Auth

Theme Song: Who Am I?

In this assignment, you'll extend the secure communication protocol from Signal to build a secure server-client authentication system. In particular, you'll explore the ways in which we can use digital signatures to expand our circle of trust and be sure that nobody is pretending to be someone they're not.

Due Date: *Friday, February 27*

Contents

1 Background Knowledge

In this assignment, you'll build a secure authentication platform. There are two programs involved: a server and a client. The server acts as a central verification authority that clients will interact with to obtain certificates. The server has its own globally-recognized public key and signs the certificates of clients who properly register or log in. In order to verify with the server, each client must provide a valid password and two-factor authentication (2FA) response. With certificates, clients will be able to communicate with each other while being protected against impersonation attacks.

1.1 Digital Signatures

You've interacted with digital signatures briefly in the warm-up project (Cipher), but we will go over them again here. Digital signatures are essentially the public-key equivalent to MACs, allowing one party to sign a message, and all other parties to verify that this message was signed by that party. To do so, we first generate a keypair consisting of a public verification key vk and a secret signing key sk, then use sk to sign various messages $\sigma_i = \mathsf{Sign}_{\mathsf{sk}}(m_i)$. When another party is given a message and signature, they can use vk to verify that the message was signed correctly: $\mathsf{Vrfy}_{\mathsf{vk}}(\sigma_i, m_i) \stackrel{?}{=} \text{true}$. We want it to be the case that it is hard to forge signatures; that is, given vk but not sk, finding valid signatures for any message, even when given valid signatures for other messages, is hard.

Using digital signatures, we can achieve **authenticated key exchange** that is secure against man-in-the-middle attacks, which we have seen in the Signal project.

1.2 Password Authentication

You probably authenticate by password every day. Password authentication relies on both a server and a user knowing some shared secret pwd, and the user proves that they know this secret by sending pwd (or some altered version of it) to the server. With the cryptographic primitives we've explored thus far in mind, there are a number of naïve ways that one might implement password authentication. One might encrypt the password and send it to the server, who will then decrypt it and store the password in a database (also in an encrypted form) for later verification. One might think that encryption makes this protocol secure both during transit and in storage. However, it is vulnerable in cases where the server or database is completely compromised, as both the database and encryption key could be leaked. Even if we hash the passwords before encryption, adversaries that have access to the hashed passwords could mount an offline

brute-force dictionary attack or consult a rainbow table to crack the passwords. We want to be careful to protect against a variety of attacks against all parts of our system.

We propose a heavily redundant but more secure password authentication scheme so that you get a sense of the techniques you may see out in the wild. Upon registration, the server generates and sends a random (say, 128-bit) **salt** to the user. A salt is a random string appended to a password before hashing it to prevent naïve dictionary attacks. The user sends the hash of their password with the salt appended: $h := H(\mathsf{pwd} \parallel \mathsf{salt})$ to the server, which then computes a random short (say, 8-bit) **pepper** and hashes the user's message with the pepper appended yet again: $h' := H(h \parallel \mathsf{pepper})$. Finally, the server stores the salt, but not the pepper. Upon login, the server sends the stored salt to the user, who then sends the hash of their password along with the salt. Then, the server tries all 2^8 possible pepper values and verifies if any one of them succeeds. This salt-and-pepper approach makes offline dictionary attacks significantly more difficult and expensive.

1.3 Pseudorandom Functions and 2FA

Random numbers are convenient because they introduce a level of unpredictability to systems which can be very useful for keeping secret values secret (e.g. ElGamal encryption uses random values to ensure that even two ciphertexts of the same message are distinct). However, sometimes you want both you and your partner to experience the same randomness, or you may want to cheaply generate more (pseudo)randomness from some base *seed* of randomness. Pseudorandom functions (PRFs) are deterministic but unpredictable functions that take a secret key and an input, and output a pseudorandom value. We want the distribution of PRF outputs to be computationally indistinguishable from that of a truly random function.

