PV204 Security technologies

LABS: Secure Channels

Petr Švenda svenda@fi.muni.cz
Faculty of Informatics, Masaryk University

CROCS
Centre for Research on
Cryptography and Security

TASK: BUILDING SECURE CHANNEL PROTOCOL

Task: Building Secure Channel protocol

- Scenario: we like to transfer extrasupersensitive data between PC and smartcard
- Simple protocol → design attack → fix it → iterate
  - Participate in discussion
- Hints for the solution are at the end of these slides, but read only after finishing the previous work

Place for protocol evolution drawing

Building SCP – steps in solution

- Scenario: we like to transfer extrasupersensitive data between PC and smartcard

1. Simple message exchanged in plaintext
2. Encrypted by static symmetric key
3. Integrity protection using plain hash
4. Integrity protection using MAC (CBC-MAC,HMAC)
5. Counter/Hash chain for message freshness and semantic security
6. Authenticated encryption (AEAD) modes of operation (GCM…)
7. Authentication based on static key
8. Challenge response for fresh authentication
9. Session keys derived from master key(s)
10. Forward secrecy based on RSA/ECDH
11. Backward secrecy based in Ratcheting (frequent ECDH)

TASK: PROTOCOL DISADVANTAGES

Group activity: methods for key establishment

- 3 people per group
- Write 1-3 disadvantages for each method
- Write into a mindmap with your group’s room
  - https://miro.com/app/board/o9J_lQ8-4dQ=/
  - (don’t cheat and don’t look at other mindmaps ;))
  - At the end, we will collate all results into a single one

1. Derive from pre-shared secret (KDF)
2. Establish with help of trusted party (Kerberos, PKI)
3. Establish over insecure channel (Diffie-Hellman)
4. Establish over other (secure, but very low-capacity/high-latency) channel
5. Establish over non-eavesdropable channel (BB84)

Collate together disadvantages

- Visit green highlighted mindmap at the bottom
- Start pasting your disadvantages (if not yet there)
- Start from the item corresponding to your room number (to avoid collisions), then move linearly forward
- See what we will get together!

TASK: ANALYZE GENERATED CODE FROM NOISE FRAMEWORK

Noise patterns

Important: this conscription contains both data send and actions executed → e == send public key, ee == perform ECDH (with eph. keys)

The first character refers to the initiator's static key:

- N = No static key for initiator
- K = Static key for initiator Known to responder
- X = Static key for initiator Xmitted ("transmitted") to responder
- I = Static key for initiator Immediately transmitted to responder, despite reduced or absent identity hiding

The second character refers to the responder's static key:

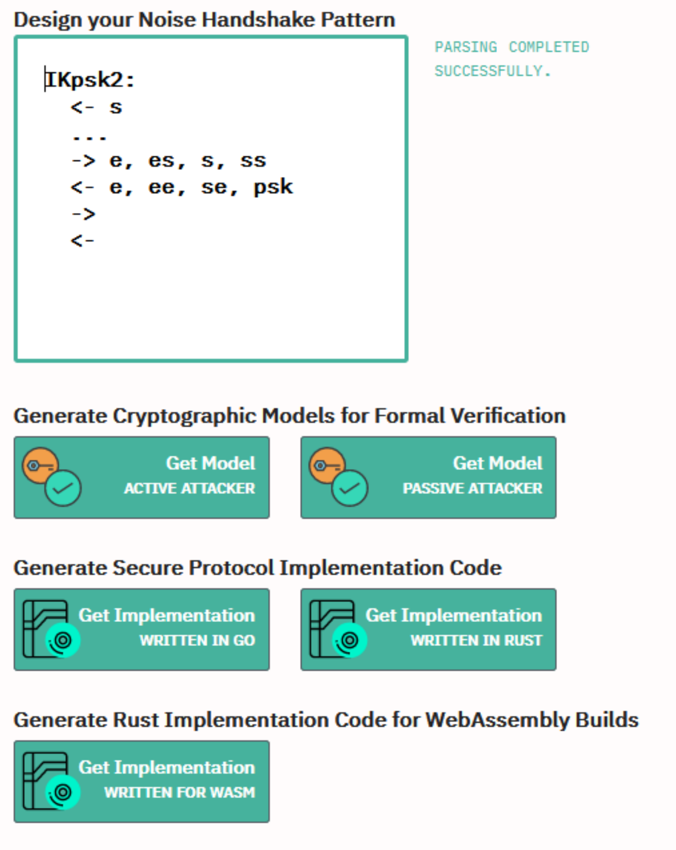
- N = No static key for responder
- K = Static key for responder Known to initiator
- X = Static key for responder Xmitted ("transmitted") to initiator

https://noiseexplorer.com/patterns/NN/
https://noiseexplorer.com/patterns/NX/

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

```
NX:
-> e
<- e, ee, s, es
```

https://noiseexplorer.com/patterns/IKpsk2/



- Noise protocols use set of rules:
  - If you already have some key, use it immediately to encrypt all subsequent messages
  - When new entropy is available (ECDH), update current state (keys) => forward&backward secrecy
  - Use AEAD for all message encryption
  - …

