Reverse Engineering

In this project, you'll reverse engineer ChatMax 2000, a simple email server. ChatMax allows users to send each other short messages. To read and send messages, a user loads the ChatMax client page in a browser. The client page interacts with the ChatMax back-end server, which is called Proton. Proton encrypts per-user mailboxes and the authentication data that is employed to validate user requests.

Project Setup

- Click on the provided GitHub Classroom assignment link, login via GitHub if necessary, and click "Accept assignment".
- Wait for the repository to be created, then create a fork.
- Enable Travis builds for the fork, and turn on "Auto cancel branch builds".
- Login to the VM.
- `cd` to your home directory and run `git clone <fork_url> reverse-engineering/` to clone your fork.
- Run `cd reverse-engineering/` to enter the project directory.
- Run `./pre_setup.sh` to download dependencies.

Refer to Project 0's writeup for elaboration on any of these steps.

> **Caution!**
>
> Before starting, remember the warning from Project 0:
>
> It is important that you **do not** modify the original repository created by GitHub Classroom (the one with `harvard-cs263` in the URL). Only ever modify the fork. This applies to **all** projects in this course.

> **Note**
>
> The ChatMax binaries **should** work on most Linux machines; however, the course staff will only support the course VM, and you are solely responsible for making sure your code passes Travis tests.

> **Tip**

Rejoice! Starting now, you may now use `apt-get` and related commands as you wish. For example, to install `tmux`:

```
sudo apt-get update
sudo apt-get install -qq tmux
```

Specification

Caution!

For all projects, trying to modify or otherwise game the test cases will result in a grade of zero and academic dishonesty sanctions. Contact the course staff if you encounter issues with the tests.

Tip

For all projects, you may commit and push your changes at your leisure. Each push will trigger a remote test, which you can view on the Travis website.

Introduction

The leaked ChatMax code and data is in the `leaked_app/` directory. The leak contains the Proton binary, as well as an encrypted password file, some shared libraries used by Proton, and the web content used by the client-side ChatMax page. You will have to configure your reverse engineering environment so that Proton can find the shared libraries; in particular, you'll need to create some symbolic links that point to the leaked libraries, and you'll have to perform an `export LD_LIBRARY_PATH="$HOME"/reverse-engineering/leaked_app/` command (add this to `~/.bashrc` to make your life easier) to ensure that the dynamic linker will search your current working directory for the libraries. For more information on how shared libraries work, and for examples of the necessary incantations, see this overview.

Once you've set up the dynamic libraries, you can run the Proton binary using a command line like this:

```
you@yourMachine:~/reverse-engineering/leaked_app$ ./proton 8080 .
```

such that you can open the client page by loading http://192.168.26.3:8080/home.html in a browser.

The Proton binary and the shared libraries were not compiled with debugging symbols enabled. This will make reverse engineering more difficult. However, you can still learn about the structure of the code using the `readelf`, `objdump`, `strings`, and `strace` tools. See the relevant class notes (available in the Canvas "Files" section) to get an overview of those tools.

You may also find it useful to employ a browser-side debugger. Those debuggers are very powerful, and have equivalents for many gdb features like breakpoints. For example, here is an introduction to Firefox's built-in debugger. One particularly nice feature of the debugger is its ability to pretty-print minified JavaScript code. Web developers minify their JavaScript to reduce its size (which makes it faster to download) and to obscure its structure (which makes it more difficult to reverse engineer).

If your local machine's processor supports hardware-based watchpoints, those watchpoints can be very useful for gdb debugging. For example, using an `rwatch` command, you can cause the program to break when a particular memory location is read. Similarly, a watch statement will inform you whenever a particular memory location is written. You can tell if your machine supports hardware-based watchpoints by issuing this command from gdb:

```
(gdb) show can-use-hw-watchpoints
Debugger's willingness to use watchpoint hardware is 1.
```

gdb can also use software-based watchpoints, but they are much slower!

Note that Proton detaches itself from the console and runs in the background as a daemon. So, as you run your experiments, you may need to check whether an old copy of Proton is running before starting a new one. You can run `pkill proton` to kill any old copies.

Part 1: Decrypting the Password Database

One of the files in the leaked app looks like a password database. Unfortunately, it also looks quite encrypted. Find any necessary decryption key(s), then decrypt the password database and extract the mapping from usernames to hashed passwords. Save your decrypted password database as a file called `decrypted_passwords.txt` (in the repository's root, not in `leaked_app/`). This file can be in whichever format you prefer, but you will find it most helpful to represent the password hashes as hex strings.