PRFs are useful in many ways. For one, they allow you to securely generate an arbitrary amount of seemingly random values deterministically for use in other cryptographic protocols. In this assignment, we'll use a PRF to implement two-factor authentication by using PRF outputs as a way of proving that we know the value of a given shared key s. We can generate a short-lived login token by inputting this key alongside the current time. The server can then validate that our values are correct by computing the same function. We can also think of this process as a pseudorandom generator (PRG) with a random seed s that produces an arbitrary amount of pseudorandom values.

1.4 Putting It All Together

The following diagrams explain how the protocols work together.

We'll first cover interactions between a user and a server. There are four main components:

1. One-sided authenticated key exchange
2. Registration
3. Login
4. Two-sided authenticated key exchange

One-sided authenticated key exchange. A user must either register or login with the server to retrieve their certificate. It is assumed that the server's public verification key, $\mathsf{vk}_{\text{SERVER}}$, is known. In either case, they must first run a key exchange protocol illustrated below. Note that from this project on, we won't be using Diffie-Hellman *ratchet* for ease of implementation.

User		**Server**
$a \in_R \mathbb{Z}_q$		$b \in_R \mathbb{Z}_q$
$A := g^a$		$B := g^b$
	$\xrightarrow{\quad A \quad}$	
		$\sigma_{\text{SERVER}} \leftarrow\!\!\$\ \mathsf{Sign}_{\mathsf{sk}_{\text{SERVER}}}(B \parallel A)$
	$\xleftarrow{\quad (B, A, \sigma_{\text{SERVER}}) \quad}$	
$\mathsf{Vrfy}_{\mathsf{vk}_{\text{SERVER}}}(B \parallel A,\ \sigma_{\text{SERVER}})$		
$s := B^a$		$s := A^b$

From this point on, all communication is encrypted under authenticated encryption.

Registration. If a user wishes to register themselves, they run the following protocol:

User to Server Registration

	User		Server
1 :	**User**		**Server**
2 :		$(\mathsf{id}_i, \text{new\_user}) \longrightarrow$	
3 :			$\mathsf{salt}_i \in_R \{0,1\}^{128}$
4 :		$\mathsf{salt}_i \longleftarrow$	
5 :	$h_i := H(\mathsf{pwd} \parallel \mathsf{salt}_i)$		
6 :		$h_i \longrightarrow$	
7 :			$\mathsf{pepper}_i \in_R \{0,1\}^8$
8 :			$h'_i := H(h_i \parallel \mathsf{pepper}_i)$
9 :			$\mathsf{seed}_i \in_R \{0,1\}^{128}$
10 :		$\mathsf{seed}_i \longleftarrow$	
11 :	$r := \mathsf{PRF}_{\mathsf{seed}_i}(t_{\text{NOW}})$		
12 :		$r \longrightarrow$	
13 :			Verify r
14 :	$(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow\!\!\$\ \mathsf{KGen}$		
15 :		$\mathsf{vk}_i \longrightarrow$	
16 :			$\sigma_{\text{SERVER}} \leftarrow\!\!\$\ \mathsf{Sign}_{\mathsf{sk}_{\text{SERVER}}}(\mathsf{id}_i \parallel \mathsf{vk}_i)$
17 :			$\mathsf{cert}_i := (\mathsf{id}_i, \mathsf{vk}_i, \sigma_{\text{SERVER}})$
18 :		$\mathsf{cert}_i \longleftarrow$	
19 :	Store keys, cert_i, and seed_i locally		Store $(\mathsf{id}_i, h'_i, \mathsf{salt}_i, \mathsf{seed}_i)$ in database

Login. Likewise, the following diagram illustrates the login protocol.

