Programming Assignment 3 Winter 2014

CS 255: Intro to Cryptography

Prof. Dan Boneh Due **Mar. 12, 11:59pm**

1 Introduction

When security-sensitive systems are deployed across a network, they can be subject to many subtle attacks by both passive and active adversaries, requiring meticulous attention to detail in designs and security proofs. To make this task easier, it is often convenient to rely on the basic primitive of a *confidential channel with mutual authentication*: that is, a protocol that provides a communication channel between two parties, Alice and Bob, such that no attacker can learn anything about messages sent between Alice and Bob, and for all of the messages the protocol delivers to Bob, it is guaranteed that Alice indeed sent those messages in that order (and vice versa).

In real life, secure channels are often realized by a protocol called TLS (Transport Layer Security, also known as SSL or Secure Sockets Layer in previous incarnations). Just as TCP abstracts out the details of establishing reliable communication over an unreliable underlying channel, TLS abstracts out the details of establishing secure communication over an insecure underlying channel. With proper usage, one can set up a TLS connection between a client and a server, and send messages back and forth between them, relying on the fact that the messages are authentic and hidden from attackers, without needing to design or prove the relevant cryptographic properties individually for each application.

In this project, we provide a skeleton implementation of a server listening for TLS connections, as well as a client who connects to that server over TLS. Our objective will be to ensure that this communication is mutually authenticated, so that after our setup procedure, the client can make interactive, security-sensitive queries to the server and receive security-sensitive results, while being assured of both confidentiality and authenticity. Since TLS has internal mechanisms to easily achieve *one-way* authentication—authenticating the server to the client, by way of *certificates* and trust chains—the main task of the project will be to complete *mutual* authentication for the connection, using a simple signature-based challenge-response authentication protocol to authenticate the client to the server.

2 Design overview

The system for this project comprises two components: a server that listens for TLS connections (implemented in `server.js`) and a client that connects to the server over TLS (implemented in `client.js`). We describe the client and server implementations below.

2.1 Generating credentials

Recall that in one of the steps of the TLS handshake, the server presents a certificate signed by a certificate authority (CA) to the client. The client then verifies that the server's certificate is valid and is properly signed by a trusted party (in this case, a CA). In this project, we provide a script that allows you to create your own CA for testing purposes. The script will also generate a server certificate, signed by your CA. In addition to creating a CA and generating a server certificate, the script will also generate a client public/private key pair to be used for the signature-based challenge-response component of the protocol. In this project,

you will use the elliptic-curve digital signature algorithm (ECDSA) as the primitive in your signature-based challenge-response system.[1]

To execute the script, run `python gen_all.py`. The script will prompt you for three passwords that will be used to protect the CA signing key, the server's private key, and the client's private key. Make note of the passwords you use, since you will need to include them later in the test script `test.js`, when you test your implementation.

2.2 General implementation details

We provide several helper functions in `lib.js`, including all those from Project 2 and a few new ones. Notably, there is a `send_message` function that you should use for client-server communication. The first argument to `send_message` is a socket (that the client and server communicate over). The second and third arguments specify the type and contents of the message, respectively. In this assignment, we define four types of messages:

- `CHALLENGE`: challenges from the server
- `RESPONSE`: responses from the client
- `SUCCESS`: sentinel denoting a successful challenge-response authentication
- `SESSION_MESSAGE`: messages sent during the actual session layered on top of TLS, once the session has been established successfully

Note: When sending messages between client and server, you should always use the `send_message` function; there is no need to communicate directly over the sockets.

2.3 Server implementation

The server constructor takes four arguments: the server's private key, the password used to protect the server's private key, the server's certificate, and the client's public key (used for challenge-response authentication). We provide code to load the server's public key (in `test.js`) as well as code to deserialize the data (`unwrap_client_pub_key()`).

The server implementation consists of a single public method, `server.start`, that you will complete. When the server starts up, it will listen for TLS connections on the specified port. You will use the `TLS` library for `node.js` to set up the server. Documentation for the `TLS` library is available at http://nodejs.org/api/tls.html. As part of the server setup process, you will need to determine the necessary arguments (`server_options`) to pass to the `tls.createServer` method.

To simplify the implementation, we will assume that only one client instance at a time will connect to the server. If the server is already connected to a client, it will reject all additional connections. The `TLS` library will handle all of the details of the TLS handshake. Thus, when a client connects to the server, you are only responsible for writing the challenge-response component of the protocol. Specifically, you will implement the following protocol:

1. When a client connects, the server will generate a new challenge and send the challenge to the client (via a call to `send_message` with a message of type `CHALLENGE`). In this project you will be building a signature-based challenge-response system. Specifically, to generate a challenge, the server will choose

[1] While we have not discussed ECDSA in class, you do need to understand any of the internal details of ECDSA to complete this project. We provide functions to load the public/private ECDSA keys from disk as well as abstractions for signing and verification.

a message ℓ and ask the client to compute a signature on ℓ. Recall that with signatures, signing is a private-key operation while verification is a public-key operation. The assumption is that only the holder of the secret key is able to compute a valid signature. In the short-answer section, we will walk you through an argument that shows how we can bootstrap a one-way confidential channel to a session that provides two-way authenticity of messages, using a challenge-response system.

