Nym Technologies
June 2022

Sphinx RFC Documentation

Contents

1. Introduction

Sphinx is a cryptographic packet format introduced by Danezis and Goldberg [Add Ref] used to rely packets via multi-hop paths in decentralized networks.

1.1. Terminology

- Message - A variable-length sequence of bytes transmitted through a decentralized network.
- Packet - A fixed-length sequence of bytes transmitted through a decentralized network, containing the encrypted message and routing instructions.
- Header - A fixed-length part of a packet consisting of several components, which convey the information necessary to verify packet integrity and correctly process the packet.
- Payload - A fixed-length part of a packet containing an encrypted message.
- Group - A finite set of elements and a binary operation that satisfy the properties of closure, associativity, invertability, and the presence of an identity element.
- Group element - An individual element of a group.
- Group Order - The number of elements present in the group.
- Group generator - A group element capable of generating any other element of the group, via repeated applications of the generator and the group operation.
- Cyclic Group - A group generated by a single element.
- Tag - A cryptographic checksum on data that detects accidental or intentional modifications of the data.

1.2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [Add Ref].

1.3. Notation

- x || y denotes the concatenation of x and y,
- x ^ y denotes the bitwise XOR of x and y,
- v[a:b] denotes the sub-vector of v where a/b denote the start/end byte indexes (inclusive-exclusive),
- v[a:] denotes a sub-vector of v from index a (inclusive) till the last index (inclusive)
- v[:a] denotes a sub-vector of v from the first index (inclusive) till index a (exclusive)
- v[i] denotes the i’th element of list v,
- v.len denotes the length of vector v,
- ZEROBYTES(N) denotes N bytes of 0x00,
- RANDBYTES(N) denotes a sequence of N bytes selected at random,

2. Cryptographic Primitives

- KDF(SALT, IKM) - A secure key derivation function which takes an arbitrary length octet array salt SALT and an arbitrary length octet array initial key IKM, to produce an octet array of arbitrary length
- EXP_KEYGEN() - return a random group element of GROUP_ELEMENT_LENGTH bytes
- PRNG(K, IV) - A pseudo-random generator (stream cipher) which takes a STREAM_CIPHER_KEY_SIZE byte octet array key K and a STREAM_CIPHER_KEY_SIZE byte octet array initialization vector IV to produce an octet array key stream of arbitrary length
- WBC(K, MSG) - A wide-block cipher which takes a PAYLOAD_KEY_SIZE byte octet array key K and a byte array MSC and returns an encrypted byte array of length PAYLOAD_SIZE.
- SEncrypt(K, M)/SDecrypt(K, M) - A stream cipher which takes STREAM_CIPHER_KEY_SIZE byte octet array key K and a byte array message M, and produces the encrypted ciphertext or decrypted plaintext respectively.

3. Sphinx Packet Parameters & Geometry

3.1. Parameters

- SECURITY_PARAMETER - the constant length of a security parameter in bytes.
- AD_LENGTH - the constant amount of per-packet unencrypted additional data in bytes.
- MAX_PATH_LENGTH - the maximum number of nodes that a Sphinx mix message will traverse before being delivered to its destination.
- BLINDING_FACTOR_SIZE -
- GROUP_ELEMENT_SIZE - the size in bytes of a single group element.
- STREAM_CIPHER_KEY_SIZE - the length in bytes of the key used for the stream cipher.
- STREAM_CIPHER_OUTPUT_LENGTH - the length in bytes of the output of a stream cipher
- INTEGRITY_MAC_KEY_SIZE - the length in bytes of the key used for the HMAC.
- PAYLOAD_KEY_SIZE - the length in bytes of the key used to encrypt/decrypt the payload.
- ROUTING_KEYS_LENGTH - the length in bytes of the routing keys per node.
- DESTINATION_ADDRESS_LENGTH - the length in bytes of the destination address.
- NODE_ADDRESS_LENGTH - the length in bytes of the node address.
- HEADER_INTEGRITY_MAC_SIZE - the length in bytes of the integrity tag included in the header.
- NODE_META_INFO_SIZE - the length of the meta info containing all the information from sender to the relay node in bytes.
- FINAL_NODE_META_INFO_LENGTH - the length of the meta info for the last relay node.
- PAYLOAD_SIZE - the length in bytes of the Sphinx payload.