User to Server Login

	User		**Server**
2 :		$\mathsf{id}_i \longrightarrow$	
3 :			Lookup id_i in database
4 :		$\longleftarrow \mathsf{salt}_i$	
5 :	$h_i := H(\mathsf{pwd} \parallel \mathsf{salt}_i)$		
6 :		$h_i \longrightarrow$	
7 :			Try all 2^8 peppers to check if
8 :			$h_i' := H(h_i \parallel \mathsf{pepper}_i)$
9 :			Abort if no pepper matches
10 :	$r := \mathsf{PRF}_{\mathsf{seed}_i}(t_{\text{NOW}})$		
11 :		$r \longrightarrow$	
12 :			Verify r
13 :	$(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow\!\!\$\ \mathsf{KGen}$		
14 :		$\mathsf{vk}_i \longrightarrow$	
15 :			$\sigma_{\text{SERVER}} \leftarrow\!\!\$\ \mathsf{Sign}_{\mathsf{sk}_{\text{SERVER}}}(\mathsf{id}_i \parallel \mathsf{vk}_i)$
16 :			$\mathsf{cert}_i := (\mathsf{id}_i, \mathsf{vk}_i, \sigma_{\text{SERVER}})$
17 :		$\longleftarrow \mathsf{cert}_i$	
18 :	Store cert_i locally		

Two-sided authenticated key exchange. After registration or login, a user has obtained a certificate and is able to communicate with other users. We'll now cover interactions between a user and another user: the main complex part is authenticated key exchange. The protocol for key exchange is illustrated below.

User to User Key Exchange

	Alice		**Bob**
1 :			
2 :	$\mathsf{cert}_{\text{ALICE}}$ (from server)		$\mathsf{cert}_{\text{BOB}}$ (from server)
3 :	$a \in_R \mathbb{Z}_q$		$b \in_R \mathbb{Z}_q$
4 :	$A := g^a$		$B := g^b$
5 :	$\sigma_{\text{ALICE}} \leftarrow\!\!\$\ \mathsf{Sign}_{\mathsf{sk}_{\text{ALICE}}}(A \parallel \mathsf{cert}_{\text{ALICE}})$		$\sigma_{\text{BOB}} \leftarrow\!\!\$\ \mathsf{Sign}_{\mathsf{sk}_{\text{BOB}}}(B \parallel \mathsf{cert}_{\text{BOB}})$
6 :		Alice sends $(A,\ \mathsf{cert}_{\text{ALICE}},\ \sigma_{\text{ALICE}})$ Bob sends $(B,\ \mathsf{cert}_{\text{BOB}},\ \sigma_{\text{BOB}})$ $\longleftrightarrow$	
7 :	$\mathsf{Vrfy}_{\mathsf{vk}_{\text{BOB}}}(B \parallel \mathsf{cert}_{\text{BOB}},\ \sigma_{\text{BOB}})$		$\mathsf{Vrfy}_{\mathsf{vk}_{\text{ALICE}}}(A \parallel \mathsf{cert}_{\text{ALICE}},\ \sigma_{\text{ALICE}})$
8 :	$\mathsf{Vrfy}_{\mathsf{vk}_{\text{SERVER}}}(\mathsf{id}_{\text{BOB}} \parallel \mathsf{vk}_{\text{BOB}},\ \sigma^{\text{BOB}}_{\text{SERVER}})$		$\mathsf{Vrfy}_{\mathsf{vk}_{\text{SERVER}}}(\mathsf{id}_{\text{ALICE}} \parallel \mathsf{vk}_{\text{ALICE}},\ \sigma^{\text{ALICE}}_{\text{SERVER}})$
9 :	$s := B^a$		$s := A^b$

After key exchange all communication can be encrypted just like Signal (without the ratchet).

In short, we proceed in the following steps: login or registration, then communication.

1.4.1 Registration

- On setup, the server has access to an RSA keypair $(\mathsf{vk}_{\text{SERVER}}, \mathsf{sk}_{\text{SERVER}})$, and the user has access to $\mathsf{vk}_{\text{SERVER}}$.
- On registration, the user initiates a connection with the server and sends their DH public value g^a.
- The server will respond with both DH public values (g^b, g^a) and a signature on both values, $\sigma_{\text{SERVER}} \leftarrow \mathsf{Sign}_{\mathsf{sk}_{\text{SERVER}}}(g^b \parallel g^a)$.
- All communication past this point takes place using secret-key authenticated encryption. Note that we do not implement the ratchet in this or future assignments.
- Next, the user will send their id_i to the server.
- The server will generate a random 128-bit salt_i for this user and send it to the user.

