High Level Design Considerations for PSA Crypto API PAKE interface

Overview

Components of (a broadly interpreted) PAKE

A complete PAKE protocol can be divided into several parts:

1. **Out of band set up:** In this phase the peers use an authenticated and secure channel to set up the secrets they need to perform the protocol. This is mentioned but out of scope for most of the studied standards. The exception is OPAQUE, but even there it is a distinct registration protocol and is well separated from the actual handshake. This clearly should be out of scope for the API as well.
2. **Exchange plaintext info:** All of these protocols start by sending some plaintext information, for example some form of identity, session/channel identification or salt. I can't think of a good reason for including this stage in the API, this is about exchanging plaintext messages, no need to support it with a Crypto API.
3. **Memory-hard hash:** Some standard mention that the password might need to be hashed with a memory-hard hash function, but they consider it out of scope for the protocol. (With the exception of OPAQUE, where it is integral part of the protocol.)
4. **Key agreement:** The actual exchange of key shares and the derivation of key material.
5. **Key confirmation:** This typically happens by using a MAC over some kind of a handshake transcript. Some standards consider it out of scope, some of them have implicit key confirmation, some allow several options and it is well defined for some others.
6. **Key derivation:** Some protocols result in raw key material, some of them in shared key that could in theory be used directly. This is however bad practice and the API should make it at least as difficult to use them directly as through key derivation. Based on this, the output of the protocol should be accessible through a key derivation method. This could be done preferably as existing key exchanges do it in PSA Crypto or maybe some other way if necessary. This however is a detail and I don't think that it is worth discussing at this stage.

In summary, steps 1. and 2. are out of scope for the API and we defer discussing 6. for a later time and thus that is out of scope for this document as well.

Classes of PAKE

There are two main classes of PAKE:

1. **Balanced:** These protocols are symmetric, both peer holds the same, shared secret and the protocol flow is identical. That said, in some of them

the transcript has to be the same on both sides and the API needs to know the order in which the messages were sent.

2. **Augmented:** There is a distinction between the client and the server and the server doesn't know the password. (That is, even if the server is compromised, the adversary can't impersonate the client without brute forcing the password.)

During the investigations it turned out that the protocol flow is divided along different lines. In this document we will call SRP, SPAKE2, SPAKE2+ and CPace **conventional** protocols as opposed to OPAQUE and JPAKE, which we will call **complex**.

Usability vs Flexibility

The protocols that we need to cover show significant differences and we will need a great deal of flexibility in the API. However the more flexible the API is, the more knowledge it requires from the user about the protocol.

Design Questions

- Is it worth to tie in some way the memory-hard hashing step into the protocol flow?
- Should we have a separate API for key confirmation or should we hardwire it into the PAKE API flow?
- Should we have separate APIs for augmented and balanced protocols or it is feasible and better to go with an identical API for both?
- What trade-offs are available and which should we take regarding usability and flexibility?

Memory-Hard Hashing

Some protocols (SRP, J-PAKE) don't even mention the topic and goes with normal hashing, but of course any pre-agreed method is still applicable. Some of them (SPAKE2, SPAKE2+, CPace) discuss the matter, but don't mandate anything specific.

The exception is OPAQUE, for which it is integral part of the protocol. That said, even if applying memory-hard functions (MHF) is part of the API, OPAQUE remains an exception, because the MHF is not applied directly on the password and can't be done upfront, like in the case of other protocols.

Adding the MHF hashing step to the API would incentivise developers to use MHF in their application and thus would improve security. The security improvement might be marginal, since the use of MHF step can be enforced by key policy anyway (allow only to use password based key derivation on the password key).

However, using MHF has to be pre-agreed between the peers and has implications on credential storage. Decisions have to be made on a higher, security design/policy level and it takes more than just the developer using the right API.

Also, adding MHF hashing step comes with a price: increases the complexity of the API and impairs usability.

Based on the above, I don't think we should include MHF in the API and we should settle for mentioning it in the API.

Key confirmation

CPace doesn't even mention key confirmation. The J-PAKE standard doesn't mandate key confirmation it merely recommends it and gives two example methods. For SRP, OPAQUE, SPAKE and SPAKE2+ key confirmation is part of the protocol. SPAKE and SPAKE2+ even mandates that the key MUST NOT be used before the key confirmation completes.

Where key confirmation is present it is often interleaved with other messages to reduce the number of roundtrips. The similarity of key confirmation is a great opportunity to provide a simple and unified interface for it across all the PAKEs. It might mean an extra function call for the user and thus it is one of the flexibility vs usability tradeoffs. Clean separation the key confirmation is possible in all studied protocols except OPAQUE, for which parts of the key confirmation might happen earlier, depending on the design of the API.

