

End-to-end encrypted protocol structure for Peer-to-Peer messaging

Problem description

When we have a traditional Client-Server architecture, messaging (even end-to-end encrypted, e.g. when only receiver can decrypt message, not server) is fairly simple. Server provides Application Programming Interface, implementing authentication, messaging and signaling (e.g. typing notification, message read notification, delivery notification etc.).

But in case of a peer-to-peer messenger, where both signaling and messaging go directly and server is used only for direct connection establishment, each client becomes a small server, providing Application Programming Interface to other clients.

It brings some complexity in distinguishing signaling and messaging if we would try to make this communication highly secured. To increase level of security, the application must not have access to long-term keys before user enter his password. At the same time, we need to “forget” these keys if user doesn’t interact with application for a long time. “Forgetting” here means putting zero or random bytes in RAM, where long-term keys have been loaded. At the same time, we surely want to receive and save messages even if we don’t have long-term keys available to notify user that he or she have new incoming messages. And, of course, we don’t want to notify user on every single event like “typing notification”, “part of a file loaded”, “your message has been read” etc.

Everything mentioned above, require a certain way to distinguish “signaling” traffic from “messaging” traffic, keeping user’s privacy at high level.

Another problem of any peer-to-peer protocol is a so-called offline synchronization. It is obvious, that peer-to-peer messaging require both users to be online simultaneously. But it is not necessary true in reality and we may want to provide the ability to synchronize messages even if users don’t come online simultaneously, keeping the best of P2P security, e.g. not persisting messages anywhere.

Proposed solution

Let's see first, what we may want to do on different incoming events:

№	Incoming packet type	Actions required
1	Message	We want to save this packet and notify user
2	Typing notification	We don't want to save nor to notify
3	Message read notification	We want to save, but not notify
4	Group chat photo changed	We want to notify, but not to save (we may load image during group information synchronization)

In this table, we can see 4 possible types of behavior when we have different types of incoming packets:

- Messages – packets with message content;
- Typing notification – packet, that indicates that another user is typing in chat;
- Message read notification – packet, that indicates that another user has read previously sent message;
- Group photo changed – packet, that indicates that group chat photo has been changed by someone.

Obviously, we have 2 sorts of behavior with respect to notification – we may want to notify or may not want to do it – and 2 sorts of behavior with respect to persistently storing these messages – we may want to save message in persistent storage or may not want to do it.

As it can be seen in first table, in ordinary messenger we have all combinations of these 2 sorts of behavior.

At the same time, we should remember, that we don't have key information, required to distinguish different types of packets before user logs in.

This is where the proposed solution comes to scene.

First, let's assume we have 2 pairs of certificate and private key. First pair is available all the time, in second pair we have certificate available all the time and private key available only after user enter password.

Let's call the always-available pair Transport pair and partly available – User pair.

The essence of the proposed solution is to have 2 separate ephemeral keys, one generated using Transport pair (this ephemeral key, as a Transport pair itself, is available all the time) and one available only after User pair is fully available.

Now we can separate our packet into two parts:

- First part is encrypted using Transport ephemeral key;
- Second part is encrypted using User ephemeral key.

Thus, we'll have a packet header, always available for decryption, where we can put information about this packet.

But, as a matter of fact, Transport-encrypted packet header is less secure, than User-encrypted data. So, we may not want to put the exact packet type there to prevent data leak.

As mentioned above, we actually have 2 general parameters we need to know about packet – weather we should notify user and weather we should save it. So, we may put not the exact message type but just these two parameters. Then, if we have multiple packet types for each flags combination, we won't leak too much data if transport header will be decrypted.

Using these technique, the overall system is shown on fig. 1.

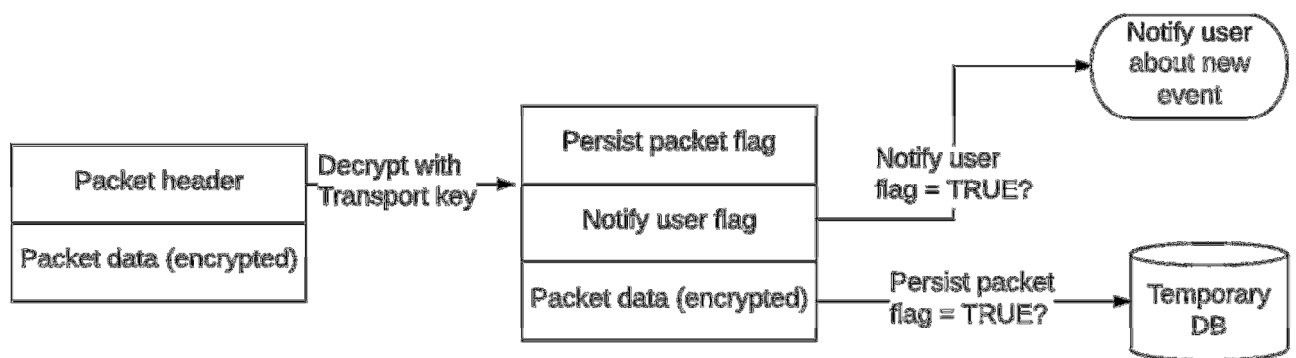


Fig. 1. Using distinguished encryption when User keys are not available

When User private key becomes available (user logs in application), the process goes according Fig. 2.

