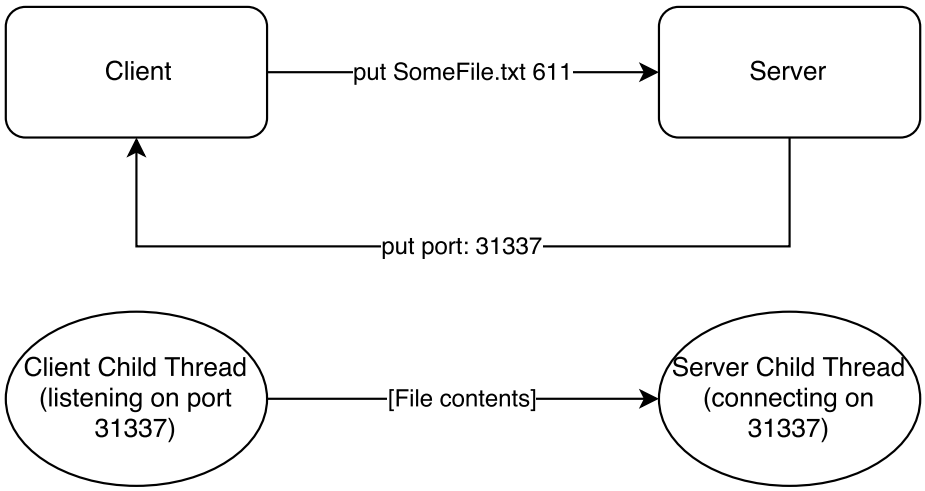

Figure 3: put Command

w The w command may only be executed after a successful authentication. The `w` command takes no parameters and returns a list of each logged in user on a single line space separated.

logout The logout command may only be executed after a successful authentication. The `logout` command takes no parameters and logs the user out of her session.

exit The exit command can always be executed and signals the end of the command session.

Note that the design allows for `put` and `get` to be executed in parallel. We will give bonus points if your implementation (both client and server) allow multiple parallel uploads and downloads at the same time. You will have to spawn a thread for each download or upload both on the server and on the client end to handle multiple data transfers on a single control connection. If you do handle such a situation, do mention it in the report.

2.2 Configuration file

The configuration file (`grass.conf`) specifies the runtime configuration for the server. The following directives are supported: base, port, and user. The `grass.conf` file needs to be placed in the same directory as the client. An example configuration is as follows:

```
# Grass configuration file
# # marks a comment.
```

```
# Current directory is the base directory
base .

# Format: port port number
port 1337

# Format: user name pass
user AcidBurn CrashOverride
```

The server takes exactly no command line arguments

```
./server
```

The client takes 2 command line arguments: the server ip and port:

```
./client server-ip server-port
```

2.3 Code vulnerabilities

Because the process of finding and exploiting bugs is extremely hard, we will simplify it a little. Think of a rogue developer that has planted deniable backdoors in the code. (An obvious backdoor cannot be argued against, but software vulnerabilities allow plausible deniability.)

During the development phase, teams have to inject **5** (five) *exploitable vulnerabilities* in their code. The less obvious a vulnerability is, the less likely is the other teams can find it, so the fewer scoring points are going to be deducted from the program. For example, it is much easier to find a buffer overflow in a buffer that is directly exposed to the attacker, instead of an advanced heap overflow that is triggered through the corruption of internal pointers of the slab allocator[2]. Furthermore, if you design a vulnerability that uses a combination of techniques then you would get more points.

Therefore, the *deeper* you hide the bugs in your code, the harder they are to be exploited. However, using source code obfuscation techniques as a protection mechanism against code auditing, are **not** allowed, as they are out of the scope of this project. Furthermore, the style and readability of the code is part of the evaluation since this project aims to simulate a life-cycle in the development of maintainable software. Each project should contain the following vulnerabilities:

- 2 (two) stack buffer overflows
- 1 (one) format string vulnerability
- 1 (one) command injection vulnerability
- 1 (one) vulnerability of your choice

[2] `https://en.wikipedia.org/wiki/Slab_allocation`

Along with these vulnerabilities, teams have to present a *Proof of Concept* (PoC) of their exploits, which is a small proof that this vulnerability can be used to take control over the program (you do not have to present a fully working exploit). This PoC should be a runnable script to automatically launch the exploit. We would highly recommend using pwntools[3].

Extra backdoors. You can add up to three extra vulnerability of your choice to the code. Rather than a PoC, you need to provide full exploitation for each extra backdoor. If nobody discovers an extra backdoor in the bug bounty, you will receive bonus points.

3 Phase 2: Bug Hunting

Once you have finished the programming part, the source code of all of the projects, will be given to all teams, so they can start looking for bugs in *other's* team's source code.

During this phase, your goal is to audit source code from other teams and look for exploitable bugs (no points will be given for non exploitable ones). Your goal here is to find and exploit **as many bugs as possible** for each other project (in order to get the full score, you have to present working exploits).