2. Once the challenge has been issued, the server should wait to receive a message of type `RESPONSE` from the client. If a message with any other type is received, the server aborts and closes the connection. In signature-based challenge-response, the client's response r will be a signature on the challenge message ℓ: that is, $r \leftarrow \mathsf{Sign}(\mathsf{csk}, \ell)$, where csk is the client's secret (ECDSA signature) key.

3. If the client replies with a message of type `RESPONSE`, the server checks that the response is valid for the challenge it sent to the client. If the response is invalid, the server aborts the protocol and closes the connection. If the response is valid, the server should send a message of type `SUCCESS` to the client (with an empty message content). In signature-based challenge-response, the server will verify the response r for challenge ℓ, by evaluating $\mathsf{Verify}(\mathsf{cpk}, \ell, r)$.

4. From this point forward, the session has been successfully established, and the server and client can exchange session messages. During this stage, whenever the server receives a message of type `SESSION_MESSAGE`, it should simply send back the length of the session message. (For example, in our testing code, the client sends "Hello World!", and the server responds with "12".)

In general, if at any point, the server receives a message it was not expecting, then it should immediately abort and close the connection. (In the starter code, we provide the code that handles these failures, via a function called `protocol_abort`.)

2.4 Client implementation

The client constructor takes four arguments: the client's private key, the password used to protect the client's private key, the certificate for the certificate authority (CA), and optionally, a "name" (for logging purposes). As with the server implementation, we provide code to load the client private key from disk (in `test.js`), as well as code to deserialize and decrypt the private key (`unwrap_client_sec_key()`).

Similar to the server implementation, the client implementation will export a single public method: `client.connect`. When called, the `connect` method should initiate a TLS connection to the server at the specified `host` and `port`. You will again use the `TLS` library to do this. Part of the assignment will be to figure out the proper parameters (`client_options`) that you will need to pass to the `tls.connect` function to start the TLS connection. As a hint, note that by default, the client will not trust your custom CA. You will need to *pin* your CA certificate in order for the client to accept connections to a server that presents a certificate signed by your CA.

As part of the TLS handshake, the server will present its certificate, signed by your CA. As part of this assignment, you will write a function to perform some basic validation of the certificate. Specifically, you will need to check the following conditions:

- Check that the certificate contains the following fields: `valid_from`, `valid_to`, `issuer`, `subject`, `fingerprint`.
- Check that the current time is within the certificate's validity window (that is, the current time is after `valid_from` and before `valid_to`).
- Check that the certificate will not expire within the next seven days.
- Check that the certificate's subject contains the following field values:

- Country Name [C]: US
- State or Province Name [ST]: CA
- Locality Name [L]: Stanford
- Organization Name [O]: CS 255
- Organizational Unit [OU]: Project 3
- Common Name [CN]: localhost
- Email Address [emailAddress]: cs255ta@cs.stanford.edu

If any of the above checks is not satisfied, then the client should abort (via the function called `protocol_abort`).

Once the client successfully negotiates a TLS session, the server will generate a challenge ℓ, and send the client a message whose type is `CHALLENGE`, and whose value is a string representation of the challenge ℓ.

Note: How you generate and represent the challenges is up to you. However, your system will need to satisfy the security requirements described in Section 3, and all messages you send over the network should consist only of strings of **valid, printable ASCII characters**. In addition, regardless of how you generate the challenges, it should be the case that when your server handles n sequential sessions (client connections), it should still use **a constant amount** of true randomness (independent of n)—in order to avoid exhausting the entropy pool, which in Javascript is fairly scarce.

Upon receiving a `CHALLENGE`, the client will need to generate a response (a message of type `RESPONSE`), and send it back to the server. As mentioned in the previous section, in our signature-based challenge-response system, the challenge will be a message ℓ and the proper response r will be a signature on m: that is, $r \leftarrow \mathsf{Sign}(\mathsf{csk}, \ell)$. Assuming the server receives a valid response from the client, it will send back a message with type `SUCCESS`. Once the client receives this message, setup is now complete. In this project, the client should send a single message of type `SESSION_MESSAGE` with contents `Hello World!`, then immediately end the session. (In our partial implementation in the starter code, we already include code for the client to perform this simple session, in the function `run_test_session`.)