Task: Analyze code of Noise framework

- Group of three
- Visit https://noiseexplorer.com/, understand patterns naming convention, pattern modifiers
- Find required pattern
- Use any text diff to compare and see the difference in implementations
  - Pick GO implementations (easier to check by diff)
  - If you will pick Rust, the relevant file is state.rs (write_message_?() and read_message_?() functions)

```
type handshakestate struct {
        ss   symmetricstate      /*AEAD cipher state*/
        s    keypair             /*own long-term static ECDH share */
        e    keypair             /*own ephemeral ECDH share */
        rs   [32]byte            /*received long-term static ECDH share*/
        re   [32]byte            /*received ephemeral ECDH share*/
        psk  [32]byte            /*preshared symmetric key*/
}
```

```
/* -------------------------------- *
 * TYPES                            *
 * -------------------------------- */

type keypair struct {
        public_key  [32]byte
        private_key [32]byte
}

type messagebuffer struct {
        ne [32]byte // new ephm. share
        ns []byte
        ciphertext []byte
}

type cipherstate struct {
        k [32]byte  // key
        n uint32    // nonce
}
```

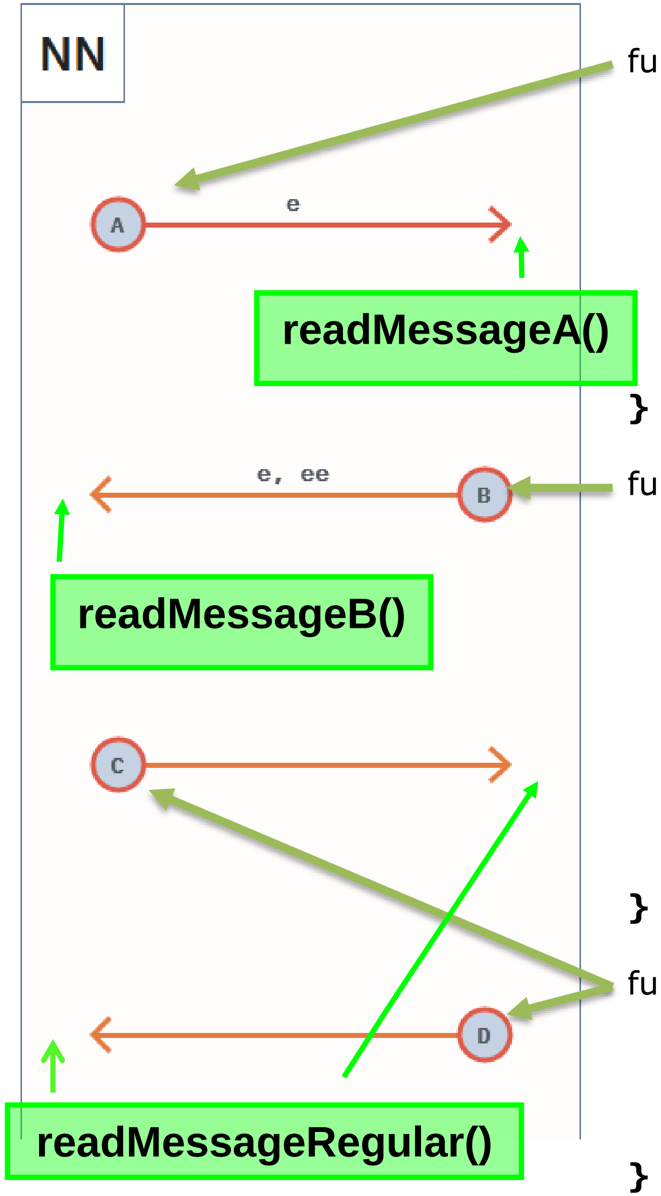
Important: not all items are used in all protocol patterns

```
type symmetricstate struct {
        cs cipherstate  // AEAD state (key and nonce)
        ck [32]byte     // chaining key
        h  [32]byte     // hash of handshake
}
type handshakestate struct {
        ss  symmetricstate
        s   keypair       // local static key pair
        e   keypair       // local ephemeral key pair
        rs  [32]byte      // remote party's static key
        re  [32]byte      // remote party's ephemeral key
        psk [32]byte      // pre-shared symmetric key
}
type noisesession struct {
        hs  handshakestate
        h   [32]byte      // handshake hash (unique for session)
        cs1 cipherstate   // cipherstate for the outgoing comm.
        cs2 cipherstate   // cipherstate for the incoming comm.
        mc  uint64        // incremental message counter
        i   bool          // True if this node is initiator
}
```

