Practical Cryptographic System

Final project - The Noise Protocol Framework Analysis

Introduction:

The main reason why we want to analyze this protocol is it is the protocol WhatsApp uses. However, few people know what this framework is, and there is no formal document that validate its security. In this report, we walk through details of this framework, and validate its security against some common threats, and compare it to TLS in different aspects.

Noise Framework:

Noise is a Framework instead of an actual protocol, you can use this framework to customize your own protocol. Noise protocols are Authenticated Key Exchange (or Agreement) protocols, their goals include generating shared symmetric key, providing authentication and forward secrecy.

Terminology

Just like TLS, Noise protocols begin with a **handshake phase.** During this phase, the protocols exchange **handshake messages** to reach a consensus on their symmetric key. Usually, the handshake phase involve two kinds of asymmetric key pairs, **ephemeral key pair** and **static key pair**. Both of them involve the process of deriving symmetric key. However, the former is used to provide randomness to the process, while the later makes linking digital credential with real-world entity possible.

One of the common term in Noise's language is **patterns**. People might see **message patterns** and **handshake patterns** appear over and over in the specification, and might be confused about what they actually means. In the previous discussion, we have talked about there are different ways to do a handshake in this framework. The **handshake pattern** is the one that specify which way to do the handshake. On the other hand, **message patterns** are different types of messages exchanged during a handshake. That is, a **handshake pattern** consists of a series of **message patterns**.

In Noise framework, different **handshake patterns** accommodate various contexts. Those specified in the documentation are all named with characters in order to indicate their initial status of **static key pair**. Specifically, if it is a **one-way pattern**, the handshake only involves one static key pair, thus, a single character is sufficient to represent the initial status. On the other hand, if it is an **interactive pattern,** it must consist of two characters to show the status of statice key pairs on both parties. The handshake patterns shown in the documentation are listed as follows.

One-way patterns:

```
N(rs):
<- s
...
-> e, es
```

```
K(s, rs):
-> s
<- s
...
-> e, es, ss
```

```
X(s, rs):
<- s
...
-> e, es, s, ss
```

Interactive patterns:

```
NN():
-> e
<- e, ee
```

```
NK(rs):
<- s
...
-> e, es
<- e, ee
```

```
NX(rs):
-> e
<- e, ee, s, es
```

```
KN(s):
-> s
...
-> e
<- e, ee, se
```

```
KK(s, rs):
-> s
<- s
...
-> e, es, ss
<- e, ee, se
```

```
KX(s, rs):
-> s
...
-> e
<- e, ee, se, s, es
```

```
XN(s):
-> e
<- e, ee
-> s, se
```

```
XK(s, rs):
<- s
...
-> e, es
<- e, ee
-> s, se
```

```
XX(s, rs):
-> e
<- e, ee, s, es
-> s, se
```

IN(s):	IK(s, rs):	IX(s, rs):
-> e, s	<- s	-> e, s
<- e, ee, se	...	<- e, ee, se, s, es
	-> e, es, s, ss	
	<- e, ee, se	

In the diagrams above, **message patterns** are specified by **tokens** and arrow. Tokens are a series of operations both initiator and responder should work on. Arrows designate the roles, writer or reader, while executing each token. For example, “-> e, es” means the initiator should operate token “e” and “es” as a writer, and the responder should operate token “e” and “es” as a reader. The operations corresponding to each role and tokens are shown as follows.

	writer	reader
"e"	Sets e (which must be empty) to GENERATE_KEYPAIR(). Appends e.public_key to the buffer. Calls MixHash(e.public_key).	Sets re (which must be empty) to the next DHLEN bytes from the message. Calls MixHash(re.public_key)
"s"	Appends EncryptAndHash(s.public_key) to the buffer.	Sets temp to the next DHLEN + 16 bytes of the message if HasKey() == True, or to the next DHLEN bytes otherwise. Sets rs (which must be empty) to DecryptAndHash(temp).
"ee"	Calls MixKey(DH(e, re)).	Calls MixKey(DH(e, re)).
"es"	Calls MixKey(DH(e, rs)) if initiator, MixKey(DH(s, re)) if responder.	Calls MixKey(DH(e, rs)) if initiator, MixKey(DH(s, re)) if responder.
"se"	Calls MixKey(DH(s, re)) if initiator, MixKey(DH(e, rs)) if responder.	Calls MixKey(DH(s, re)) if initiator, MixKey(DH(e, rs)) if responder.
"ss"	Calls MixKey(DH(s, rs)).	Calls MixKey(DH(s, rs)).

Functions and paramenters:

To simplify explanations in previous subsections, we did not include detail descriptions of all functions that are mentioned. It is assumed that people are able to understand what those functions do through their name. However, since some of their names are misleading, or may lead to ambiguity, we still describe some of them in the following paragraphs. But before we do that, we have to introduce the overall architecture of this framework.

In Noise, processing rules of handshake messages are defined in a object-oriented manner. Variables and functions are encapsulated into three hierarchical classes: **CipherState**, **SymmetricState**, and **HandshakeState**.

Variables in **CipherState** are used to encrypt and decrypt ciphertexts. For example, symmetric key “k” and counter-based nonce “n”. Usually, after the handshake phase, each party should have two **CipherState** objects. One for sending, and one for receiving.

A **SymmetricState** object is a combination of all "symmetric crypto" used by Noise. It is obvious that it should also include a **CipherState** object. Besides, it contains chaining key “ck” and handshake hash values “h”, which are used to derive symmetric keys. This object is usually deleted after the handshake is completed for security consideration.