In general, if at any point, the client receives a message it was not expecting, then it should immediately abort and close the connection. (In the starter code, we provide the code that handles these failures, via a function called `protocol_abort`.)

3 Security requirements

In your implementation, you will need to complete the files `client.js` and `server.js` to set up an interactive session on top of TLS, implementing the behaviors described in Sections 2.3 and 2.4 (with formal APIs described in Appendix A).

The standard security definition for a TLS-like session is that it achieves a **mutually-authenticated confidential channel**. Unfortunately, we cannot specify this definition formally without introducing the simulation paradigm, which is outside the scope of this course. However, your session should **at least** satisfy the following weaker security properties:

- A confidentiality-like property (P1): An active attacker on the network should not be able to distinguish between the network messages exchanged for any two sessions, S_0 and S_1, if the same number of messages (at the same times, of the same lengths) were exchanged in the implementation of both S_0 and S_1.
- An authenticity-like property (P2): An active attacker on the network should not be able to cause the client (resp., server) to report receipt of any sequence of session messages that was not (possibly a prefix of) the sequence of session messages that the server (resp., client) actually sent.

Having stated these properties, we now specify the requirements for your implementation:

1. If the underlying network reliably and correctly transports messages, then your implementation should provide a correct session—i.e., authentication should succeed, and each party should receive exactly the sequence of session messages sent by the other party.
2. Even in the presence of an active attacker on the network, your implementation should provide a channel for the session messages that satisfies (P1) and (P2).
3. Properties (P1) and (P2) should still hold even for the very strong attacker in short answer question 5(b), who has managed to trick the CA into issuing one temporary bad certificate. In other words, the system need not (and cannot!) be secure during the time the bad certificate is valid, but as soon as it expires, even an active attacker must not be able to break (P1) or (P2) on **new** connection instances.
4. If an active attack is detected, your implementation is allowed to abort the protocol (this is done via the function `protocol_abort`, which is already provided and used for this purpose in the starter code). However, **even an active attacker** should not cause either the client or the server process to crash. (Crashing might happen due to an unhandled exception, for instance. Note that crashing is not the same thing as deciding to abort the protocol.)

You are not required to prove these properties (except the parts that are addressed in the final short answer question), but the properties should still hold in the system you implement.

4 Short-answer questions

In addition to your implementation, you should include answers to the following questions. Your answers need not be long, but should include important details.

1. In our challenge-response system, the server chooses a challenge ℓ, and the client responds with $r = \mathsf{Sign}(\mathsf{csk}, \ell)$; the server verifies by running $\mathsf{Verify}(\mathsf{cpk}, \ell, r)$. What if, instead, the client were to respond with a hash of the challenge, $r = \mathsf{SHA\text{-}256}(\ell)$, and the server were to verify that $r = \mathsf{SHA\text{-}256}(\ell)$? Either give an attack on this modified protocol, or give a high-level overview of an argument that it is as secure as the original protocol.
2. Why is there no need for the client to check the "Issuer" field of the server's X.509 certificate?
3. Examine the code we have written for you in the function `unwrap_client_sec_key` (in `client.js`). Briefly describe any two mistakes (or other design decisions) we could have made in storing and/or loading the client secret key from disk, such that the system would still work correctly, but might not be provably secure against an attack (in some model) that it currently is secure against.
4. Instead of public-key (signature-based) challenge-response, we might have used symmetric-key (MAC-based) challenge-response.
   (a) What is one advantage of using symmetric-key challenge-response here?
   (b) What is one disadvantage? (Hint: suppose that a server has many clients, and a client wants to connect to many servers, but each has only a very small amount of trusted storage.)
5. Instead of public-key (signature-based) challenge-response, we might have used simple password authentication (i.e., the client just sends the same password over the TLS connection every time).
   (a) Briefly describe how such an authentication scheme would work. What information should the server store (instead of the client's public key)? Why shouldn't this information just be the client's password?

(b) Suppose that an attacker has tricked the CA into issuing a single "bad" certificate, claiming that the server's public key is a public key generated by the attacker. However, this bad certificate expires after a short time. Show that if the system uses simple password authentication, then the attacker might be able to connect to the server even after the bad certificate's expiration date.

5 Summary of requirements

- You should complete the files `client.js` and `server.js` to set up an interactive session on top of TLS, implementing the behaviors described in Sections 2.3 and 2.4 (with formal APIs described in Appendix A). You need not document your implementation, except for answering the short answer questions (unless your implementation choices are unusual or will be unclear without documentation).
- You should address the following systems issues as you complete your implementation:
  - Your server should be capable of handling multiple sequential sessions (client connections) correctly. It is not required to handle multiple concurrent (i.e. simultaneous) sessions. As written, our starter code should already drop attempts to set up concurrent connections.
  - Messages you send over the network (including string representations of challenges and responses) should consist only of **valid printable ASCII characters**.
  - When your server handles n sequential connections, your server should still use **a constant amount** of true randomness (independent of n)—in order to avoid exhausting the entropy pool, which in Javascript is fairly scarce.
  - In order to test your code, you may find it helpful to use and/or modify the test script `test.js`.
- You should include answers to the short-answer questions (Section 4) in a file called `answers.txt` or `answers.pdf`.