- The user will use the salt to generate $h_i := H(\mathsf{pwd} \parallel \mathsf{salt}_i)$ and send h_i to the server.
- The server then generates a random 8-bit pepper_i and generates $h'_i = H(h_i \parallel \mathsf{pepper}_i)$.
- The server then generates a PRF seed seed_i and sends it to the user for use in 2FA.
- The user generates a 2FA response $r := \mathsf{PRF}_{\mathsf{seed}_i}(t_{\text{NOW}})$, where t_{NOW} is rounded down to the nearest second. We can also think of it as a PRG with a random seed seed_i.
- The server verifies that this response is valid by checking the past 60 seconds of PRF responses.
- The user generates an RSA keypair $(\mathsf{vk}_i, \mathsf{sk}_i)$ and sends vk_i to the server for signing.
- The server generates a certificate for this user σ_i over the fields $(\mathsf{id}_i, \mathsf{vk}_i)$, then sends it to the user.
- The server stores $(\mathsf{id}_i, h'_i, \mathsf{salt}_i, \mathsf{seed}_i)$ in the database.

1.4.2 Login

- On setup, the server has access to an RSA keypair $(\mathsf{vk}_{\text{SERVER}}, \mathsf{sk}_{\text{SERVER}})$, and the user has access to $\mathsf{vk}_{\text{SERVER}}$.
- On login, the user initiates a connection with the server and sends their DH public value g^a.
- The server will respond with both DH public values (g^b, g^a) and a signature on both values, $\sigma_{\text{SERVER}} \leftarrow \mathsf{Sign}_{\mathsf{sk}_{\text{SERVER}}}(g^b \parallel g^a)$.
- All communication past this point takes place using secret-key authenticated encryption; note that we do not implement the ratchet in this or future assignments.
- Next, the user will send their id_i to the server.
- The server will retrieve $(\mathsf{id}_i, h'_i, \mathsf{salt}_i, \mathsf{seed}_i)$ from the database and sends salt_i to the user.
- The user will use the salt to generate $h_i := H(\mathsf{pwd} \| \mathsf{salt}_i)$ and send h_i to the server.

- The server then tries all possible 8-bit pepper_i and generates $\hat{h}'_i = H(h_i \,\|\, \mathsf{pepper}_i)$ until one matches h'_i.
- The user sends a 2FA response $r := \mathsf{PRF}_{\mathsf{seed}_i}(t_{\text{NOW}})$, where t_{NOW} is rounded down to the nearest second.
- The server verifies that this response is valid by checking the past 60 seconds of PRF responses.
- The user generates an RSA keypair $(\mathsf{vk}_i, \mathsf{sk}_i)$ and sends vk_i to the server for signing.
- The server generates a certificate for this user σ_i over the fields $(\mathsf{id}_i, \mathsf{vk}_i)$, then sends it to the user.

1.4.3 Communication

- On setup, both users should have registered and obtained a certificate, and both users have access to $\mathsf{vk}_{\text{SERVER}}$.
- On startup, both users run Diffie-Hellman, signing every message with the certificate they received from the server.
- Upon receipt of a DH public value from the other party, each user verifies that the server's signature on the certificate is valid, and that the user's signature on the public value is valid.
- Following these steps, the users have come to a shared secret and use symmetric-key authenticated encryption using these values.

2 Assignment Specification

Please note: you may NOT change any of the function headers defined in the stencil. Doing so will break the autograder. If you don't understand a function header, please ask us what it means and we'll be happy to clarify.

2.1 Functionality

You will primarily need to edit 🗀 `src/drivers/crypto_driver.cxx`, 🗀 `src/pkg/user.cxx`, and 🗀 `src/pkg/server.cxx`. The following is an overview of relevant files:

- 🗀 `src/cmd/user.cxx` is the main entrypoint for the `auth_user` binary. It calls the `User` class.
- 🗀 `src/cmd/server.cxx` is the main entrypoint for the `auth_server` binary. It calls the `Server` class.
- 🗀 `src/drivers/crypto_driver.cxx` contains all of the cryptographic protocols we use in this assignment.
- 🗀 `src/pkg/user.cxx` Implements the `User` class.
- 🗀 `src/pkg/server.cxx` Implements the `Server` class.