Even if key confirmation is part of the protocol definition, this part might be omitted. This typically would happen when embedding the PAKE in a higher level protocol. In this case the higher level protocol takes care of the key confirmation, like in the case of TLS-SRP.

If the PSA Crypto implementations are not required to implement key confirmation in these schemes, then the API will need to be able to provide keys with either implicit or explicit key confirmation.

Before the key confirmation completes the schemes provide only implicit key confirmation for the key. This means that at this point we don't know who are we talking to. We just know that if we are not talking to whom we want, they can't compute the key. After the key confirmation completes the schemes provide explicit key confirmation for the key. At this point we know that we are talking to whom we want to. This is a difference in security guarantees and if extraction of implicitly confirmed keys is allowed, then we should sharply differentiate between them (eg. by different function calls).

If we don't allow extraction of keys with implicit confirmation the implementations will need provide key confirmation even for schemes where it is not defined by the standard. In the case of J-PAKE this means implementing one or two key confirmations that were given as examples and might or might not serve the

users purposes. In the case of CPace the uncertainty and probability of different implementors coming up with an interoperable solution is even smaller.

Some schemes output raw keying material that must not be used as keys directly, some others session keys that in theory could be used directly (but in practice they still should be used in key derivation as a typical session needs at least two symmetric keys). To make embeddings in higher level protocols possible, the user might need to extract raw keying material even if the scheme outputs session keys. This happens for example in the case of TLS-SRP. Many of the PAKE schemes have security proofs and extracting the raw keying material invalidates them. If the API allows for such extraction, this must be constrained and exceptional (eg. separate API function, with tons of warnings, and constrained to specific PAKE/higher level protocol pairs).

If the standard mandates explicit key confirmation, then giving the user access to implicitly confirmed keys is controversial and probably best handled similarly to the key extraction mentioned above.

In summary, in the output we need to differentiate between the following:

- implicit vs explicit
- key vs key material
- key extraction

Questions

Do we want to

- have unified key confirmation API?
- allow access to keys with implicit key confirmation?

The first question ties into the complexity trade-off, but based on the above it is worth having a separate key confirmation stage. For the second question no would be the pedantic answer, however allowing it is unavoidable. Not supporting embedding and forcing key confirmation behind the API impacts usability so much, that it is way too high of a price for a slight gain in security.

Augmented vs Balanced Split

Balanced

Although they are called balanced and the protocol flow is symmetric, SPAKE2 and CPace needs to have an agreed order between the identities to calculate the message transcript. This is not like a client server division, but reduces the possible benefits gained by separating the API.

If we separate the key confirmation, the interfaces for SPAKE2 and CPace are quite similar: taking the parties identities and some session information as an input, they produce a session key. They both exchange key shares that are group elements. These similarities would make API design easier, but unfortunately

J-PAKE is very different: it has more rounds, needs two key shares in the first round and a third one in the second round. It also requires to send zero knowledge proofs along with each key share and it has a massively different abstract API.

In summary, I can't see any obvious advantage of having a dedicated balanced PAKE API.

Augmented

SPAKE2+ and SRP follow the standard flow: they have some session and identity information, exchange key shares that are group elements. SPAKE2+ needs two values for verification, but that is not an insurmountable obstacle.

OPAQUE on the other hand is a combination of two protocols and thinks in terms of messages. It requires the exchange of several group elements and elaborate structures for his operation. Unlike other protocols it considers memory-hard hashing, key confirmation and communicating session information (steps 2, 3 and 5) in scope. These are built in the protocol, removing them wouldn't be a trivial exercise and would increase the probability of implementation mistakes on both sides of the API.

Can't discern any obvious advantage for the separation on the augmented side either. A natural line of separation seems to be between schemes that follow the usual setup info and exchange atomic key shares pattern and those which require the exchange of complex messages.

Existing APIs

Found no signs of any PAKE APIs in Open SSL or GCrypt. (Open SSL used to have J-Pake, but it was experimental and has been removed.) Found an SRP patch for NSS (https://sqs.me/security/tls-srp-in-nss.html).

Bouncy Castle

J-Pake is a separate class directly from object, to be used directly (no factories involved) no universal PAKE API is apparent:
https://javadox.com/org.bouncycastle/bcprov-jdk15on/1.51/org/bouncycastle/crypto/agreement/jpake/JPAKE
Same goes for SRP:
https://www.bouncycastle.org/docs/tlsdocs1.5on/org/bouncycastle/tls/crypto/impl/jcajce/srp/SRP6Client.htm

No CPace, OPAQUE, SPAKE2 or SPAKE2+ implementation found in bouncy castle.