To simplify exploitation, the compiled binaries will not have any protections applied (such as ASLR, DEP or canaries). To open a shell temporarily disabling ASLR use the following command: `setarch ‘uname -m‘ -R /bin/bash`

Exploitable vulnerabilities are those which can cause control flow to be diverted i.e. you can demonstrate control of the instruction pointer. To automate the assessment, we restrict the exploitation to following options:

Remote code execution: open a calculator.

Control-flow hijacking: run the `HijackFlow` method from grass library.

Crashing: the program is **not** considered a successful exploit.

If you have an exploit which cannot be converted to one of the following options, ask the TA for further instructions.

4 Phase 3: Fixing Vulnerabilities

In the 3rd, and final phase, each team receives a list containing all the bugs that have been found during previous phases. The goal of this phase is to patch the discovered vulnerabilities and to release a new version. Beyond that, the student has to fix all the original vulnerabilities that were inserted in the code during 1st phase.

[3] `https://github.com/Gallopsled/pwntools`

The goal for each team is to mitigate *all* of the previous working exploits at the end of this phase.

5 Additional Notes

In order to evaluate the correctness and the functionality of your code, we will use a set of test cases that your code has to pass. Also, your server is expected to work with clients from other code and vice versa: your client must be compatible with the servers of other students.

For ease of implementation, we have provided you with a barebones skeleton file to follow while implementing your code.

Teams will be assigned lab machines to run your tests on for a consistent environment. Furthermore, we will take note as to which member in each team contributed to the assignment by their git commit history.

6 Deliverables and evaluations

The program that you are instructed to develop is an online file transfer, and command service. On the server side, there is a daemon listening on a specific port number for new, incoming TCP connections. On the client side, a user uses the client program to connect and interact with the server (this is the classic client-server model in networking).

The deadlines are at 11:59pm CET. You *must* give your TA (kedalat) access to a private github/bitbucket repository and have your code committed by then. We will grade the latest code committed to the master branch by that time. **The deadline to pick a team and to make the github project is 1/3/2019. The git repositories for all teams must be private**. Furthermore, do not publish your code anywhere even after the class is over.

For the development part, submit the source code of the project, any test cases, a PDF presentation of the proof of concept exploits for the 5 original bugs, and full exploitation for extra backdoors. Also, the proof of concept exploit code, if any, should be submitted.

For the first phase, you will be evaluated based on functional correctness (60% of the points), for code quality and documentation (10%), and for the proof of concept vulnerabilities (30%). Note that features that do not pass the functional correctness test will not receive the performance bonus.

Deliverables for this phase are your code (in a git repository), test cases to test the functionality of your server/client, and a one-page documentation. You may implement the service either in C or C++, only using the standard C library (libc, libstdc++) but no other libraries. If you are unsure about a library, do contact the TA to confirm. You must provide a makefile to compile the program and the executable names must be `server` and `client` as in the examples above. Do test that the programs and exploits work successfully on

the lab machines. You will be assigned a lab machine per team as well as ports to work with.

The client must support two modes of operation: in the default mode, the input is read from the keyboard and return values of the server are written to the screen, files are stored in the same directory. In "automated" mode, the client takes two additional arguments: an infile and an outfile.

```
./client server-ip server-port infile outfile
```

The infile contains a list of commands that are executed one after the other. All responses from the server are written to the outfile, files are placed in the local directory. The second mode allows us to script your service for automated functionality testing.

Server and client use a very strict protocol to communicate. Please follow the exact specifications for the protocol, because your server has to communicate with clients of other teams and vice versa.

For the hacking contest, submit a directory for each different project, presenting the working exploits (submit only the exploit source code), and a write up (PDF or plain text) of each found vulnerability.

For the patching part, you have to submit an updated archive of your project and a report on how you addressed the individual bugs.

For all of the parts, your exploits have to be self-contained. That is, in order to reproduce an exploit, you should only have to run a script that is provided. You should also submit a README.txt which details how to run the scripts.

7 Scoring

To score your developed implementation, we will test the functionality of your code by using your API in a set of test programs. Your code must be functional and consistent with the specification. The maximum possible score is 1000, whereas both conformance to the specification and performance will be evaluated.

For every bug/problem that is found in your code during the hacking phase, you will lose up to 100 points. This includes discovered extra backdoors too. You only lose points for every unique bug you have. This means that the number of teams that report the same bug does not affect the number of points you lose. Also, for each bug that you exploit in other students code, you get up to 100 points. Similarly, for each undiscovered extra backdoor, you get up to 100 points.

Finally, at the patching stage, you can get some of your lost points back by fixing your bugs. For any bug you patch, you recover 50% of the lost points. For example, if you lost 10 points for a bug, you will get 5 points for fixing it. You will get no points if you fix bugs that were not reported during the hacking phase. After all phases, the maximum possible score is 100 for the development and patching phases and up to X points for finding vulnerabilities in other people's code. The minimum amount of points is 0.