3.2. Geometry

- ROUTING_KEYS_LENGTH - The total length of the routing information in bytes

  ROUTING_KEYS_LENGTH = STREAM_CIPHER_KEY_SIZE + INTEGRITY_MAC_KEY_SIZE + PAYLOAD_KEY_SIZE + BLINDING_FACTOR_SIZE

- HEADER_SIZE - The total size of the Sphinx header in bytes

  HEADER_SIZE = GROUP_ELEMENT_SIZE + HEADER_INTEGRITY_MAC_SIZE + ENCRYPTED_ROUTING_INFO_SIZE

- PACKET_LENGTH - The length of the Sphinx Packet in bytes

  PACKET_LENGTH = HEADER_SIZE + PAYLOAD_SIZE

- ENCRYPTED_ROUTING_INFO_SIZE

  ENCRYPTED_ROUTING_INFO_SIZE = (NODE_META_INFO_SIZE + HEADER_INTEGRITY_MAC_SIZE) * MAX_PATH_LENGTH;

- FILLER_STEP_SIZE_INCREASE - the length by which we increase the filler padding for each hop

  FILLER_STEP_SIZE_INCREASE = NODE_META_INFO_SIZE + HEADER_INTEGRITY_MAC_SIZE

- FINALPADDING - the length in bytes of a padding added in the innermost layer of Sphinx header encapsulation

  FINALPADDING = ENCRYPTED_ROUTING_INFO_SIZE - (FILLER_STEP_SIZE_INCREASE * (route_len - 1)) - FINAL_NODE_META_INFO_LENGTH,

  were route_len is a length of the route selected for the given Sphinx packet

4. Sphinx Packet Structure

A Sphinx Packet is composed of two segments: a header, containing routing instructions for the intermediate nodes, and a payload hiding the message content. The byte length of the Sphinx header is specified by parameter **HEADER_LENGTH**, while the byte length of the Sphinx payload is defined by the parameter **PAYLOAD_LENGTH**. The total size of a Sphinx packet is defined by **PACKET_LENGTH**.

```
struct SphinxPacket {
    header : [u8; HEADER_SIZE],
    payload: [u8; PAYLOAD_SIZE],
}
```

The header of a Sphinx packet contains routing instructions for the intermediate nodes and information necessary to verify packet integrity. The payload contains encrypted message content.

4.1. Sphinx Header Overview

The header of a Sphinx packet comprises of the following parts: an element of a cyclic group of prime order, encrypted routing information and an integrity authentication tag covering the encrypted routing information. A plaintext vector with additional information e.g., software version is attached at the beginning of the header.

```
struct SphinxHeader {
    additional_data: [u8; AD_LENGTH],
    group_element: [u8; GROUP_ELEMENT_SIZE],
    encrypted_routing_information [u8; ENCRYPTED_ROUTING_INFO_SIZE],
    integrity_tag: [u8; HEADER_INTEGRITY_MAC_SIZE],
}
```

The group element is used by each mix in the packet route to derive a **secret** that is shared with the original sender of the packet. A secure key derivation function **KDF** with input **secret** is used to further extract an encryption key **header_enc_key**, integrity key **integrity_key**, **blinding_factor** and payload key **payload_key**. Value **blinding_factor** is used by a relay node to blind the **group_element** to prevent packet linking while it traverses the route. The integrity key **integrity_key** is used to derive a hash-based message authentication code using **enc_routing_information** which is next compared against **integrity_tag** to ensure no part of the header containing routing information has been modified. The key **header_enc_key** is used to remove layer of symmetric encryption from **enc_routing_information** to extract the new routing instruction **enc_routing_information** and new **integrity_tag** for the next relay. The payload key **payload_key** is used to remove a layer of encryption from the payload part.