Important: single source file for both parties

- Initiator (A) and responder (B)
  - noisesession.i **bool** // True if this node is initiator
- Not all functions will be used by both parties
- When executed, you need to specify who is initiator
  - Initiator (A) will use writeMessageA, readMessageB…
  - Responder (B) will use readMessageA, writeMessageB…

- *.go; *.rs – source code of implementation
- *.pv – script for ProVerif formal verification (see annotated claims at specific message of given pattern)



Generate ephemeral ECDH keypair

Read own eph. ECDH public key

Hash it into key state

AEAD of payload (optional)

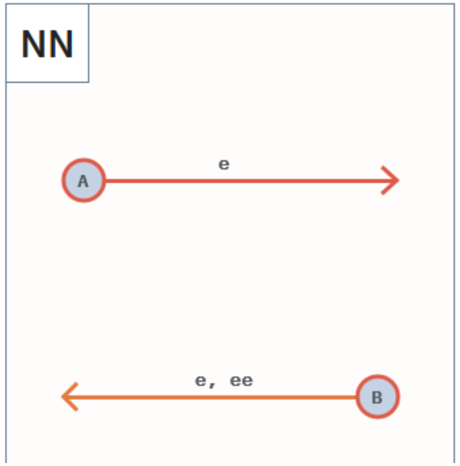
Format whole message

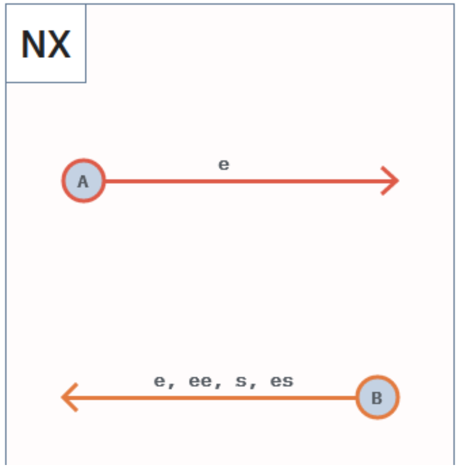
```
func writeMessageA(hs *handshakestate, payload []byte) (*handshakestate, messagebuffer) {
        ne, ns, ciphertext := emptyKey, []byte{}, []byte{}
        hs.e = generateKeypair()
        ne = hs.e.public_key
        mixHash(&hs.ss, ne[:])
        /* No PSK, so skipping mixKey */
        _, ciphertext = encryptAndHash(&hs.ss, payload)
        messageBuffer := messagebuffer{ne, ns, ciphertext}
        return hs, messageBuffer
}

func writeMessageB(hs *handshakestate, payload []byte) ([32]byte, messagebuffer, cipherstate, cipherstate) {
        ne, ns, ciphertext := emptyKey, []byte{}, []byte{}
        hs.e = generateKeypair()
        ne = hs.e.public_key
        mixHash(&hs.ss, ne[:])
        /* No PSK, so skipping mixKey */
        mixKey(&hs.ss, dh(hs.e.private_key, hs.re))
        _, ciphertext = encryptAndHash(&hs.ss, payload)
        messageBuffer := messagebuffer{ne, ns, ciphertext}
        cs1, cs2 := split(&hs.ss)
        return hs.ss.h, messageBuffer, cs1, cs2
}

func writeMessageRegular(cs *cipherstate, payload []byte) (*ci
        ne, ns, ciphertext := emptyKey, []byte{}, []byte{}
        cs, ciphertext = encryptWithAd(cs, []byte{}, payload)
        messageBuffer := messagebuffer{ne, ns, ciphertext}
        return cs, messageBuffer
}
```

Similarly, readMessageA(), readMessageB, readMessageRegular() methods are used to process received inputs from writeMessageA()…

Important: writeMessage() takes also optional arbitrary payload atop of key exchange data. Is encrypted by AEAD if needed

NN vs. NX protocol pattern





The first character refers to the initiator's static key:

- N = No static key for initiator
- K = Static key for initiator Known to responder
- X = Static key for initiator Xmitted ("transmitted") to responder
- I = Static key for initiator Immediately transmitted to responder, despite reduced or absent identity hiding

The second character refers to the responder's static key:

- N = No static key for responder
- K = Static key for responder Known to initiator
- X = Static key for responder Xmitted ("transmitted") to initiator

