Crypto White-Hat Write-Up

Daniel Petti

December 2018

1 Introduction

All of my work on this project was focused on designing and then implementing a secure handshake protocol. The protocol uses PKC to authenticate both parties and eventually to exchange a session key, which is then used to encrypt the remainder of the messages. Both the protocol and the implementation are designed to be flexible, so that they can be used with arbitrary cryptosystems. Therefore, in this write-up, both will be described in a general sense, with no reference to specific cryptosystems.

2 Protocol Design

The handshake protocol is modeled off of SSL, with some slight modifications. The messages that are sent are as follows:

1. **client_hello:** {pkc: $[p_1, p_2, ...p_n]$, sym: $[s_1, s_2, ...s_n]$}
2. **server_hello:** {pkc: p, sym: s, pub_key: K_{us}}
3. **client_challenge:** {key: e_s, pub_key: K_{uc}, chal: c_1, mac_key: e_m}
4. **server_challenge:** {chal: c_2, resp: d_1}
5. **client_verify:** {resp: d_2}

2.1 client_hello Message

The *client_hello* message is the first message send from the client to the server. It is meant to establish what cryptosystems the client is capable of using.

2.1.1 pkc

This parameter is a list of the cryptosystems that the client is capable of using. It is sent in the clear.

2.1.2 sym

This parameter is a list of the symmetric cryptosystems that the client is capable of using. It is sent in the clear.

2.2 server_hello Message

The *server_hello* message is sent by the server to indicate what cryptosystems will be used for the remainder of the session. The cryptosystems to use are chosen by the server based on the mutual capabilities of the server and the client.

2.2.1 pkc

This parameter is the name of the PKC that will be used for this session. It is sent in the clear.

2.2.2 sym

This parameter is the name of the symmetric cryptosystem that will be used for this session. It is sent in the clear.

2.2.3 pub_key

This is the server's public key for the selected PKC. It is sent in the clear.

2.3 client_challenge Message

The *client_challenge* message is sent by the client. It is the first part of the process that the client uses to verify the server's identity.

2.3.1 key

This is the session key to use. It is randomly generated by the client, and sent encrypted using the server's public key.

2.3.2 pub_key

This is the client's public key. It is sent in the clear.

2.3.3 chal

This is a challenge value that is randomly generated by the client, and encrypted using the server's public key.

2.3.4 mac_key

This the key that will be used for the HMAC. It is generated randomly, and encrypted using the server's public key.

2.4 server_challenge Message

The *server_challenge* message is sent by the server. It serves both to continue the server identity verification procedure, and also to initiate the process by which the server verifies the identity of the client.

2.4.1 chal

This is a challenge value that is randomly generated by the server, and encrypted using the client's public key.

2.4.2 resp

This is the challenge value that the client previously sent. The server decrypts it, and then re-encrypts it using the session key before sending it back. This procedure ensures both that the server knows its own private key, and that the session key has been properly shared.

2.5 client_verify Message

The *client_verify* message is sent by the client. It contains only a response to the server's challenge value, completing the handshake procedure.

2.5.1 resp

This is the response to the server's challenge. The client first decrypts the challenge, and then re-encrypts it with the session key before sending it back to the server.

2.6 Encrypted Fields

In this protocol, all fields that are encrypted automatically also have a nonce and MAC added. First, the nonce is concatenated with the plaintext, and the MAC of the concatenation is taken. This MAC is then concatenated with them. Finally, the entire thing is encrypted, and the resultant ciphertext is sent over the wire. The receiver verifies that both the MAC and nonce are valid when decrypting the message. Furthermore, after the handshake completes, session messages encrypted using the session key undergo this same treatment.

2.7 Protocol Security

This protocol accomplishes several main goals:

- Allows the server and client to agree upon a mutually-supported cipher suite.
- Allows a session key to securely be shared between the two parties.

- Allows the server and client to verify each-other's identity.

Though the initial cipher suite negotiation messages are sent in the clear, most of the rest of the handshake is encrypted. Once a cipher suite is agreed upon, the server and client exchange public keys, and use these to encrypt the rest of the handshake. Additionally, the protocol is designed to settle on a session key as early as possible, so that some of the handshake protocol messages can also be encrypted using this key.

The protocol makes heavy use of a challenge-response design in order to verify the identity of both parties. Of course, without an actual CA, the amount of verification that can be done is very limited, and in this case, pretty much amounts to ensuring that each party actually knows the corresponding private key for their public keys. Even so, the protocol is still vulnerable to all the kinds of DoS and session hijacking attacks that not having a CA leaves one open to. It is known, however, that no other groups have CA's either, so consequently, little effort was made to address this vulnerability. Since it is so easy to exploit in every project, this likely won't garner the Black Hat team much in the way of points.

Furthermore, the protocol makes use of nonce values to guard against replay attacks. In this respect, the protocol design sacrifices some efficiency for simplicity and peace-of-mind: It mandates that every encrypted field have a nonce, even in cases where the nonce may not be necessary. A similar philosophy was used with MAC values. Every encrypted field has a corresponding MAC to guard against garbled messages, whether that's technically necessary or not.

3 Implementation

The implementation was focused on enabling modularity and ease of extension. From the start, a major goal was to ensure that new cryptosystems, nonce schemes, and MAC algorithms could be implemented with ease, and would "just work" with the existing infrastructure. To that end, the resulting system employed "interface" classes. These were special classes with stubbed-out methods that the actual implementations of cryptographic primitives were expected to subclass and override, respectively. This offered the major advantage of allowing the handshake system to work with pretty much any combination of primitives out-of-the-box. Since the division of labor in our team also resulted in different people working on the implementations of the primitives and the implementation of the handshake, this design greatly simplified our lives when it came time to integrate all our code.

Once the primitives were defined, the software design allowed for a specific cryptosystem, MAC and nonce generator to be packaged up into a single entity, referred to in the code as a "secure context". Secure contexts were the basic abstraction that was used throughout the handshake code in order to provide encryption and decryption services. The secure context was useful, because it could also handle generating/verifying nonces and MACs automatically.

Secure contexts for every algorithm supported by a client or server are all aggregated by another manager class. The main purpose of this class is to abstract away the cipher suite negotiation. Given a set of the cryptosystems supported by the other party, (which are actually cryptosystem-MAC-nonce triples), it chooses the cipher suite to use. Each cryptosystem has an innate priority value that is used to break any ties in cases where multiple combinations are supported.

The implementation is done entirely in Python 3, and uses the natice socket library for communication. Further, it uses Python's build-in secrets module for generating cryptographically secure random numbers.