5. Sphinx Packet Creation

To create a Sphinx packet, the sender specifies the message **msg**, designated destination **dest** and a sequence of relays **route** through which the packet should be routed. The sender generates an ephemeral private key **emph_key** and using the Diffie-Hellman protocol combines it with the public keys of the relays to derive a vector of all session secrets. The secrets are next used to encode the routing instructions by wrapping it in multiple layers of encryption using secure stream cipher **SCPH** and calculating the correct message authentication codes using **HMAC** for each stage of the journey. The secrets are also used to layer encrypt the payload part of the packet. The payload, containing the message, is kept separate from the header and encrypted using a wide-block cipher **WBC**.

```
fn create_packet(msg, route, dest) -> SphinxPacket {

    let emph_key = EXP_KEYGEN();
    let mut routing_keys = Vec::with_capacity(route.len());

    let mut accumulator = emph_key.clone();
    for (i, node) in route.iter().enumerate() {
        let shared_key = EXP( node.public_key, emph_key);
        let node_routing_keys = KDF(shared_key);
        if i != route.len() + 1 {
            accumulator *= blinding_factor_scalar;
        }
        routing_keys.push(node_routing_keys);
    }

    let payload_keys = key_material
```

```
                .routing_keys
                .iter()
                .map(|routing_key| routing_key.payload_key)
                .collect();

        let header = create_header(routing_keys, route, dest);
        let payload = encapsulate_message(msg, dest, payload_keys);
        Ok(SphinxPacket { header, payload })
    }
```

5.1. Sphinx Header Creation

Creating a Sphinx header requires first creating the filler_string, an encrypted padding sequence for each hop. This filler ensures that the total byte size of a Sphinx packet is the same at each hop within the network.

```
    fn create_filler(routing_keys){
        let filler_value = routing_keys
                .iter()
                .map(|node_routing_keys| node_routing_keys.stream_cipher_key)
                .map(|cipher_key| {
                    PRNG(&PRNG_KEY, &PRNG_IV)[:STREAM_CIPHER_OUTPUT_LENGTH]
                })
                .enumerate()
                .map(|(i, pseudorandom_bytes)| (i + 1, pseudorandom_bytes))
                .fold(
                    Vec::new(),
                    |filler_string_accumulator, (i, pseudorandom_bytes)| {
                        filler_step(filler_string_accumulator, i, pseudorandom_bytes)
                    },
                );
    }

    fn filler_step(filler_string_accumulator, i, pseudorandom_bytes) -> Vec<u8> {
        assert_eq!(
            pseudorandom_bytes.len(),
            STREAM_CIPHER_OUTPUT_LENGTH
        );
        assert_eq!(
            filler_string_accumulator.len(),
            FILLER_STEP_SIZE_INCREASE * (i - 1)
        );

        let zero_bytes = vec![0u8; FILLER_STEP_SIZE_INCREASE];
        filler_string_accumulator.extend(&zero_bytes);


        xor_with(
            &mut filler_string_accumulator,
            &pseudorandom_bytes[pseudorandom_bytes.len() - i * FILLER_STEP_SIZE_INCREASE..],
        );

        filler_string_accumulator
    }
```

Next, the routing information for all the nodes is encapsulated. The routing information designated for each intermediate node is first encrypted and next combined with a HMAC for integrity protection. The innermost layer is constructed as a concatenation of the destination address **dest**, an identifier **I** used for SURBs, and a sequence of padding, which ensures the path length is not accidentally revealed. These are then encrypted by XORing with the output of a pseudo-random number generator seeded with shared key. The result is merged with a filler string, the addition of which ensures the header packets remain constant in size as layers of encryption are added or removed. Once the final encrypted routing information is ready, an integrity tag is computed using HMAC, and the routing information is encrypted using a stream cipher, and concatenated with the integrity tag. The process is recursively repeated for each of the nodes in the route.

```
fn encapsulate_routing_information(route, dest, I, node_routing_info, routing_keys, filler) {

    assert_eq!(route.len(), routing_keys.len());

    let dest_info = dest || I || RANDBYTES(FINALPADDING);
    let enc_dest_info = SENCRYPT( routing_keys.stream_cipher_key, dest_info);
    let destination_routing_info = enc_dest_info || HMAC(routing_keys.last().integrity_key);

    route.iter()
        .skip(1)
        .map(|node| node.address)
        .zip(
            routing_keys.iter().take(routing_keys.len() - 1)
        )
        .zip(node_routing_info.iter().take(node_routing_info.len() - 1))
        .rev()
        .fold(destination_routing_info, | mut next_hop_routing_information,
             ((current_node_address, previous_node_routing_keys), node_routing_info)|
             {
                let encrypted_routing_info =
                SENCRYPT(previous_node_routing_keys.stream_cipher_key,
                    current_node_address || node_routing_info
                    || next_hop_encapsulated_routing_information);

                let integrity_tag =
                    HMAC(previous_node_routing_keys.integrity_key, encrypted_routing_info);

                next_hop_routing_information = encrypted_routing_info || integrity_tag
            },
        )
}
```