The following roadmap should help you organize concerns into a sequence:

- **RSA Signatures**: Implement RSA key generation, signing, and verification.
- **Revamped Diffie-Hellman**: Implement our modified DH key exchange protocol in registration and login.
- **Register/Login**: Implement register functionality to add new users to the system and login functionality to verify old users.
- **Communication**: Implement communication functionality to allow users to talk to each other.

Some tips:

- The `encrypt_and_tag` and `decrypt_and_verify` functions are wrapper functions that should cut down on the amount of repetitive code in your implementation. Use these functions to save yourself a lot of debugging time — do not call `AES_*` or `HMAC_*` functions raw!
- Remember to call `network_driver->disconnect()` at the end of the `ServerClient::HandleConnect` and `DoLoginOrRegister(...)` functions.
- If a protocol fails for any reason (e.g. invalid signature, incorrect keys, decryption failed, etc.), throw an `std::runtime_error`. On the user side, make sure you

disconnect the network driver before throwing an error.

- Use our constants from `include-shared/constants.hpp` where applicable. In particular from now on, Diffie-Hellman parameters are now hard-coded here instead of exchanged.

- Use the `chvec2str` and `str2chvec` functions to convert to and from strings and bytevecs, and `byteblock_to_string` and `string_to_byteblock` to convert to and from strings and byteblocks.

- You don't need to replicate our CLI functionality: however, using it as a debugging tool is helpful.

- Use our function `crypto_driver->nowish()` to get the time rounded down to the nearest second.

- Use our function `crypto_driver->hash()` to hash values.

- If you're debugging and want a fresh database, you are free to simply delete the database file at `keys/server.db`. Alternatively, the database driver has a `reset_tables` function which will do just this.

- Remember to use the functions in `include-shared/keyloaders.hpp` to save any keys, seeds, or certificates you may have obtained.

2.1.1 RSA Signatures

Implement RSA Signatures by editing the following functions. Once you do so, your clients will be able to verify the integrity of messages sent between them.

Cryptographic functions:

- `CryptoDriver::RSA_generate_keys()`

- `CryptoDriver::RSA_sign()`

- `CryptoDriver::RSA_verify()`

2.1.2 Revamped Diffie-Hellman

Implement our updated Diffie-Hellman key exchange protocol by editing the following functions. Once you do so, your clients will be able to come to a shared secret without risk of a man-in-the-middle attack occurring.

Application functions:

- `ServerClient::HandleConnection(...)`
- `ServerClient::HandleKeyExchange(...)`
- `UserClient::HandleServerKeyExchange()`
- `UserClient::HandleUserKeyExchange()`

2.1.3 Register/Login

Implement registration and login by editing the following functions. Once you do so, your clients will be able to register and verify new users.

Application functions:

- `ServerClient::HandleConnection(...)`
- `ServerClient::HandleLogin(...)`
- `ServerClient::HandleRegister(...)`
- `UserClient::HandleServerKeyExchange()`
- `DoLoginOrRegister(...)`

2.2 Communication

Implement communication by editing the following functions. Note: You should be able to reuse a lot of code from Signal, especially in communicating between users.

Application functions:

- `UserClient::HandleUserKeyExchange(...)`

2.3 Support Code

Read the support code header files before coding so you have a sense of what functionality we provide. This isn't a networking class, nor is it a software engineering class, so we try to abstract away as many of these details as we can so you can focus on the cryptography.

The following is an overview of the functionality that each support code file provides.