High level approach

The setup of the context can vary slightly even with the most conventional schemes and the security properties of the output can vary as well. Still the

API for these should be relatively straightforward and I am happy to discuss and work out the details during review. The question remains the high level approach on the main operation part of the protocols for which there are several possible options.

Streaming

The API treats the schemes as protocols. The main flow is executed by two functions: one of which provides a bytestream output and the other which processes bytestream input received from the peer. The user sets up the context and then runs a simple loop and in the end the handshake either completed or errored out.

Pros:

- easy to use
- maximal algorithm agility
- allows memory efficient implementation

Cons:

- difficult to debug, pressure on error codes
- call structure can hinder embedding in higher level protocols

Pseudocode example:

```
psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_JPAKE, ... );
psa_status_t ret;

do {
    if(ret == PSA_NEED_TO_WRITE) {
        ret = psa_pake_write(op, uint8_t* out, size_t out_len);

        // Add code for processing/sending output
    }
    else if(ret == PSA_NEED_TO_READ) {
        // Add code for generating/receiving input

        ret = psa_pake_read(op, const uint8_t* in, size_t in_len);
    } else {
        goto cleanup;
    }
} while(ret != PSA_SUCCESS);
```

Message based

After an initial setup there is a uniform function call taking messages as inputs and producing messages as outputs. This similar to the streaming option,

but for more complex schemes it can be less efficient in terms of memory (for standard schemes there is very little difference). Another drawback is that the implementations need to take message format into account. This is not a problem for most of the schemes, but can increase implementation complexity when embedding the PAKE in a larger protocol.

Pros:

- easy to use
- maximal algorithm agility

Cons:

- message format can hinder embedding in higher level protocols

Pseudocode example:

```
psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_JPAKE, ... );
psa_status_t ret;
int rounds = 0;

while(rounds++ < PSA_PAKE_ROUNDS(op)) {
    ret = psa_pake_step(op, const uint8_t* in_msg, size_t in_msg_len,
                        uint8_t* out_msg, size_t out_msg_size, size_t* out_msg_len);
    if(ret != PSA_SUCCESS)
        goto cleanup;

    // send and receive messages
}
```

Maximum flexibility

The API consists of two functions: one of which inputs a parameter to the context and one of which outputs a value from the context. Optionally a third could be added that moves the state machine forward explicitly but I am not sure how that would be useful. The input and output functions have a parameter that inform the API about which parameter is being received or requested. For maximum flexibility, this is a string and a different set of such attributes are defined for each protocol. Opposed to the message based one, this solution inputs and outputs atomic values (like character string, big integer or group element).

This is more efficient in terms of memory than the message based when it comes to more complex schemes, but it requires close familiarity with the scheme and provides opportunity for much more differences between the call sequences accepted by various implementations.

Pros:

- High flexibility

Cons:

- Very little algorithm agility in practice
- difficult to use (user needs to know the scheme in detail)

For example JPAKE would look something like this:

```
#define CHECK_RET(f)                                \
    do                                              \
    {                                               \
        if( ( ret = (f) ) != PSA_SUCCESS )          \
            goto cleanup;                           \
    } while( 0 )

psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_JPAKE, ... );
psa_status_t ret;

CHECK_RET(psa_pake_output("epk1", epk1, sizeof(epk1), &epk1_len));
CHECK_RET(psa_pake_output("zkp1", zkp1, sizeof(zkp1), &zkp1_len));
CHECK_RET(psa_pake_output("epk2", epk2, sizeof(epk2), &epk2_len));
CHECK_RET(psa_pake_output("zkp2", zkp2, sizeof(zkp2), &zkp2_len));

// send and receive first round

CHECK_RET(psa_pake_input("pepk1", pepk1, pepk1_len));
CHECK_RET(psa_pake_input("pzkp1", pzkp1, pzkp1_len));
CHECK_RET(psa_pake_input("pepk2", pepk2, pepk2_len));
CHECK_RET(psa_pake_input("pzkp2", pzkp2, pzkp2_len));
CHECK_RET(psa_pake_output("epk3", epk3, sizeof(epk3), &epk3_len));
CHECK_RET(psa_pake_output("zkp3", zkp3, sizeof(zkp3), &zkp3_len));

// send and receive second round

CHECK_RET(psa_pake_input("pepk3", pepk3, pepk3_len));
CHECK_RET(psa_pake_input("pzkp3", pzkp3, pzkp3_len));
```

Extended Flexible

Design an API for the conventional PAKE flow (setup, exchange atomic key shares, etc.) and add functions for adding further inputs and getting more output for handling more complex schemes. This can be done by universal functions like in the flexible option above, or could potentially be done by ad hoc extensions.

