ethereum / **devp2p**

Branch: **master** devp2p / **rlpx.md** Find file Copy path

romanman Update rlpx.md 5ae90ef on Oct 8, 2015

4 contributors

```
437 lines (338 sloc)   23.5 KB
```

```
RLPx: Cryptographic Network & Transport Protocol
Alex Leverington
Version 0.3
```

Introduction

RLPx is a cryptographic peer-to-peer network and protocol suite which provides a general-purpose transport and interface for applications to communicate via a p2p network. RLPx is designed to meet the requirements of decentralized applications and is used by Ethereum.

The current version of RLPx provides a network layer for Ethereum. Roadmap:

- Completed:
  - UDP Node Discovery for single protocol
  - ECDSA Signed UDP
  - Encrypted Handshake/Authentication
  - Peer Persistence

  - Encrypted/Authenticated TCP
  - TCP Framing
- May '15: Beta
  - Node Discovery for single protocol
  - Transport is feature complete
  - Encrypted UDP
- July '15: Beta
  - Revisit node table algorithm
  - Discovery support for multiple protocols
- Winter '15: 1.0
  - Node Discovery for multiple protocols
  - Feature complete UDP

Features

- Node Discovery and Network Formation
- Encrypted handshake
- Encrypted transport
- Protocol Mux (framing)
- Flow Control
- Peer Preference Strategies
- Peer Reputation
- Security
  - authenticated connectivity (ECDH+ECDHE, AES128)
  - authenticated discovery protocol (ECDSA)
  - encrypted transport (AES256)
  - protocols sharing a connection are provided uniform bandwidth (framing)
  - nodes have access to a uniform network topology
  - peers can uniformly connect to network
  - localised peer reputation model

Transport

Objectives

- multiple protocols over single connection
- encrypted
- flow control

Authenticated encryption is employed in order to provide confidentiality and protect against network disruption. This is particularly important for a well-formed network where nodes make long-term decisions about other nodes which yield non-local effects.

Dynamic framing and flow control ensure that each protocol is allotted the same amount of bandwidth.

Network Formation

Objectives

- new nodes can reliably find nodes to connect to
- nodes have sufficient network topology information to uniformly connect to other nodes
- node identifiers are random

RLPx utilizes Kademlia-like routing which has been repurposed as a p2p neighbour discovery protocol. RLPx discovery uses 512-bit public keys as node ids and sha3(node-id) for xor metric. DHT features are not implemented.

Implementation Overview

Packets are dynamically framed, prefixed with an RLP encoded header, encrypted, and authenticated. Multiplexing is achieved via the frame header which specifies the destination protocol of a packet.

All cryptographic operations are based on secp256k1 and each node is expected to maintain a static private key which is saved and restored between sessions. It is recommended that the private key can only be reset manually, for example, by deleting a file or database entry.

An RLPx implementation is composed of:

- Node Discovery
- Encrypted Transport
- Framing
- Flow Control

Node Discovery

Node: An entity on the network.
Peer: Node which is currently connected to host node.
NodeId: public key of node

Node discovery and network formation are implemented via a kademlia-like UDP. Major differences from Kademlia:

- packets are signed
- node ids are public keys
- DHT-related features are excluded. FIND_VALUE and STORE packets are not implemented.
- xor distance metric is based on sha3(nodeid)

The parameters chosen for kademlia are a bucket size of 16 (denoted k in Kademlia), concurrency of 3 (denoted alpha in Kademlia), and 8 bits per hop (denoted b in Kademlia) for routing. The eviction check interval is 75 milliseconds, request timeouts are 300ms, and the idle bucket-refresh interval is 3600 seconds. Until encryption is implemented packets have a timestamp property to reduce the window of time for carrying out replay attacks. How the timestamp is handled is up to the receiver and it's recommended that the receiver only accept packets created within the last 3 seconds; timestamps can be ignored for Pong packets. In order to reduce the chance that packets are fragmented the maximum size of a datagram is 1280 bytes, as this is the minimum size of an IPv6 datagram.

Except for the previously described differences, node discovery employs the system and protocol described by Maymounkov and Mazieres.

Packets are signed. Verification is performed by recovering the public key from the signature and checking that it matches an expected value. Packet properties are serialized as RLP in the order in which they're defined.

RLPx provides a list of 'potential' nodes, based on distance metrics, and can maintain connections for well-formedness based on an ideal peer count (default is 5). This strategy is implemented by connecting to 1 random node for every 'close' node which is connected.