- `src/drivers/db_driver.cxx` implements a class to manage database connections and operations: use this instead of interacting with the database directly. We use `sqlite3` under the hood, so you can run `sqlite3 <dbpath>` to debug the database directory if necessary.
- `src/drivers/repl_driver.cxx` implements a convenience class to run different REPL commands.
- `src-shared/drivers/config.cxx` implements a class to load configuration files.
- `src-shared/keyloaders.cxx` implements a class to load keys.
- `src-shared/util.cxx` contains a variety of utility functions which you may find useful.
- Everything else from prior assignments is unchanged.

2.4 Messaging

The following example shows how to use our new encryption helpers alongside our networking library; we'll be using this pattern for the entire course, so it's good to get it down now.

```
1 // Declare the message struct that we want to send and
2 // populate its fields
3 Message msg_s;
4 msg_s.value =  foo ;
5
```

```
// Encrypt and tag the message: this serializes the message,
// encrypts it using AES, tags it with an HMAC, and
// rolls the IV into one convenient vector.
std::vector<unsigned char> message_bytes =
    crypto_driver->encrypt_and_tag(AES_key, HMAC_key, &msg_s);

// Send it away!
network_driver->send(message_bytes);

// -----

// Receive the message
std::vector<unsigned char> raw_data = network_driver->read();
auto msg_data = crypto_driver->decrypt_and_verify(AES_key, HMAC_key, raw_data);

// Check if MAC verification succeeded
if (!msg_data.second) {
  throw std::runtime_error( Invalid MAC! );
}

// Deserialize the data into a message.
Message msg_s;
msg_s.deserialize(msg_data.first);
```

2.5 Libraries: CryptoPP

You may find the following wiki pages useful during this assignment:

- CryptoPP RSA Signatures
  - In particular, we follow the "Signature Scheme with Appendix (Filters)" section with the `PUT_RESULT` flag
  - See SignatureVerificationFilter for how to use filters with `PUT_RESULT`. If you prefer `THROW_EXCEPTION`, that is also fine but you will need to alter the corresponding flag in `CryptoPP::RSA_verify`

- CryptoPP Hash Functions
- CryptoPP SHA-256
- CryptoPP Random Number Generators

3 Getting Started

To get started, get your stencil repository here and clone it into the 🗀 `devenv/home` folder. From here you can access the code from both your computer and from the Docker container. We suggest you also check out the Debugging Guide for tips on debugging common issues.

To prevent 🗀 `crypto_driver.cxx` solutions to earlier assignments being leaked in later assignments, we ask that you copy your code from `crypto_driver` functions implemented in the last assignment into this one. The functions you copy over now will not need to be copied over in the following assignment.

3.1 Running

To build the project, `cd` into the 🗀 `build` folder and run `cmake ..`. This will generate a set of Makefiles building the whole project. From here, you can run `make` to generate a binary you can run, and you can run `make check` to run any tests you write in the 🗀 `test` folder.

To run the user binary, run `./auth_user <config file>`. We have provided user config files for you to use: you shouldn't need to change them unless you would like to experiment with more users. Afterwards, you can either choose to `login`, `register`, `connect`, or `listen`; the former two deal with other server binaries, the latter two deal with other user binaries. They call the corresponding `Handle` functions in code.

To run the server binary, run `./auth_server <port> <config file>`. We have provided server config files for you to use: you shouldn't need to change them. Afterwards, the server will start listening for connections and handle them in separate threads.

3.2 Testing

You may write tests in any of the 🗀 `test/**.cxx` files in the Doctest format. We provide 🗀 `test/test_provided.cxx`, feel free to write more tests in this file directly. If you want to add any new test *files*, make sure to add the file to the `cmake` variable, `TESTFILES`, on line 7 of 🗀 `test/CMakeLists.txt` so that `cmake` can pick up on the new files. Examples have been included in the assignment stencil. To build and run the tests, run `make check` in the 🗀 `build` directory. If you'd like to see if your code can interoperate with our code (which is what it will be tested against), feel free to download our binaries here - we try to keep these up to date, so if you're unsure about the functionality of our binaries, please ask us on Ed!

3.3 Collaboration

Collaboration is allowed and encouraged! However, at the end of the day, you *must* write up the solutions yourself and acknowledge any collaborators in your `README.md`.