Literally, a **HandshakeState** object contains everything that involve in a handshake. Not only **SymmetricState** object, but also DH variables such as local and remote ephemeral and static key pair “s”, “e”, “rs”, “re”, and a variable representing the handshake pattern are all included. Just like **SymmetricState** object, it should be deleted after the handshake.

Here are some functions that confuse people.

MixKey, MixHash, and MixKeyAndHash:

- What **MixKey** does is computing a DH share secret, that is, g^(ab). In XX pattern, this is the mechanism that ensures mutual authenticity. Since the secret key of static key pair is unknown to the adversary, token “es” or “se” are basically a CDH problem.
- **MixHash** is a function that updates handshake hash value “h” repeatedly.
- **MixKeyAndHash** is used for handling pre-shared symmetric keys. Since our discussion mainly focuses on XX pattern, we won’t get into any detail.

ENCRYPT, EncryptWithAd, and EncryptAndHash

- **ENCRYPT** is the function that used to encrypts plaintext. It uses the cipher key “k” of 32 bytes and an 8-byte unsigned integer nonce “n” which must be unique for the key “k”. Returns the ciphertext. Encryption must be done with an "AEAD" encryption mode with the associated data “ad”.
- **EncryptWithAd** is a member function of **CipherState** object. It returns ENCRYPT(k, n++, ad, plaintext) if “k” is non-empty. Otherwise it returns plaintext.
- **EncryptAndHash** is a member function of **SymmetricState** object. It sets ciphertext = **EncryptWithAd**(h, plaintext), calls **MixHash**(ciphertext), and returns ciphertext. Since it is calling **EncryptWithAd**, ciphertext might be plaintext if “k” is empty.

Initialize, InitializeKey, and InitializeSymmetric

- **Initialize** is the constructor of handshake state object. It is called when starting a handshake.
- **InitializeKey** is a member function of **CipherState** object, which sets k = key, n = 0.

- **InitializeSymmetric** is a member function of **SymmetricState** object. It initializes “h”, “ck”, then calls **InitializeKey**.

Comparison with TLS:

Encrypted handshake: The TLS handshake is in the clear, which leaks the parties' identities. Aside from ephemeral public keys, the Noise handshake is encrypted.

Low latency handshake: TLS specifies 2 round trips before clients can start sending data. Noise requires 1 round trip.

Forward secrecy: Even when TLS is used with ephemeral Diffie-Hellman, its forward secrecy is limited: compromise of an endpoint to an active connection will compromise previous traffic sent or received by that endpoint. In contrast, whenever the Noise ChaCha20/Poly1305 cipher suites send or receive ciphertext they destroy the keys for that ciphertext.

Simplicity: Noise is simpler than TLS. Noise omits many features that have resulted in TLS security flaws (version and ciphersuite negotiation, compression, renegotiation, chaining CBC IVs, MAC-then-encrypt, error alerts, etc.).

All Diffie-Hellman (no signatures): Using TLS with ephemeral ECDH requires signatures. Noise relies only on ECDH (no signatures). This yields a simpler and more robust protocol, reduces bandwidth, and avoids creating hard-to-deny evidence of who has communicated with who.

Security considerations:

Since XX pattern is the most widely used pattern. The discussion below will only focus on it. In the paper “A Framework for Universally Composable Diffie-Hellman Key Exchange” written by Ralf Ku ¨sters and Daniel Rausch, the author had already shown that as long as the exponents remain secret, the protocol is secure. Now, the question has become whether an adversary could learn the exponents used by senders and receivers, or could the adversary bypass the secure handshake process. We discuss several common security concerns as follows.

Man in the middle:

In this pattern, the mechanism used to prevent man in the middle attack is the same as all other normal PKIs. There are several options to link the electronic identity with the real entity in Noise framework. The most common way is to include the certificates in payload when sending handshake messages. Thus, what we know is this framework is at least as secure as the existing PKIs.

Side channel attacks:

Since whether a side channel attacks can work is dependent on its implementation. Sample codes written in different programming languages are provided on the official website. The one that we mainly focus on is python. In their code, most of the functions are done by a library pyca/cryptography. So far, there is no vulnerability found in its implementation. Therefore, we may consider the implementation to be secure.

Rollback attack

If parties decide on a Noise protocol based on some previous negotiation that is not included as prologue, then a rollback attack might be possible. This is a particular risk with fallback handshakes, and requires careful attention if a Noise handshake is preceded by communication between the parties.

Summary

In our analysis, Noise is just a framework that employ DH key exchange. Its security has been proved. The only thing we should concern about is the implementation. A slightly deviation of specification may lead to disastrous security issues such as making timing or power analysis possible, or forgot to delete the ephemeral parameters after the handshake. Nonetheless, they are just minor issues since flaws in implementation can always be fixed.

Appendix and reference:

AEAD encryption:
https://en.wikipedia.org/wiki/Authenticated_encryption

Noise Protocol:
https://noiseprotocol.org

A Framework for Universally Composable Diffie-Hellman Key Exchange:
https://eprint.iacr.org/2017/256.pdf

Slides from Stanford Security Seminar 2016
https://noiseprotocol.org/docs/noise_stanford_seminar_2016.pdf

Noise properties and protocol comparisons
https://github.com/noiseprotocol/noise_spec/wiki/Noise-properties-and-protocol-comparisons