CS447-00x : Networks and Data Communications

Programming Assignment #1 (P3)

Total Points: 100

Assigned Date : Wednesday, April 10, 2025
Due Date : Thursday, April 24, 2025 @ 01:59:59 p.m. (*hard deadline*)

Overview

Building upon the video games rental system from P1, you will now enhance its security. This assignment focuses on implementing security mechanisms to protect the system from unauthorized access and ensure data confidentiality and integrity. You will implement Transport Layer Security (TLS) for secure communication, a secure login system, an authentication protocol, a secure password generator, and learn how to protect sensitive data at rest.

Note: This final assignment has a **hard deadline** and does not include the typical 48-hour late penalty period. Your learning objectives are:

a. Gain practical experience with OpenSSL for TLS implementation, secure password hashing, and data encryption.
b. Understand the importance of secure password management and implement techniques for generating, storing, and transmitting passwords securely.
c. Develop skills in securing network applications and protecting sensitive data in transit and at rest.

The nature of this last assignment will force you to do significant amount of unsupervised learning – online research, forum scans, trial-and-error, and troubleshooting. Most likely, you will also discover (on your own) that there are more than one library package implementation of `openssl` standard, especially for C/C++, which might further complicate your task. So, don't get frustrated but instead use this as a valuable learning opportunity.

Back Story

Captain Haddock's ingenious video game system, a welcome respite on the open sea, was infiltrated by a mysterious entity known as "*The Kraken*." This digital intruder, with a code as dark as the deepest ocean trench, breached the system's defenses, threatening the crew's data and the ship's secrets.
Tintin, ever vigilant, recognized the danger, while Haddock, bellowing with maritime fury, demanded Calculus secure their digital haven. Calculus, facing a challenge worthy of his most complex inventions, must now implement a fortress of security: TLS encryption, impenetrable logins, and data vaults shielded from The Kraken's digital tentacles. The safety of the Tintin and the crew's peace of mind hang in the balance, as they navigate this new, treacherous digital sea.

Technical Requirements

ALL P1 technical requirements fully apply to P3. Additionally, the following new requirements must be met.

1. **Secure Communication**
   - Establish **`TLS 1.3`** connections between the `clients` and the `server` for all communication using **`OpenSSL 3.2.2+`** (available on zone instances). This ensures confidentiality and integrity.
   - Use OpenSSL's `SSL_CTX` functions to configure the TLS context:
     - `SSL_CTX_set_min_proto_version()` and `SSL_CTX_set_max_proto_version()` to set the protocol version exclusively to **`TLS 1.3`**.
     - `SSL_CTX_set_ciphersuites()` to select strong cipher suites (e.g., consult OpenSSL documentation for recommended TLS 1.3 suites like `TLS_AES_256_GCM_SHA384`).
   - Generate *self-signed certificates* for testing using OpenSSL's command-line tools. Ensure your generated private key and certificate files are named **`p3server.key`** and **`p3server.crt`**, respectively.

2. **Implement a Secure Login System**
   - Replace the P1 `HELO` command with a `USER` → `PASS` command sequence for authentication.
   - `USER` <`username`>: Used to send the `username` to the `server`. The `server` checks if the received `username` exists. If *YES*, it should await the `PASS` command by signaling "300 Password required". If *NO*, follow the New User Registration Protocol *(see below)*.
   - `PASS` <`password`>: Used to send the user's `password` to the `server`. The `server` follows the Authentication Protocol *(see below)* upon receiving this command.

3. **Password & Salt Generation**
   - **Password Generation**:
     - Generate a random password of exactly **8 characters**[†] for each new user.
     - The password must include at least one uppercase letter (A-Z), at least one lowercase letter (a-z), at least one number (0-9), and at least one symbol from the set **`!@#$%&*_-+=`**. Passwords may not begin with a symbol.
     - Implement a password strength checker function within your server to verify that generated passwords meet these criteria. Regenerate the password if it fails the check.
   - **Salt Generation**:
     - Generate a unique, random salt of exactly **16 bytes** for each new user.
     - Use a cryptographically secure pseudo-random number generator (CSPRNG) for both password character selection and salt generation. OpenSSL's `RAND_bytes()` function is recommended.
4. **Password Hashing/Key Derivation (using KDF)**
   - Use the **`PBKDF2-HMAC-SHA256`** algorithm (Password-Based Key Derivation Function 2 with SHA-256 HMAC) to hash user passwords before storage.
   - **Parameters** for PBKDF2:
     - Use the unique **16-byte salt** generated for the user (see Item 3).
     - Use a fixed **10,000** iteration count (a.k.a. work factor).
     - Use **SHA-256** as the underlying pseudo-random function (PRF) via HMAC.
     - Generate a **32-byte (256-bit)** hash output (derived key).
   - You must use OpenSSL's **`PKCS5_PBKDF2_HMAC()`** function, found in the `libcrypto` library. Ensure your project links against it (e.g., using the `-lcrypto` flag during compilation/linking).
5. **Securely Storing Credentials**
   - All user credentials (username, salt, iteration count, hash/derived key) must be stored persistently on the server-side in a hidden file named **`.games_shadow`** located in the server's current working directory.
   - Use the following text-based format for each line in the file, inspired by the Modular Crypt Format (MCF). Use '$' as the delimiter:

