Project report - CS 412 Software Security

Clement Burgelin, Mathieu Ducroux, Andrea Piccione

High-level design

This project is designed to be a simple implementation of the GRASS service and to strictly follow the protocol and specifications given. The primary goal was to implement the features and the commands required in an effective and correct way and, at the same time, make the system secure and mitigate the possible attack vectors. It is mainly written in C but leverages C++ libraries in several parts, above all for taking advantage of the provided data structures and the C++ string class.

Throughout the development of the system, we decided to keep the implementation as simple as possible because we are aware that complex systems tend to become more difficult to manage after some time. That's why we put particular attention on the implementation of each command by applying some of the software security principles learnt in class:

- replacing unsafe functions (e.g. strcpy(), strcat()) with structures from C++ library
- avoiding to dynamically allocate memory and create buffers as much as possible
- validating the input by doing appropriate sanitization of the arguments
- continuously testing each feature implemented with automated tests

Structure of the project

The three main header files (*server.h, client.h, grass.h*) contain the constants and the declaration of the data structures used in the corresponding source files; *errmsg.h* only contains constant error messages that have been used both by the client and the server. The three source files are:

- *grass.cpp*: it contains only the hijack_flow() method which was part of the template provided.

- *client.cpp*: it implements the basic features that allow the client to connect to the server and send messages, print the messages received from the server and handle the files transfer on client side. Both manual and automated mode are supported for the interaction with the server.

- *server.cpp*: here we can find the whole logic of the system with the required commands implementation according to the protocol and the specifications given

Sample execution

In the main function, first the configuration file is loaded and parsed; then, the server starts listening to the specified port for new incoming connections. Every time a client connects, the server creates a new thread for that client and keeps listening for other clients. Each thread is handled by the method connection_handler() which simply implements a REPL for handling client input until the connection is closed with the command exit. Every input by the client is handled (using a lock in order to prevent concurrency bugs) in the method handle_input() which tokenizes the message, parse it, preprocess it and finally, according to the first argument (the command name), calls the function responsible to execute the command by using a defined collection (struct shell_map) which maps the command name to the command function with the correct number of arguments (see server.h for details). Each function (e.g. do_ping(), do_cd, etc.) executes the command and is responsible to do all the checks if they are required (authentication check, path length check, character filtering etc.). Each command is handled separately in order to guarantee isolation, modularity and contain possible faults.

Implementation highlights and optimizations

- A connected user is saved in a map<socket, user> since the socket can be considered a unique identifier and this allows a simple management of the authentication and permissions part.

- Some commands (like ping or date) uses popen() to pipe commands to a shell by calling the method run_command() after having appropriately processed, filtered and sanitized the input.

- In order to check the validity of a path issued by the client, the stat function is used to get information and attributes of a file/folder and this allowed us to manage correctly the input for the commands mkdir, rm and cd in order to prevent unauthorized access to forbidden paths.

- To avoid command injection attacks while allowing characters that are escaped and spaces in file names and such, we decided to use wordexp (which is part of glibc) which, given a string, outputs the parameters separated and de-escaped while counting them and while forbidding certain characters when non-escaped. Wordexp also does command substitution in one of these two formats:

  “anything ‘a command’ anything”
  “anything $(a command) anything”

  However this was not a desired behavior in our case but we realized it’s possible to forbid the malicious characters by setting the WRDE_NOCMD flag to true. Unfortunately, we actually forgot to set this flag and this created an additional unwanted issue. Note that the de-escaped string that wordexp returns was not used and wordexp was actually only used to check the parameters and count them.

Discovered vulnerabilities

The 5 intended bugs we have included in our code have all been found. Below, you will find the description of each of them and and the corresponding fix. In the last bug, we also give details about an unintended bug that has been discovered and how we addressed this issue.

Buffer overflow 1 in mkdir (line 298 in server.cpp)
This was an easy buffer overflow with a simple strcpy() on a buffer (*dirname*) of fixed size.
How we fixed it: we could have used a safer strncpy() but since this buffer was not really useful, we simply removed it.

Buffer overflow 2 in mkdir (line 329 in server.cpp)
The parent path is strcpy-ied inside a buffer of fixed size. This exploit requires a user to construct a path (by doing multiple *mkdir*, *cd*) such that it would overflow the buffer.
How we fixed it: we could have used a safer strncpy() but since this buffer was not really useful, we simply removed it.

Format string in cd (line 297 in server.cpp)
There is a printf of a buffer controlled by the client which is sensitive to format string attacks.
How we fixed it: printing the value of the buffer was already done in the function write_message (line 41 in server.cpp) so we simply removed it.

Command injection in ping (line 181 in server.cpp)
Since ‘;’, ‘(‘, ‘), ‘`’ and ‘$’ are not among the forbidden characters of the parameter of the *ping* command, it is possible to execute any arbitrary command.
How we fixed it: instead of checking if the argument of the ping command does not contain any of the forbidden characters (which we can miss), we check if it only contains authorized characters (alphanumeric ones and few others) and if it doesn’t, we output an error.

Command injection in ls and in other commands (twice in cd, line 772 and 230 in server.cpp)
This was an intended bug for ls only but it undesirably affected some other commands. Since we didn’t set the WRDE_NOCMD flag as explained above, strings with $(cmd) or ‘cmd’ were processed by wordexp that actually executed the command to do command substitution.
How we fixed it: we simply set the WRDE_NOCMD flag when using wordexp.