Other connection strategies which can be manually implemented by a protocol; a protocol can use it's own metadata and strategies for making connectivity decisions.

Note: It is intended that the final version of this protocol be framed and encrypted using the same or similar structures as-is used for TCP packets. This change is expected following the first working implemention of the node discovery protocol, framing, and encryption.

Packet Encapsulation:

```
hash || signature || packet-type || packet-data
        hash: sha3(signature || packet-type || packet-data)     // used to verify integrity of
datagram
        signature: sign(privkey, sha3(packet-type || packet-data))
        signature: sign(privkey, sha3(pubkey || packet-type || packet-data)) // implementation w/MCD
        packet-type: single byte < 2**7 // valid values are [1,4]
        packet-data: RLP encoded list. Packet properties are serialized in the order in which they're
defined. See packet-data below.
```

DRAFT Encrypted Packet Encapsulation:

```
mac || header || frame
        header: frame-size || header-data
    frame-size: 3-byte integer size of frame, big endian encoded
    header-data:
        normal: rlp.list(protocol-type[, context-id])
        chunked-0: rlp.list(protocol-type, context-id, total-packet-size)
                chunked-n: rlp.list(protocol-type, context-id)
        values:
            protocol-type: < 2**16
            context-id: < 2**16 (this value is optional for normal frames)
            total-packet-size: < 2**32
```

Packet Data (packet-data):

```
All data structures are RLP encoded.
Total payload of packet (excluding IP headers) must be no greater than 1280 bytes.
NodeId: The node's public key.
inline: Properties are appened to current list instead of encoded as list.
Maximum byte size of packet is noted for reference.
timestamp: When packet was created (number of seconds since epoch).

PingNode packet-type: 0x01
struct PingNode
{
        h256 version = 0x3;
        Endpoint from;
        Endpoint to;
        uint32_t timestamp;
};

Pong packet-type: 0x02
struct Pong
{
        Endpoint to;
        h256 echo;
        uint32_t timestamp;
};

FindNeighbours packet-type: 0x03
struct FindNeighbours
{
        NodeId target; // Id of a node. The responding node will send back nodes closest to the
target.
        uint32_t timestamp;
};

Neighbors packet-type: 0x04
struct Neighbours
{
        list nodes: struct Neighbour
        {
                inline Endpoint endpoint;
                NodeId node;
        };

        uint32_t timestamp;
};

struct Endpoint
{
        bytes address; // BE encoded 4-byte or 16-byte address (size determines ipv4 vs ipv6)
        uint16_t udpPort; // BE encoded 16-bit unsigned
        uint16_t tcpPort; // BE encoded 16-bit unsigned
}
```

Encrypted Handshake

Connections are established via a handshake and, once established, packets are encapsulated as frames which are encrypted using AES-256 in CTR mode. Key material for the session is derived via a KDF with ECDHE-derived keying material. ECC uses secp256k1 curve (ECP). Note: "remote" is host which receives the connection.

The handshake is carried out in two phases. The first phase is key exchange and the second phase is authentication and protocol negotiation. The key exchange is an ECIES encrypted message which includes ephemeral keys for Perfect Forward Secrecy (PFS). The second phase of the handshake is a part of DEVp2p and is an exchange of capabilities that each node supports. It's up to the implementation how to handle the outcome of the second phase of the handshake.

There are several variants of ECIES, of which, some modes are malleable and must not be used. This specification relies on the implementation of ECIES as defined by Shoup. Thus, decryption will not occur if message authentication fails. http://en.wikipedia.org/wiki/Integrated_Encryption_Scheme

There are two kinds of connections which can be established. A node can connect to a known peer, or a node can connect to a new peer. A known peer is one which has previously been connected to and from which a corresponding session token is available for authenticating the requested connection.

If the handshake fails upon initiating a connection TO a known peer, then the nodes information should be removed from the node table and the connection MUST NOT be reattempted. Due to the limited IPv4 space and common ISP practices, this is likely a common and normal occurrence, therefore, no other action should occur. If a handshake fails for a connection which is received, no action pertaining to the node table should occur.