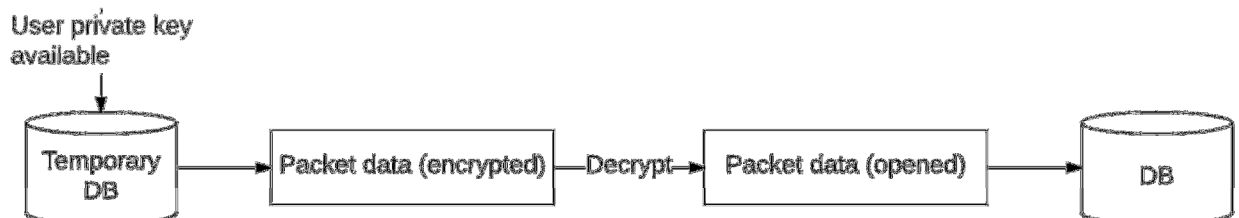


Fig. 2. Extraction of persisted encrypted packet

As it could be seen from Fig. 1 and 2, Temporary DB contains encrypted packets, if their header contained Persist flag. When User private key becomes available, it is used to extract data from Temporary DB.

It should be noted here, that this mechanism becomes useless if we have packets with fixed-size in our system, because encryption keeps packets size (depending on algorithm, it may be up-rounded to the cypher block size) and packet type may be defined just by its size.

To avoid it, we should extend fixed-size packets on random number of blocks, that should be discarded during decryption. Thus, typing-notification have no variations in its size and should be randomized. Randomization is even more important, if intruder knows about distinguishing protocol and will – because, if he knows the specific packet to by typing notification and that it has Transport header with 2 flags set to 0 (don't persist, don't notify), intruder may attempt to attack transport encryption.

There is a simple rule to select the minimum number of blocks. For example, we have 3 types of packets. One of them have size of 2 blocks (fixed), another can vary from 5 to 10 blocks, and the last one can be 4-6 blocks.

To prevent 3d party guessing the packet type, we should make all these packets to have the same lower bound (in this case – 5 blocks) and varying upper bound to prevent statistical guessings. So, for the first packet we should have at least 3 blocks added and for the third packet – at least 1 block added if it has length of 4 blocks.

Another mechanism, that should make attacks on Transport headers more complicated is header randomization. We use only 2 bits of our header – for two flags. But on most symmetric cyphers we'll have this header 1-block length after encryption. So, it is reasonable to randomize all other, unused header bits.

So, the multi-layer security mechanism includes:

- 1) 2 key pairs – Transport and User
- 2) 2 ephemeral keys, negotiated during connection establishment, 1 key relies on Transport certificate and private key and 1 key relies on User certificate and private key
- 3) Transport header randomization (for all non-meaningful bits)
- 4) User data size randomization

And addresses the following issues:

- 1) Correct behavior when receiving packets of different types even if decryption of these packets is impossible at the time of receiving (long-term keys are unavailable);
- 2) If application data will be compromised (meaning, open data become available to the 3d party), only transport layer of this protocol will be compromised, meaning the 3d party will only be able to detect types of packets, not the content itself;

- 3) There will be very little possibility to distinguish encrypted packets by their size, because the size is variable even for packets with fixed payload;

Let's now address the second issue – offline synchronization, keeping P2P-style behavior. It is obvious, that to provide “delayed” synchronization we have to save messages somewhere. It surely should be a “server” or some “node” with permanent availability. First of all, we have to be sure, that message is not saved on any persistent storage (e.g. SSD or HDD). It should be saved in RAM. In this case, the behavior will be the same like with ordinary network packets – they are also saved in RAM of routers and switches, while traveling from peer to peer. In case a server or a node is rebooted, this message would be lost. In terms of programming, this is not called “persisting”, so we are still in the field of P2P-style behavior.

The hard part here is synchronization. We should assume two cases:

- 1) Message delivered to a peer successfully – in this case, the second peer should be notified “message was delivered”
- 2) Message failed to deliver (for example, a node was rebooted) and a new P2P connection established – in this case, we have to synch non-synched messages via P2P

So, it is clear now, that we have to have message state (sent-to-node, delivered-to-peer) to be persisted on device and synchronization of that state should go through a server. We have to extend our protocol with signaling packet “message delivered”, that can also be synched via P2P or a server. This will address both cases – in first case, even for asynchronously-online users, sender will receive a “message-delivered” packet and will notify user that specific message was delivered, in second case, sender of a message, when P2P session is established, will notice, that he has non-synched messages and synch via P2P.

It should be noted here, that, if receiver receives 2 same messages, he should only persist one of them, because in some race conditions we can have situations when sender tries to synch messages that are already received.

It differs from traditional usage of P2P synchronization. When we use only P2P for sending and receiving packets we can rely on the protocol stack level that ensures delivery (so-called transport level, e.g. TCP or custom reliability level). In our case of asynchronously online users, reliability will only provide us with information about delivery to the node or server, not to the actual receiver.