6 Logistics

If you have general questions, please post them on Piazza at https://piazza.com/class/hou9vrkx5ls5rw so that all students can benefit from the answer. If you have a question about your specific implementation or design, you should ask at office hours or post a private question on Piazza.

A API Description

A.1 Server API Description

A.1.1 `get_new_challenge()`

- Return: `string`

This function should generate a challenge (encoded as a string) to send to the client. Note that your string should only contain **valid, printable ASCII characters**.

A.1.2 `on_connect(connection_socket)`

- `connection_socket`: socket between client and server
- Return: nothing

This is a callback function that is called whenever a client makes a connection to the server. If there is already a client connected to the server, then the server immediately closes the connection. On a successful connection, the server must generate a new challenge and send it to the client.

A.1.3 `process_client_msg(json_socket)`

- `json_data`: JSON encoding of a message from the client. The input consists of two fields: `type` and `message`. The `type` field can take on four possible values (as explained in Section 2.2): `CHALLENGE`, `RESPONSE`, `SUCCESS`, and `SESSION_MESSAGE`. The type will be encoded as a string.
- Return: nothing

This is a callback function that is called whenever the server receives a network message. The server should check that the message is either of type `RESPONSE` or of type `SESSION_MESSAGE` (and abort the protocol otherwise). If the message received is of type `RESPONSE`, the server should verify that the response is valid, and respond with a `SUCCESS` message if so (again, aborting the protocol otherwise). On the other hand, if the message received is of type `SESSION_MESSAGE`, then it should log the message (using the provided function `server_log`), and send back (a string representation of) the length of the message, as described in Section 2.3.

A.1.4 `server.start(port)`

- `port`: integer representing the TCP port on which the server should listen
- Return: nothing

Creates a TLS server listening on the specified port.

A.2 Client API Description

A.2.1 `check_cert(crt)`

- `crt`: JSON encoding of the server certificate (`string`)

- Return: `boolean`

This function is responsible for validating the server's certificate. Specifically, you must verify the conditions described in Section 2.4. This function should return `true` if all of the conditions are satisfied. Otherwise, returns `false`.

A.2.2 `process_server_msg(json_socket)`

- `json_data`: JSON encoding of a message from the server. The input consists of two fields: `type` and `message`. The `type` field can take on four possible values (same as in `process_client_msg`: `CHALLENGE`, `RESPONSE`, `SUCCESS`, `SESSION_MESSAGE`).
- Return: nothing

This is a callback function that is called whenever the client receives a network message. The server should check that the message is of type either `CHALLENGE`, `SUCCESS`, or `SESSION_MESSAGE` (and abort the protocol otherwise). If the message received is of type `CHALLENGE`, the client should verify that the response is valid, and respond with an appropriate `RESPONSE` message if so (again, aborting the protocol otherwise). If the message received is of type `SUCCESS`, the client should invoke the callback that was originally provided as the argument `session_callback_f` to `client.connect`, indicating that the session has been successfully established. (In our testing code, this callback will then request the client to send `Hello World!` as its only session message.)

On the other hand, if the message received is of type `SESSION_MESSAGE`, then it should log the message (using the provided function `client_log`).

A.2.3 `client.connect(host, port, session_callback_f, session_close_callback_f)`

- `host`: The address or domain (in our case, always `localhost`) the client should connect to.
- `port`: The TCP port on which the client should connect.
- `session_callback_f`: A function that takes no arguments and returns nothing.
- `session_close_callback_f`: A function that takes no arguments and returns nothing.
- Return: nothing

Connects to a TLS server at the indicated host and port (and executes the challenge-response and session protocols described above). Saves the two callbacks, `session_callback_f` and `session_close_callback_f`, for later use during the protocol (the former will be called when a session is successfully established, and the latter when a session ends).

A.3 `client.get_state()`

- Return: `string`

This function returns the current protocol state. Possible values are `START, CHALLENGE, RESPONSE, SUCCESS, END, ABORT`.

A.4 `client.session_send(msg)`

- `msg`: string corresponding to the session message the client sends.
- Return: nothing

This method is used to send messages once the mutually authenticated session has been established. The client sends the specified message to the server.

A.5 `client.disconnect()`

- Return: nothing

This method ends the session.