[†]NOTE: Real-world passwords should be significantly longer and ideally generated using methods beyond basic character constraints. This length and complexity requirement is simplified for assignment purposes and is not a security best practice recommendation.

```
username:$pbkdf2-sha256$work_factor$salt_base64$hash_base64
```

Where

- `username` is the user's login name.
- `$pbkdf2-sha256$` is a literal string identifying the algorithm.
- `work_factor` is the integer work factor/iteration count (10000 for this assignment).
- `salt_base64` is the 16-byte salt encoded using Base64.
- `hash_base64` is the 32-byte PBKDF2 output hash encoded using Base64.

Example:

```
haddock:$pbkdf2-sha256$10000$AKeZ/3pkQ3AmwgFLxNO4+w==$v8+h8QyJ...Omitted...jcL5A=
```

- **Note:** Both the 16-byte salt as well as the 32-byte hash output/derived key are in raw byte form, which **must** be encoded using **Base64** before being stored in any text based file. Remember to decode it back to raw bytes when retrieved for authentication.
- Your server must be able to parse this format correctly to retrieve the necessary fields for authentication. Ensure proper handling of file I/O (creation, reading, appending).

6. **New User Registration Protocol**: Triggered when a `USER` command provides an unknown username. The protocol is as follows:
   The server generates a 8-character password and a 16-byte salt (item 3) → generate a password hash (item 4) → securely store the credentials (item 5) → send the generated password to the client over TLS → close connection from the server side.
7. **Authentication Protocol**: Triggered after a successful `USER` command for an existing user, followed by a `PASS` command with an argument *(presumably user's password)*. Here's the protocol:
   The server retrieves the work factor, salt, and the stored hash from the `.book_shadow` (remember to decode base64 values) → compute a new hash for the provided, *(i.e., received)* password → compare the new hash with the stored hash.
   (a) If the hashes match: Send success message (e.g., `210 Authentication successful`). The connection remains open for subsequent P1/P3 application-layer commands.
   (b) If the hashes do not match: Send failure message (e.g., `410 Authentication failed`). The server must close the TLS connection immediately after 2 successive authentication failures.

Functional Requirements

ALL P1 functional requirements fully apply to the P3. Additionally, following new requirements should be met.

1. Fix any lingering issues in P1 before starting on P3 tasks.
2. Use **`openssl s_client`** to interact with your server, similar to how you used **`telnet/nc`** in P2. Here's the syntax:

```
openssl s_client -connect -tls1.3 server_ip:server_port
```

3. Multiple concurrent client functionality will be fully tested in P3.
4. You must provide proof of secure communication. To do this, run `Wireshark` with and without TLS separately, and provide appropriately annotated screenshots in your report. Failure to provide sufficient proof of secure communication will be considered as an indication of not meeting project requirements.
   - Two terminal equivalents for `wireshark` are `tcpdump` and `tshark`. You can save your capturing session as a `.pcap` file, which can be later opened in `wireshark` for easier analysis. The `wireshark` manual explains how to use both. See here https://www.wireshark.org/docs/wsug_html_chunked/AppToolstcpdump.html and here https://www.wireshark.org/docs/man-pages/tshark.html
5. The hidden password store (i.e., **`.games_shadow`**) file should be created when you run your server executable. If the server restarts, the password store should re-initialize.

Extra Credit

- **Only** take this extra credit option if you have ample time, your core implementation is complete, and (to a lesser extent) you have some prior experience programming **`ncurses`**. Submissions that meet (or exceed) at least 80% of project objectives are eligible for up to **20% extra credit** by submitting a **Secure Client** with a **`ncurses`** based text-based user interface (TUI) https://www.gnu.org/software/ncurses/. Consider incorporating features like a login screen, menu-driven navigation for browsing and searching books, and interactive forms for user input.