Handshake:

```
New: authInitiator -> E(remote-pubk, S(ephemeral-privk, static-shared-secret ^ nonce) || H(ephemeral-pubk) || pubk || nonce || 0x0)
     authRecipient -> E(remote-pubk, remote-ephemeral-pubk || nonce || 0x0)

Known: authInitiator = E(remote-pubk, S(ephemeral-privk, token ^ nonce) || H(ephemeral-pubk) || pubk || nonce || 0x1)
       authRecipient = E(remote-pubk, remote-ephemeral-pubk || nonce || 0x1) // token found
       authRecipient = E(remote-pubk, remote-ephemeral-pubk || nonce || 0x0) // token not found

static-shared-secret = ecdh.agree(privkey, remote-pubk)
ephemeral-shared-secret = ecdh.agree(ephemeral-privk, remote-ephemeral-pubk)
```

Values generated following the handshake (see below for steps):

```
ephemeral-shared-secret = ecdh.agree(ephemeral-privkey, remote-ephemeral-pubk)
shared-secret = sha3(ephemeral-shared-secret || sha3(nonce || initiator-nonce))
token = sha3(shared-secret)
aes-secret = sha3(ephemeral-shared-secret || shared-secret)
# destroy shared-secret
mac-secret = sha3(ephemeral-shared-secret || aes-secret)
# destroy ephemeral-shared-secret

Initiator:
egress-mac = sha3.update(mac-secret ^ recipient-nonce || auth-sent-init)
# destroy nonce
ingress-mac = sha3.update(mac-secret ^ initiator-nonce || auth-recvd-ack)
# destroy remote-nonce

Recipient:
egress-mac = sha3.update(mac-secret ^ initiator-nonce || auth-sent-ack)
# destroy nonce
ingress-mac = sha3.update(mac-secret ^ recipient-nonce || auth-recvd-init)
# destroy remote-nonce
```

Creating authenticated connection:

```
1. initiator generates auth from ecdhe-random, static-shared-secret, and nonce (auth = authInitiator handshake)
2. initiator connects to remote and sends auth

3. optionally, remote decrypts and verifies auth (checks that recovery of signature == H(ephemeral-pubk))
4. remote generates authAck from remote-ephemeral-pubk and nonce (authAck = authRecipient handshake)

optional: remote derives secrets and preemptively sends protocol-handshake (steps 9,11,8,10)
```

```
5. initiator receives authAck
6. initiator derives shared-secret, aes-secret, mac-secret, ingress-mac, egress-mac
7. initiator sends protocol-handshake

8. remote receives protocol-handshake
9. remote derives shared-secret, aes-secret, mac-secret, ingress-mac, egress-mac
10. remote authenticates protocol-handshake
11. remote sends protocol-handshake

12. initiator receives protocol-handshake
13. initiator authenticates protocol-handshake
13. cryptographic handshake is complete if mac of protocol-handshakes are valid; permanent-token is
replaced with token
14. begin sending/receiving data

All packets following auth, including protocol negotiation handshake, are framed.
```

Either side may disconnect if and only if authentication of the first framed packet fails, or, if the protocol handshake isn't appropriate (ex: version is too old).

Framing

The primary purpose behind framing packets is in order to robustly support multiplexing multiple protocols over a single connection. Secondarily, as framed packets yield reasonable demarcation points for message authentication codes, supporting an encrypted stream becomes straight-forward. Accordingly, frames are authenticated via key material which is generated during the handshake.

When sending a packet over RLPx, the packet is framed. The frame header provides information about the size of the packet and the packet's source protocol. There are three slightly different frames, depending on whether or not the frame is delivering a multi-frame packet. A multi-frame packet is a packet which is split (aka chunked) into multiple frames because it's size is larger than the protocol window size (pws; see Multiplexing). When a packet is chunked into multiple frames, there is an implicit difference between the first frame and all subsequent frames. Thus, the three frame types are normal, chunked-0 (first frame of a multi-frame packet), and chunked-n (subsequent frames of a multi-frame packet).

```
  normal = not chunked
  chunked-0 = First frame of a multi-frame packet
  chunked-n = Subsequent frames for multi-frame packet
  || is concatenate
  ^ is xor

Single-frame packet:
header || header-mac || frame || frame-mac

Multi-frame packet:
header || header-mac || frame-0 ||
[ header || header-mac || frame-n || ... || ]
header || header-mac || frame-last || frame-mac

header: frame-size || header-data || padding
frame-size: 3-byte integer size of frame, big endian encoded (excludes padding)
header-data:
    normal: rlp.list(protocol-type[, context-id])
    chunked-0: rlp.list(protocol-type, context-id, total-packet-size)
    chunked-n: rlp.list(protocol-type, context-id)
    values:
        protocol-type: < 2**16
        context-id: < 2**16 (optional for normal frames)
        total-packet-size: < 2**32
padding: zero-fill to 16-byte boundary

header-mac: right128 of egress-mac.update(aes(mac-secret,egress-mac) ^ header-ciphertext).digest

frame:
    normal: rlp(packet-type) [|| rlp(packet-data)] || padding
    chunked-0: rlp(packet-type) || rlp(packet-data...)
    chunked-n: rlp(...packet-data) || padding
padding: zero-fill to 16-byte boundary (only necessary for last frame)

frame-mac: right128 of egress-mac.update(aes(mac-secret,egress-mac) ^ right128(egress-
mac.update(frame-ciphertext).digest))

egress-mac: h256, continuously updated with egress-bytes*
ingress-mac: h256, continuously updated with ingress-bytes*
```

