EPOCH Protocol

Endpoint Oriented Channel

Abstract.

The EPOCH protocol is based on the XS_ACE ‘Arbitrary Code Execution’ subsystem of TF Protocol, in fact, is a subset of XS_ACE (see TF Protocol documentation for further reading). The idea behind EPOCH is to provide an abstraction layer for communication and encryption allowing the construction of functionalities or commands ‘Called endpoints in EPOCH’ on top of it. Those endpoints are programs that can be written in any interpreted or JIT language, such Python or Java; or any AOT language like C, C++, Pascal, and so on. As far as concern to the ‘endpoint’, this is, the program running on top of EPOCH, only matters reading from standard input and writing to the standard output.

Overview

EPOCH Protocol works by exchanging data with the server in a TCP connection. The default buffer used for the communication is 65536 bytes long if not has been specified a greater one in the endpoint string construction.

The communication between the server and the client occurs in an encrypted way. This encryption is a client-to-server one. It uses at first public key encryption to exchange a session key. Thereafter this session key is used for the encryption to transmit and receive data.

Every sent or received data must be preceded by a header indicating the size of the payload that coming next. The header of EPOCH Protocol is a 64bit signed integer. Before sending it must be converted to network byte order (big-endian).

Beginning the connection

The communication start with the client sending a random generated session key encrypted with the public server key. Hereinafter the client start to using the previous generated session key to encrypt/decrypt data.

After the client sent the session key, it has to send the connection string. Once the server receive the connection string and process it, it will send back to the client a header with value 0 which means communication can start flowing. If any error occur, the server will send a non-zero value and communication is over.

The encryption system

The EPOCH Protocol transmits and receives data encrypted. The client must have the public key of the server. The RSA must be 2048 bit and RSA_PKCS1_OAEP_PADDING padding. The procedure of exchanging keys and generating session keys occurs as follows:

1) The client generates an arbitrary session key which max length cannot exceed 2048 / 8 – 42 bytes.
2) The client encrypts the session key with the server's public key and sends it.
3) The server decrypts the session key with its own private key.

If there is an error in this process the connection is terminated by the server.

The encryption session key is an array of n characters up to 2048 / 8 – 42 that is generated in a random way and it will be xored with the data sent to or received from the server.

The only requirement for the algorithm to xoring the data is to be symmetric at both side, client and server.

The connection string

The connection string contains the parameters to inform the server about the endpoint to execute and other important parameters. It look like this one:

endpoint{/bin/ls -la}auth{auth_token}mode{single_thread_sndfirst}bufsz{65536}

endpoint: The program to execute at the server side.
auth: The authorization token.

It is free to the implementation the mechanism of authorization. However, it must be guaranteed that the auth_token is related to the binary indicated by ‘endpoint’. For example, auth_token could be a file where resides all binaries allowed to run under that token.

mode: The mode indicates how the server will interact with the client and the program being executed ‘the endpoint’. It has three options:

- single_thread_sndfirst
- single_thread_rcvfirst
- multi_thread

- single_thread_sndfirst: This mode tells the server to receive data first from the client side and send it to the program being executed ‘the endpoint’. When is done, it will start to receive data from the endpoint and send it back to client side.

- single_thread_rcvfirst: This mode tells the server to receive data first from the endpoint and send it back to client side. When is done, it will start to receive data from the client side and send it to the endpoint.

- multi_thread: This mode tells the server that it has to receive data from the client side and send it to the endpoint, and at the same time, receive data from the endpoint and send it back to client side.

bufsz: This is the buffer intended for communication. If the specified buffer is less than 65536, the default is used.

The interprocess communication

The communication between the server and the endpoint ‘the program being executed’ can be achieved by any means that guarantee that the program executed can read the data sent by the server from the standard input, as well as it can write the data to the server trough the standard output.

No matter which ‘mode{single_thread_sndfirst or any other}’ is used, the dynamics of the process is as follow:

Before every payload sent by the client or the server, a header has to be sent indicating the amount of data that is coming next. This header as stated before is a 64bit signed integer normalized ‘converted to big-endian’. The server or the client will continue sending/receiving data until a header with value 0 shows up. In case of receiving 0 as a header it means that the other side does not have any more data to send, and that receiving operation can consider completed. In case of sending, when there is no more data to send, a header will value 0 will be sent.

At the server side must be a way for the endpoint ‘the program being executed’ and the server to inform each other when an operation ‘read/write’ is done just like the server and the client inform to each other by sending a header with value 0 through the network. The way in which the server and the end point tell each other that reading or writing is done is by closing the corresponding descriptor.

We will assume pipes as the interprocess communication to explain the dynamic of exchanging data between the endpoint and the server.

The server will keep reading data from network and send it to the standard input of the endpoint until it receives 0 as the value of the header. At that

point the server closes the write-end of the pipe. This is reflected inside the endpoint due the standard input being closed.

In the same way, the server will keep reading data from the endpoint and send it to the client until the endpoint closes its standard output, at that point the server sends 0 as the value of the header to the client. The closure of the endpoint standard output is reflected inside the server due the pipe read-end being closed.

It is important to know that the server will set the same descriptor to the standard output and the standard error. This means that the endpoint must close both descriptors in order to inform the server that it's write operations are done.

Now the server must send to the client a 32bit header converted to big-endian with the endpoint exit status.

At this point a two-way handshake must take place in order to allow the other end to finish the reading operation. If **single_thread_sndfirst** was used, after the server sent all data and the endpoint exit status, it must wait for a 64bit header sent by the client. If **single_thread_rcvfirst** was used, after the server receive all data and sent the endpoint exit status, it must write a 64bit header. If **multi_thread** was used, after the server sent and receive all data; and sent the endpoint exit status, it must wait for a 64bit header and right after sent a 64bit header. The value of the header used for this handshake really does not matter.

Note of caution: The client and the endpoint must keep in mine that one write or read at one end is not necessary equivalent to one read or write at the other end. It's crucial for both to continue reading, in the case of client, until it reads from network a header with value 0; in the case of endpoint, until **read primitive** returns EOF or any signal indicating its standard input closure.