In addition, answer the written questions in `keys.txt` (save your answers directly in the file).

Testing: `make test_keys` will check that the correct key(s) are in `keys.txt`.

Part 2: Reverse Engineering the Hash Function

In order to crack some passwords, we also need the hash function that Proton uses to convert a plaintext password into a hash value that is compared against a hash in the password table. Reverse engineer this hash function and implement it in `hashpw.c`. In the comment at the top of the file, **thoroughly** describe how you reverse-engineered the hash function.

Caution!

Your reverse-engineered hash function should not be assembly code that is copied from Proton and called by a real C function; instead, your reverse-engineered hash function should be real, high-level C code with equivalent functionality to the assembly code in Proton.

Compiling: Run `make hasher` to compile your hash function.

Running: Run `./hasher 'my_string'` to run your hash function on the string and print the result in hex.

Testing: Run `make test_hasher` to test your hash function.

Part 3: Writing a Password Cracker

Using your reverse-engineered hash function, implement a password cracking function in `crackpw.c` according to the specification in the file. Do not use off-the-shelf password crackers (e.g. John the Ripper) -- instead, write your own. In the comment at the top of the file, briefly describe how your cracker works.

Important

You may find it useful to use external text or data files. Put these files in the `data/` directory. The total size of the `data/` directory should be at most **1 MB**.

If you do choose to use a list, we highly recommend that you use this list (RockYou top 25000). You should not need any other list to successfully complete this project.

Finally, any file paths you use in your cracker should be **relative** paths (relative to the repository root), not absolute paths. For example, to open a data file `test123.txt`, you should do `fopen("data/test123.txt", ...)`, not `fopen("/home/username/reverse-engineering/data/test123.txt", ...)`. If you use the latter, Travis tests will undoubtedly fail.

Compiling: Run `make cracker` to compile your crack function.

Running: Run `./cracker deadbeef` to run your crack function (replace `deadbeef` with the hex of your hash to crack).

Cracking the Database

Using your cracker, find the cleartext passwords for at least 8 usernames, including the password for the root account. For each listed user, the cracker must have cracked their password in **60 seconds** or less (assuming single-core on a 2.6GHz Intel Xeon -- we will give you some leeway on

Travis and let you go up to 120 seconds). This might seem like a tight constraint, but the passwords that some of these users have chosen are truly atrocious.

> **Tip**
>
> If you don't feel like being a human timer (who does), the `timeout` tool is for you. Running:
>
> ```
> timeout 60s ./cracker deadbeef
> ```
>
> will try to crack `deadbeef` for 60 seconds, then terminate with a non-zero exit code if it hasn't finished by then. You can use this to write a bash script to attempt a bunch of hashes without babysitting the VM.

Save these username/password pairs in `plaintext-passwords-cracker.csv` (CSV format, and do **not** delete the pre-existing header row).

Testing: `make test_cracker` will verify that the cracked passwords in `plaintext-passwords-cracker.csv` are correct. It will then verify that the cracker can actually crack all of the users' hashes in <= 60 seconds each.

Part 4: Remote Password Cracking

So far, the analysis we have done has been offline, with the assumption that we have a full leak of the app and database. In this part and the following part of this project, you will develop remote attacks against ChatMax without such an assumption.

Keep in mind that, like any web application, ChatMax has some quirks. For example, the client ChatMax page occasionally writes debug statements and warnings to the console log. The ChatMax page also does not automatically poll the server for new messages in a user's inbox; so, to receive new messages, you'll need to refresh the page. The Proton server is also quirky; for example, it may not handle certain kinds of messages that you think are reasonable. You'll have to work around these quirks as you reverse-engineer the system.

Write a remote password cracking function in `network_cracker.py` according to the specification of `crack()`. This function should only use network messages to the ChatMax server. In the comment at the top of the file, briefly describe how your cracker works.

> **Caution!**
>
> You may use your `decrypted_passwords.txt` file as a list of usernames, but you password cracking code should **not** assume knowledge of the hashed passwords or the hash function that the server uses.

This simulates a scenario in which the ChatMax admins have forced users to select new passwords, but the ChatMax server itself runs the same code.