Message authentication is achieved by continuously updating egress-mac or ingress-mac with the ciphertext of bytes sent (egress) or received (ingress); for headers the update is performed by xoring the header with the encrypted output of it's corresponding mac (see header-mac above for example). This is done to ensure uniform operations are performed for both plaintext mac and ciphertext. All macs are sent CLEARTEXT.

Padding is used to prevent buffer starvation, such that frame components are byte-aligned to block size of cipher.

`context-id` distinguishes concurrent packet transfers within a protocol. All chunked frames for a certain packet share the same `context-id`. The assignment of packet ids is implementation-defined. The ids are local to the protocol-type and not the connection; two protocols may reuse the same context ids.

Notes on Terminology:

"Packet" is used because not all packets are messages; RLPx doesn't yet have a notation for destination address. "Frame" is used to refer to a packet which is to be transported over RLPx. Although somewhat confusing, the term "message" is used in the case of text (plain or cipher) which is authenticated via a message authentication code (MAC or mac).

Flow Control

Note: The initial version of RLPx will set a static window-size of 8KB; fair queueing and flow control (DeltaUpdate packet) will not be implemented.

Dynamic framing is a process by which both sides send frames which are limited in size by the sender window size and the number of active protocols. Dynamic framing provides flow control and is implemented by a sender transfer window and protocol window. The data transfer window is a 32-bit value set by the sender and indicates how many bytes of data the sender can transmit. The protocol window is the transfer window, divided by the number of active protocols. After a connection is established, but before any frames have been transmitted, the sender begins with the initial window size. This window size is a measure of the buffering capability of the recipient. The sender must not send a data frame larger than the protocol window size. After sending each data frame, the sender decrements its transfer window size by the amount of data transmitted. When the window size becomes less than or equal to 0, the sender must pause transmitting data frames. At the other end of the stream, the recipient sends a DeltaUpdate packet back to notify the sender that it has consumed some data and freed up buffer space to receive more data. When a connection is first established the initial window size is 8KB.

```
pws = protocol-window-size = window-size / active-protocol-count

The initial window-size is 8KB.
A protocol is considered active if it's queue contains one or more packets.

DeltaUpdate protocol-type: 0x0, packet-type: 0x0
struct DeltaUpdate
{
        unsigned size; // < 2**31
}
```

Multiplexing of protocols is performed via dynamic framing and fair queueing. Dequeuing packets is performed in a cycle which dequeues one or more packets from the queue(s) of each active protocol. The multiplexor determines the amount of bytes to send for each protocol prior to each round of dequeuing packets.

If the size of a frame is less than 1 KB then the protocol may request that the network layer prioritize the delivery of the packet. This should be used if the packet must be delivered before all other packets. The senders network layer maintains two queues and three buffers per protocol: a queue for normal packets, a queue for priority packets, a chunked-frame buffer, a normal-frame buffer, and a priority-frame buffer.

```
If priority packet and normal packet exist: send up to pws/2 bytes from each (priority first!)
else if priority packet and chunked-frame exist: send up to pws/2 bytes from each
else if normal packet and chunked-frame exist: send up to pws/2 bytes from each
else read pws bytes from active buffer

If there are bytes leftover -- for example, if the bytes sent is < pws, then repeat the cycle.
```

