Programming Assignment 2 Winter 2015

CS 255: Intro to Cryptography

Prof. Dan Boneh Due **Mar. 11, 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 client that connects to a cloud server (hosted on Amazon AWS) 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 consists of a single client that connects to our server over TLS (implemented in `client.js`). We first outline the interaction that occurs between the client and the server. We assume here that the client has already registered a public key with the server in a separate registration step which we describe in greater detail in Section 2.1. The public key will be used for a signature-based challenge response protocol. The interaction between the client and server then takes the following form:

1. First, the client will connect over TLS to `ec2-54-67-122-91.us-west-1.compute.amazonaws.com` on a port between `8800` and `8849`.
2. The server will present a certificate. The client should check that the server has presented a valid certificate signed by a trusted party (in this case, a CA).
3. Once a TLS connection has been established, the server will send a challenge to your client. In this project, we will be using a signature-based challenge-response protocol.

4. In a signature-based challenge-response protocol, the client will reply with its signature on the server's challenge.
5. The server will verify the client's signature. If the client's signature is valid, the server sends back a secret token. This completes the protocol.

2.1 Registration

When starting this project, your first task will be to register for an account on the server. By now, you should have received a registration token via email from the course staff. If you have not, please send an email to the TA list as soon as possible. To register for an account on the server, you will need to connect to port `8900` on `ec2-54-67-122-91.us-west-1.compute.amazonaws.com`, and provide the following information:

- Your SUNet username (not your ID number).
- A signature verification key that you will use for the challenge-response authentication. In this project, we will be using the ECDSA signature scheme.
- A signature on your registration token using your ECDSA signing key.

The registration server will then verify the signature on the registration token associated with your SUNet username. If registration was successful, the registration server will reply with a message `SUCCESS`. Otherwise, it will reply with a message `FAILED`. As in Project 1, for your convenience, we have provided you wrappers for the ECDSA signing function (see `lib.js`).

We have provided you a script that handles all of the registration steps in `register.js`. Before you can complete the registration however, you will have to fill in some required fields and functions in `register.js` (as described below):

- Fill in the `params` dictionary with your SUNet ID, your registration token, and a password that will be used to protect your secret signing key.
- Write the `sign_token` function that will compute a signature on your registration token. Your registation token is a hex encoding of a bitarray (as output by `bitarray_to_hex`). When computing your signature, you should first convert your token back to a bitarray before signing.

Once you have completed these two components, you should be able to run the registration script: `node register.js`. If all fields have been populated correctly, you should see a confirmation message that your registration was successful.

Note: If you are working with a partner on this assignment, only **one** of you needs to register. Likewise, you will only need to submit one assignment, with one secret token (described in Section 2.3).

2.2 General implementation details

We provide several helper functions in `lib.js`, including all those from Project 1 and a few new ones. While you will not need to write any of the client-server communication code, it will still be helpful to understand our interfaces. We provide a `send_message` function in `client.js` that handles all of the 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 three types of messages:

- `CHALLENGE`: challenges from the server

- `RESPONSE`: responses from the client
- `SUCCESS`: sentinel denoting a successful challenge-response authentication

2.3 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 your SUNet username. 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 (`lib.ECDSA_load_sec_key()`).

The client implementation exports 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`. In this project, you will use the Node.js `TLS` library to do this. The `TLS` library handles most of the TLS handshake, so you will not have to worry about most of the details of the handshake in your implementation. 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 our provided CA. The CA's certificate (`cs255ca.pem`) is included in the starter code. You will need to *pin* this certificate in order for the client to accept connections to a server that presents a certificate signed by our CA.

As part of the TLS handshake, the server will present its certificate, possibly by signed by our 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 120 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 2
  - Common Name [`CN`]: `ec2-54-67-122-91.us-west-1.compute.amazonaws.com`
  - 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. In this project, the challenge ℓ will be a hex encoded bitarray (as output by `bitarray_to_hex`).

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`. This message will also contain a secret message x (in practice, this might be a session cookie). The secret message x will be a core component of your submission.

We have provided all of the client-server communication code for you in `client.js`. Your job will be to fill in the certificate checking and the response computation components of the code. These methods have been stubbed out and marked with `TODO` tags in `client.js`.

3 Additional Notes

We have provided you a test file `test.js` that connects to the server over TLS and echoes the messages it receives. You can use this to test your client.

The server on `ec2-54-67-122-91.us-west-1.compute.amazonaws.com` is known to be devious. There is a high probability that when your client connects, the server will present you with an invalid certificate. Refer to Section 2.3 for details on what constitutes a valid certificate. Your client should close the connection immediately if it receives an invalid certificate. You should only trust servers who have the right credentials to know the secret message x, so don't be fooled! Practically speaking, this means that you may need to connect to the server multiple times to learn the secret message.

Note: Our server can only handle a limited number of concurrent connections. Thus, you should try to complete the programming component of this project as early as possible. If you wait until the last minute to try and obtain the secret token, you might not be able to complete the assignment on time. We will not accept late submissions because you did not have enough time to obtain the secret token from the server.

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, $r = \text{SHA-256}(\ell)$, and the server were to verify that $r = \text{SHA-256}(\ell)$? 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 in the library function `ECDSA_load_sec_key` (in `lib.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 will need to complete the file `register.js` in order to register with the server. We have included `TODO`s in `register.js` that specify what needs to be done. The necessary modifications are also described in Section 2.1.
- You will need to complete the file `client.js` to set up an interactive session on top of TLS, implementing the behaviors described in Section 2.3 (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).
- Any message you send over the network should consist only of **valid printable ASCII characters**.
- You should include answers to the short-answer questions (Section 4) in a file called `answers.txt` or `answers.pdf`.
- You should include your secret message x you receive from the server in a separate file named `SECRET`. You should **not** include any other information in `SECRET`. The autograder will be grading this component, and the autograder does not award partial credit!

6 Questions?

If you have general questions, please post on them on Piazza at `https://piazza.com/class/i4jbd95to3hgp` so that all students can benefit from the answer. If you have a problem with your specific implementation, come to office hours or ask a private Piazza question.

7 Acknowledgments

This project is supported by an Amazon AWS in Education Grant.

A Client API Description

A.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 2.3. This function should return `true` if all of the conditions are satisfied. Otherwise, returns `false`.

A.2 `compute_response`

- `challenge`: hex encoding of the server's challenge
- Return: `string`

This function is responsible for computing the response to the server's challenge. When computing the signature, you will need to first convert the challenge into a bitarray. The response should consist of the signature on the server's challenge encoded using hex.

A.3 `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 three possible values: `CHALLENGE`, `RESPONSE`, `SUCCESS`.
- 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 `CHALLENGE` or `SUCCESS` (and abort the protocol otherwise). If the message received is of type `CHALLENGE`, the client should verify that the accompanying message is valid, and respond with an appropriate `RESPONSE` message if so (again, aborting the protocol otherwise). If the client receives a message with type `SUCCESS`, then authentication has completed and the session has been setup. In this assignment, the server will also include a secret token (e.g., a session cookie) with the `SUCCESS` message.

A.4 `client.connect(host, port)`

- `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.
- Return: nothing

Connects to a TLS server at the indicated host and port (and executes the challenge-response protocol described above).