Important

As in the non-remote cracking exercise, you may use external text or data files under the same directory and size conditions. Again, we highly recommend that you use this list (RockYou top 25000). Be careful to use relative instead of absolute paths!

Important

Although not supported by the course staff, you may use a language other than Python. See `network_cracker.sh` for instructions.

Running: After starting a proton server (say, on port 8080), run `./network_cracker.sh some_username localhost 8080`, replacing `some_username` with the one whose password you want to crack.

Cracking Passwords

Using your network cracker, find the cleartext passwords for at least 4 usernames, including the password for the root account. For each listed user, the cracker must have cracked their password in **60 seconds** or less (assuming single-core on a 2.6GHz Intel Xeon -- we will give you some leeway on Travis and let you go up to 120 seconds). Again, this might seem like a tight constraint, but the passwords that some of these users have chosen are truly atrocious. As before, the `timeout` command and bash scripting might be useful.

Save these username/password pairs in `plaintext-passwords-network-cracker.csv` (CSV format, and do **not** delete the pre-existing header row).

Testing: `make test_network_cracker` will verify that the cracked passwords in `plaintext-passwords-network-cracker.csv` are correct. It will then verify that the network cracker can actually crack all of the users' hashes in <= 60 seconds each.

Part 5: Remote Exfiltration

Imagine that you do have the username and password for a non-root user, but you do **not** have the password for root; for the purposes of this exercise, use one of the non-root username/password pairs that you generated in a previous part of the project. As before, you still do not have access to either the ChatMax binaries or database.

Your goal is to exfiltrate the cleartext data from root's mailbox. The exfiltration attack should be launched remotely, i.e., only by sending network messages to the Proton server. "Exfiltration"

means that the attack must be able to send the contents of root's mailbox, either to a remote server (e.g., `upload.attacker.com`), or some other endpoint the attacker can access (e.g. the user's own mailbox). You may assume that `root` logs in and checks their inbox regularly.

> **Tip**
>
> Your exfiltration attack can send more information that strictly necessary if that makes the attack easier for you. For example, you could leak more than one user's server-side mailbox, or you could leak all of the HTML in `root`'s client-side ChatMax page that has been loaded within `root`'s browser.

Describe your attack in `exfiltration.txt`. Your description should be thorough enough that a typical "script-kiddle" (knows how to use a terminal, browser, and not much else) could execute your attack perfectly. You may create additional source code files for this attack, as long as the usage is thoroughly specified in `exfiltration.txt`.

Running/**Testing**: on your own.

Submitting

> **Important**
>
> Before submitting, make sure all your work is committed and pushed to the master branch of your fork, and make sure the Travis build is passing for master. You can verify by going to your fork's GitHub page, clicking on "commits", and looking for a green checkmark at the top of the list.

On the fork's GitHub page. click on "New pull request". The base fork should be the original repository (prefixed with `harvard-cs263`), and the head fork should be your fork (prefixed with your GitHub username). Then, click on "Create pull request" to submit your work! The title can be whatever, and the comment can be left blank (or non-blank if you have a note for the grader).

If you need to edit your submission before the deadline, just commit and push your new changes to the master branch of your fork. The original pull request will be automatically updated with those commits (of course, be sure to check the GitHub pull request page to verify).

> **Caution!**
>
> Do **not** click "Merge pull request" after submitting, as this will modify the original repository. We will merge your pull request when grading.

Caution!

The deadlines for all assignments are on Canvas. Deadlines are enforced to the minute (based on pull request/push times, not commit times), and the course late policy is a 10% deduction per 8 hours of lateness.

Note that the Travis tests can take a while, and no testing-related extensions will be granted.

Deliverables and Rubric

"Automated" grading means we will assign points based on the result of the Travis test case(s).

Criteria	Points	Grading method
`keys.txt` (Question 1)	5	Automated
`keys.txt` (all other questions)	5	Manual
`decrypted_passwords.txt`	5	Manual
`hashpw.c` (correctness)	7	Automated
`hashpw.c` (reverse engineering description)	8	Manual
`plaintext-passwords-cracker.csv` and `crackpw()` (correctness/efficiency)	16	Automated
`crackpw()` (brief description)	4	Manual
`plaintext-passwords-network-cracker.csv` and `network_cracker.py` (correctness/efficiency)	20	Automated
`network_cracker.py` (brief description)	5	Manual
`exfiltration.txt`	25	Manual