Release History

- Upcoming
  - flow control
  - capabilities
- Version 0.x (in progress)
  - IPv6 and split-port endpoints
  - External IP & Public Key discovery
  - Move protocol-type to frame
- Version 0.3:
  - ignore unsolicited messages
  - persistence (Go)
- Version 0.2:
  - versioning
  - persistence (C++)
- Version 0.1:
  - Encrypted/Authenticated TCP
  - Basic TCP Framing (only frame-size is used)
  - Signed UDP (via ecdsa)
  - Basic Node Discovery
  - removal of TCP peer sharing

Implementation Status

Client implementation status of RLPx.
x! py implemented but not active

Known Issues

- Clients use left128 instead of right128
- whether to update mac state after sending mac digest
- Go/C++ client distance is xor(pubkA,pubkB) instead of sha3(pubkA,pubkB)

Node Discovery

- Go [ ] C++ [ ] Py [ ] Java [ ]: proper endpoint encapsulation
- Go [ ] C++ [ ] Py [x] Java [ ]: distance based on xor(sha3(NodeIdA),sha3(NodeIdB))
- Go [ ] C++ [ ] Py [ ] Java [ ]: refresh: perform discovery of random target every 56250ms
- Go [x] C++ [ ] Py [x] Java [ ]: timeout any packet operation if response is not received within 300ms
- Go [x] C++ [x] Py [ ] Java [ ]: ignore unsolicited messages
- Go [ ] C++ [x] Py [x] Java [ ]: tcp addresses are only updated upon receipt of Pong packet
- Go [x] C++ [x] Py [ ] Java [ ]: init udp addresses determined by socket address of recvd Ping packets
- Go [x] C++ [x] Py [x] Java [ ]: init tcp address determined by contents of Ping packet
- Go [x] C++ [x] Py [x] Java [ ]: perform discovery protocol with a concurrency of 3
- Go [x] C++ [x] Py [x] Java [ ]: Signed discovery packets
- Go [ ] C++ [ ] Py [ ] Java [ ]: discovery performed via 8 bits-per-hop routing

WiP:

- peer protocol maintains an ideal peer count

Authentication Handshake

- Go [x] C++ [x] Py [x] Java [ ]: Initiation and Acknowledge implemented with ECIES std w/AES128-CTR
- Go [x] C++ [x] Py [x] Java [ ]: Initiation, Acknowledgement, and Authentication for unknown node
- Go [x] C++ [ ] Py [ ] Java [ ]: Initiation, Acknowledgement, and Authentication for known node
- Go [x] C++ [x] Py [x] Java [ ]: Derive shared secrets from handshake
- Go [x] C++ [x] Py [ ] Java [ ]: Move capabilities into authentication payload (replaces Hello packet)

Framing

- Go [x] C++ [x] Py [x] Java [x]: mac of header and frame
- Go [ ] C++ [ ] Py [x] Java [ ]: basic frame, move protocol-type into frame header (replace magic sequence w/frame header)
- Go [ ] C++ [ ] Py [ ] Java [ ]: chunking with static 1KB frame size (requires fair queueing)
- Go [ ] C++ [ ] Py [x] Java [ ]: dynamic framing
- Go [ ] C++ [ ] Py [ ] Java [ ]: sequence-ids for non-chunked packets

Flow Control

- Go [ ] C++ [ ] Py [ ] Java [ ]: fair queueing (required for chunking)
- Go [ ] C++ [ ] Py [ ] Java [ ]: DeltaUpdate packet (required for fair queueing)
- Go [ ] C++ [ ] Py [x] Java [ ]: dynamic framing

Encryption

- Go [x] C++ [x] Py [x] Java [x]: AES256 CTR

References

- Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. 2002. URL {http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf}
- Victor Shoup. A proposal for an ISO standard for public key encryption, Version 2.1. 2001. URL {http://www.shoup.net/papers/iso-2_1.pdf}
- Gavin Wood. libp2p Whitepaper. 2014. URL {https://github.com/ethereum/wiki/wiki/libp2p-Whitepaper}
- Mike Belshe and Roberto Peon. SPDY Protocol - Draft 3. 2014. URL {http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3}
- Vitalik Buterin. Ethereum: Merkle Patricia Tree Specification. 2014. URL {http://vitalik.ca/ethereum/patricia.html}

Contributors

(Contributors: Please create PR w/name or alias to reference)

- protocol-type into frame header from packet-type offset
- considering unlimited channels
- full review, bug fixes
- review of cryptography
- intersection of nodes
- devp2p protocol and paper

RLPx was inspired by circuit-switched networks, BitTorrent, TLS, the Whisper protocol, and the PMT used by Ethereum.