5.2. Sphinx Payload Creation

The Sphinx payload is computed separately from the header, using a wide-block cipher. Just as in the header, the payload encapsulation is performed in reverse order of the route using the payload keys computed during the initial key derivation. The inner most layer of the payload is a concatenation of the message msg, destination address dest, and a sequence of zero-byte padding of length SECURITY_PARAMETER, which allows verification that the body was not modified in transit.

```
fn encapsulate_message(msg, dest, payload_keys) {
    let payload = msg || dest || ZEROBYTES(SECURITY_PARAMETER);
    for payload_key in payload_keys.iter().rev() {
        payload = WBC(payload_key, payload);
    }
}
```

The compute Sphinx header and Sphinx payload are then concatenated into a single Sphinx packet.

6. Sphinx Packet Processing

In order to process a SphinxPacket we first process the header part, and next the payload part. Upon receiving a packet, the relay node extracts the group element from the header and combines it with its own secret key in order to derive the shared key. Next, the KDF is used to derive all routing keys from the shared secret.

```
fn process_packet(packet, node_secret_key) -> SphinxPacket {
    let shared_key = EXP(packet.shared_secret, node_secret_key);
    let routing_keys = KDF(shared_key);

    let new_header = process_header(packet.header, routing_keys);
    let new_payload = process_payload(packet.payload, routing_keys.payload_keys);

    SphinxPacket{
        new_header,
        new_payload,
    }
}
```

6.1. Sphinx Header Processing

Using the integrity key and obtained encrypted routing information, the relay node computes and HMAC and compares it against the integrity tag encoded in the packet header. If the verification fails, the node MUST abort. Otherwise, the relay node first appends a zero byte padding at the end of the encrypted routing information and decrypts the padded block by XORing it with a pseudo-random stream of bytes generated using the shared secret. Finally, the relay node blinds the received group element using a blinding factor derived from the shared key to obtain a fresh group element.

```
fn process_header(header, routing_keys) -> SphinxHeader{

    if !(HMAC(routing_keys.integrity_key, header.encrypted_routing_information)
            == header.integrity_tag) {
        return False;
    }

    let padded_encrypted_routing_information =
        encrypted_routing_information ||
        ZEROBYTES(NODE_META_INFO_SIZE + HEADER_INTEGRITY_MAC_SIZE)

    let pseudorandom_bytes = PRNG(shared_key, STREAM_CIPHER_INIT_VECTOR);
    let new_encrypted_routing_information, new_integrity_tag =
    XOR(padded_encrypted_routing_information, pseudorandom_bytes)

    let new_group_element = EXP(header.group_element, routing_keys.blinding_factor);

    SphinxHeader{
        header.additional_data,
        new_group_element,
        new_encrypted_routing_information,
        new_integrity_tag,
    }
}
```

6.2. Sphinx Payload Processing

The relay node also performs wide-block cipher decryption to unwrap a single encryption layer from the payload.

```
fn process_payload(payload, payload_key) -> SphinxPayload{

    let unwrapped_payload = WBC(payload_key, payload);
    SphinxPayload{
        unwrapped_payload
    }
}
```

7. Single Use-Reply Blocks

A Single Use Reply Block (SURB) is a data structure derived by the originator of a Sphinx packet in order to facilitate anonymous replies. it can be used by the recipient to reply to the initial sender.
When a SURB is created, a matching reply block decryption data is created, which is used to decrypt the reply message that is produced and delivered via the SURB. The creation of a SURB is similar to the creation of a SphinxHeader.

```
struct SURB {
    header: [u8; HEADER_SIZE],
    first_hop: [u8; NODE_ADDRESS_LENGTH],
    payload_keys: Vec<[u8; PAYLOAD_KEY_SIZE]>
}
```