CS52700 Software Security
Term Project (Spring 2017)

Revision 0.81, 03/01 2017

Due dates:

Phase 1:	03/19/2017
Phase 2:	04/02/2017
Phase 3:	04/16/2017

1 Introduction

Software security is of concern not just during development but throughout the lifetime of a program. In the course's project you will take the role of a developer 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 students 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 existing code.

2 Phase 1: Program Development

You are asked to develop an online file transfer and command execution service called "SPlOIT" – the Singular Powerful Online Internet Transfer 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.

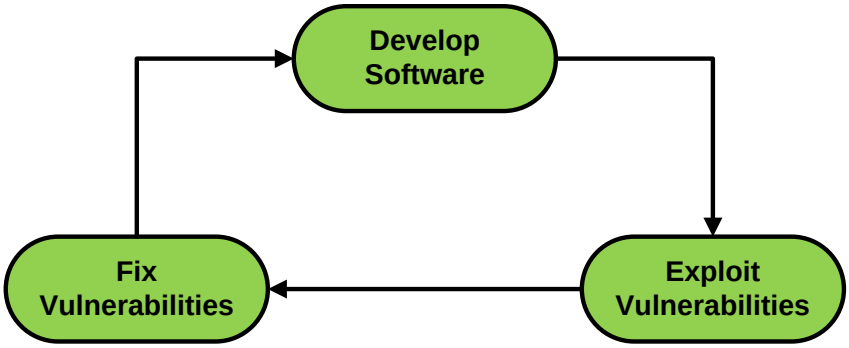

Figure 1: The (security) life-cycle of a program

In the spirit of ssh and ftp, you will have to implement a clear-text service that allows remote command execution and file transfer. 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, get, put, date, whoami, w, ping, logout, and exit. The commands can be extended through the configuration file which may list additional commands that are allowed. 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 basic commands and the additionally defined 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.

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.

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): `get port: $PORT size: $FILESIZE` (followed by a newline) where the client can connect to retrieve the file. 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 any stale ones.

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): `put port: $PORT`.

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.

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

ping. The ping may always be executed. 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`.

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.

Other commands. Any other commands return, in the command channel, the output from the specified Unix command. Any parameters specified in the command channel will be passed to the Unix command.

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.

2.2 Configuration file

The configuration file (`sploit.conf`) specifies the runtime configuration for the server. The following directives are supported: base, port, user, and command. An example configuration follows:

```
# Sploit 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

# Format: command command name, command, parameters
command echo echo
command ll ls -lsah
```

The server takes exactly no command line arguments

```
./server
```

The client takes 2 command line arguments, server's 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 back-

doors in the code. (An obvious backdoor cannot be argued against but software vulnerabilities allow plausible deniability.)

During the development phase, students have to inject **5** (five) *exploitable vulnerabilities* in their code. The less obvious a vulnerability is, the less likely is the other students to find it, so the less 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[1].

Therefore, the *deeper* you hide the bugs in your code, the harder they are to exploit. 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. Each project should contain at least the following vulnerabilities:

- 2 (two) stack buffer overflows
- 1 (one) pointer corruption
- 1 (one) format string vulnerability
- 1 (one) command injection vulnerability

Along with these vulnerabilities, students have to present a *Proof of Concept* (PoC) of their exploits, which is a small proof that this vulnerability can used to take control over the program (you do not have to present a fully working exploit).

2.4 Deliverables and evaluation

For this phase, you will be evaluated based on functional correctness (70% of the points), for code quality and documentation (10%), and for the proof of concept vulnerabilities (20%). Note that a 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++, using the standard library (libc, libstdc++) but no other libraries.

The program that you are called to develop is an online file 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 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 arguments: an infile and an outfile. The infile contains a list of

[1] https://en.wikipedia.org/wiki/Slab_allocation

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 students and vice versa.

3 Phase 2: Bug Hunting

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

During this phase, your goal is to audit source code from other students 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 full score, you have to present working exploits).

To simplify exploitation, the compiled binaries will not have any protections applied (like ASLR, DEP or canaries).

4 Phase 3: Fixing Vulnerabilities

In the 3rd and final phase, each student receives a list containing all the bugs that have been found during previous phase. The goal of this phase is to patch the discovered vulnerabilities and to release a new version. Beyond that, the student has to fix the 5 original vulnerabilities that were inserted in the code during 1st phase.

The goal for each student 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.

The deadlines are at 11:59pm Eastern Time and the files for each file must be submitted as tgz to the TA. Alternatively, you may give the TA access to a bitbucket repository. You have to submit the following files for each phase:

- Phase 1: The source code of the project, your test cases, and a pdf presenting the proof of concept exploits for the 5 original bugs.
- Phase 2: A directory for each different project, presenting the working exploits (submit only the exploit source code), and a write up of each found vulnerability.

- Phase 3: An updated version of the source code and a write-up on all patched vulnerabilities.

6 Scoring

The maximum possible score for the 1st phase is 1000. You will receive up to 100 bonus points for implementing parallel uploads and downloads. For every exploitable bug that is found in your code during the 2nd phase, you will lose between 10 and 50 points. Also for each bug that you exploit in other students you get 100 points.

Finally at 3rd phase, you can get some of your lost points back by fixing your bugs. For each bug you fix, you recover between 10 and 20 points, based on how critical is your fix.