Instructions

- **Start early & backup often:** This assignment requires careful planning and implementation and remember to save your progress frequently.
- **Don't procrastinate!!** As mentioned earlier, this assignment involves a significant amount of self learning secure network programming and cybersecurity concepts. Allow yourself plenty of time to absorb new knowledge.
- Maintain clean, readable code. Adhere to Google's C++ Style Guide or another established standard. Use one of Google's coding standard found here https://google.github.io/styleguide/, if you don't already follow one.
- Your code must compile and run flawlessly on a standard Linux environment. Test it thoroughly using command-line tools.
- Implement your solution in C++. Submit a `.tgz` and a report.
- Handling parallelism is not trivial. Start with a single-client version, then gradually scale up.
- Focus on core C++ socket and I/O functionalities. Avoid external libraries unless explicitly permitted.
- DEADLINE: **Thursday, April 24, 2025 @ 01:59:59 p.m. (*hard deadline*)** through Moodle. Email submissions are not honored.

Deliverables

A complete solution comprises of:

1. Report (**`pdf`**):
   - Introduction: Describe your learning objectives from a cybersecurity and secure network programming perspective. Explain what you knew about secure programming techniques before starting P3 and what you hope to learn by completing P3.
   - Design: Explain your overall system design, key choices you made, and specific challenges you faced.
   - Sample Run: Include screenshots showcasing a typical interaction with your system. These can help illustrate functionality and potentially earn partial credit if certain features aren't fully implemented.
   - Proof of secure communication, password and salt specification conformity, base64 encoding, etc.
   - Summary and Issues: Summarize your achievements and what you learned from a cybersecurity perspective. Explicitly list any unimplemented or buggy functionality.
2. Compressed Tarball (**`siue-id-p3.tgz`**):
   - Source Code Directory: Include all your C++ source code. No need to include your `client.cpp` unless you took the extra credit option. Also, no need to send your self-signed certificates either (as long as they following the naming standard specified earlier).
   - `Makefile`: A Makefile is required. The instructor should be able to compile your code without errors or warnings by simply typing **`make`**.
   - `README`: Provide clear instructions on how to manually compile and run your code in case **`make`** fails.
   - Your `.pcap` file as proof of secure communication.

Use the following command to create the compressed tarball:
`tar -zcvf siue-id-p3.tgz p3/`.

(Replace `siue-id` with your real siue email id and `p3/` with the name of your source code directory)

Your code should be a testament to your abilities, not a copy of someone else's work. Collaborate, don't copy! Learning from peers is great, but copying code (from classmates or online) is strictly prohibited. Cite any external resources you use for ideas, and then implement those ideas in your own way.

This assignment is a challenge designed to push your boundaries and foster growth. Embrace the learning process and don't hesitate to seek help when needed. Remember, the goal is to master these concepts, not just complete the assignment. Plagiarism, whether from classmates, online sources, or AI tools, stifles learning and compromises academic integrity. The instructor actively uses MOSS http://theory.stanford.edu/~aiken/moss/ to check for software similarity. Plagiarism has severe consequences including and will result in a failing grade. No exceptions.

Some Useful Resources

- Linux Man pages – found in all Linux distributions
- Beej's Guide to Network Programming – A pretty thorough free online tutorial on basic network programming for C/C++ https://beej.us/guide/bgnet/.
- Linux Socket Programming In C++ – https://tldp.org/LDP/LG/issue74/tougher.html
- The Linux HOWTO Page on Socket Programming – https://www.linuxhowtos.org/C_C++/socket.htm
- Learn Makefiles With the tastiest examples – https://makefiletutorial.com/
- Fedora Security Team – Defensive Coding
  https://docs.fedoraproject.org/en-US/Fedora_Security_Team/1/html/Defensive_Coding/index.html
- OpenSSL Wiki – Simple TLS Server
  https://wiki.openssl.org/index.php/Simple_TLS_Server
- The Illustrated TLS 1.3 Connection – https://tls13.xargs.org/
- A Readable Specification of TLS 1.3 – https://www.davidwong.fr/tls13/
- Creating a Self-Signed SSL Certificate – https://linuxize.com/post/creating-a-self-signed-ssl-certificate/
- Ncurses Programming Guide – http://jbwyatt.com/ncurses.html