9.4. Pattern modifiers

To indicate PSK mode and the placement of the "psk" token, pattern modifiers are used (see Section 8). The modifier psk0 places a "psk" token at the beginning of the first handshake message. The modifiers psk1, psk2, etc., place a "psk" token at the end of the first, second, etc., handshake message.

```
func writeMessageB(hs *handshakestate, payload []byte) ([32]byte, :
        ne, ns, ciphertext := emptyKey, []byte{}, []byte{}
        hs.e = generateKeypair()
        ne = hs.e.public_key
        mixHash(&hs.ss, ne[:])
        /* No PSK, so skipping mixKey */
        mixKey(&hs.ss, dh(hs.e.private_key, hs.re))
        _, ciphertext = encryptAndHash(&hs.ss, payload)
        messageBuffer := messagebuffer{ne, ns, ciphertext}
        cs1, cs2 := split(&hs.ss)
        return hs.ss.h, messageBuffer, cs1, cs2
}
```

```
func writeMessageB(hs *handshakestate, payload []byte) ([32]byt
        ne, ns, ciphertext := emptyKey, []byte{}, []byte{}
        hs.e = generateKeypair()
        ne = hs.e.public_key
        mixHash(&hs.ss, ne[:])
        /* No PSK, so skipping mixKey */
        mixKey(&hs.ss, dh(hs.e.private_key, hs.re))
        spk := make([]byte, len(hs.s.public_key))
        copy(spk[:], hs.s.public_key[:])
        _, ns = encryptAndHash(&hs.ss, spk)
        mixKey(&hs.ss, dh(hs.s.private_key, hs.re))
        _, ciphertext = encryptAndHash(&hs.ss, payload)
        messageBuffer := messagebuffer{ne, ns, ciphertext}
        cs1, cs2 := split(&hs.ss)
        return hs.ss.h, messageBuffer, cs1, cs2
}
```

Protocols to analyze

1. Find pattern corresponding to non-authenticated ephemeral ECDH from both sides
2. Find pattern, where both parties share long-term ECDH share and update with fresh ephemeral one
3. Find pattern where responder has long-term static ECDH share, pre-shared with initiator
   - Corresponding to 0-RTT of data send from client to server with pre-shared static share of server's key

- For every protocol: Find parameters chosen for implementation of a protocol
  - What hash and cipher algorithms were used?
  - What elliptic curve is used?
- For every protocol: look at functions writeMessageA, writeMessageB…
  - What is hashed/mixed into shared state?
  - What is encrypted (AEAD) before send?
- How can you utilize pre-shared password if exists? (read https://noiseprotocol.org/noise.pdf)

NO HOMEWORK ASSIGNMENT THIS WEEK ☺

CHECK-OUT

Checkout

- Which of the seminar parts you enjoyed most?
- Rank it according the level of enjoyment (most enjoyable => first)
- Write to sli.do when displayed

PV204_02 Rank the topics covered today based on the level of enjoyment

ⓘ Start presenting to display the poll results on this slide.

THANK YOU FOR COMING, SEE YOU NEXT WEEK

SOLUTIONS – KIND OF ☺
READ ONLY AFTER THE SEMINAR DISCUSSION

READ ONLY AFTER THE SEMINAR DISCUSSION!

Building SCP – steps in solution

- Scenario: we like to transfer extrasupersensitive data between PC and smartcard

1. Simple exchange in plaintext
2. Encrypted by static symmetric key
3. Integrity protection using plain hash
4. Integrity protection using MAC (CBC-MAC,HMAC)
5. Counter/Hash chain for message freshness and semantic security
6. Authenticated encryption (AEAD) modes of operation (GCM…)
7. Authentication based on static key
8. Challenge response for fresh authentication
9. Session keys derived from master key(s)
10. Forward secrecy based on RSA/DH
11. Backward secrecy based in Ratcheting (frequent ECDH)

Building SCP – steps in solution

1. Simple exchange in plaintext
   - Many problems, attacker can eavesdrop sensitive data
2. Encrypted by static symmetric key
   - Attacker can modify sensitive data (no integrity)
3. Integrity protection using plain hash
   - Hash is not enough, attacker can modify then recompute hash
4. Integrity protection using MAC (CBC-MAC,HMAC)
   - Attacker can replay older message (no freshness)

Building SCP – steps in solution

5. Counter/hash chain for message freshness and semantic security
   - No explicit authentication of parties
6. Authenticated encryption (AEAD) modes
   - Secure composition of ENC and MAC. Currently GCM, but soon to finish CAESAR competition with
7. Authentication based on static key
   - Authentication message can be replayed from previous legit run
8. Challenge response for fresh authentication
   - Single static key can cause problems
     - Interchange of encrypted message and valid MAC
     - Large amount of data encrypted under same key (cryptoanalysis)

Building SCP – steps in solution

9. Session keys derived from master key(s)
   - If master keys are compromised, older captured communication can be decrypted
10. Forward secrecy based on RSA/DH
    - Future messages can read after compromise
    - Key has to be kept for a long time for out-of-order messages
11. Backward secrecy based on ratcheting
    - Secure?
    - Key management with multiple parties?
    - Proof of message origin? Deniability?
    - … gather your requirements!