For simple schemes this is easier to use and easier to implement than other options, and shares some of the disadvantages of the maximum flexibility option for more complex schemes.

Pros:

- High flexibility
- Algorithm agility for conventional protocols
- Conventional protocols are easy to use

Cons:

- Very little algorithm agility for complex protocols
- Complex protocols are difficult to use (user needs to know the scheme in detail)

For example CPace would look like this with the flexible API:

```
psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_CPACE, ... );
psa_status_t ret;

CHECK_RET(psa_pake_output("ks", key_share, sizeof(key_share), &key_share_len));

// send and receive key share

CHECK_RET(psa_pake_input("pks", peer_key_share, peer_key_share_len));
```

And like this in the extended version:

```
psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_CPACE, ... );
psa_status_t ret;

CHECK_RET(psa_pake_get_keyshare(key_share, sizeof(key_share), &key_share_len));

// send and receive key share

CHECK_RET(psa_pake_set_peer_keyshare(peer_key_share, peer_key_share_len));
```

These functions would be called by protocols like OPAQUE and ECJPAKE as well, they just would need to call `psa_pake_output()` and `psa_pake_input()` for their non-standard parameters. For example, this is how the first round of JPAKE would look like:

```
psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_JPAKE, ... );
psa_status_t ret;

CHECK_RET(psa_pake_get_keyshare(epk1, sizeof(epk1), &epk1_len));
CHECK_RET(psa_pake_output("zkp1", zkp1, sizeof(zkp1), &zkp1_len));
CHECK_RET(psa_pake_get_keyshare(epk2, sizeof(epk2), &epk2_len));
CHECK_RET(psa_pake_output("zkp2", zkp2, sizeof(zkp2), &zkp2_len));
```

Extended Message Based

After an initial setup there is a sequence of function calls (extra rounds and key confirmation optional) taking messages as inputs and producing messages as outputs. This is more readable than more universal options and shares the disadvantages of the pure message based approach.

Pros:

- Algorithm agility for almost all protocols
- Easy to use

Cons:

- Message format can hinder embedding in higher level protocols

This is how JPAKE first round would look like:

```
psa_pake_operation_t op = PSA_PAKE_OPERATION_INIT;
psa_pake_setup(op, PSA_ALG_JPAKE, ... );
psa_status_t ret;

CHECK_RET(psa_pake_get_keyshare(first_msg, sizeof(first_msg), &first_msg_len));
```

With this API JPAKE would need to call both get and set twice because of the extra round.

Parallel

Have a separate, specific API for the conventional PAKE flow (setup, exchange atomic key shares, etc.) and have flexible, streaming or message based API for complex schemes.

Unlike in the case of extended options, here instead of a hybrid API where the two approaches are moulded together, we have two parallel APIs one of which only supports conventional protocols.

For simple schemes this is easier to use and easier to implement than other options, and shares the disadvantages of the adopted option for more complex schemes. (Ideally for simple schemes this could be very similar to the existing key exchange API.)

Choice

We have discussed it on the weekly PSA Crypto working group meeting (Gilles, Andrew and Edmund present, thank you all!) and concluded that the best way forward is the extended flexible option because:

- The stream based approach is too high level, the flexible is to flexible to be helpful.

- We rejected the message based options, because we don't want to parse external messages inside the security boundary, reassembing them for the transcript is the safer option.
- The exptended flexible option represents the same level of trade-offs that are already present in the existing key agreement API

Cipher Suites

All of these PAKE protocols use lower level cryptographic primitives and algorithms and provide flexibility regarding these. These can be a combination of 2 or more algortithms which increases complexity rapidly. Furthermore the types of these algorithms is not at all homogeneous across different protocols.

Possible approaches:

- Enumerate all permitted combinations: some standards define their cipher suites and for these enumeration is the perfect choice. There are however others for which full enumeration would be necessary, which is cumbersome and lengthy. Also the available options are highly implementation dependent. On the other hand the corresponding code could be generated and I saw something like that already being used in PSA. Also, this would be redundant, because most of these would be specific to their PAKEs. Wouldn't be very flexible.
- Have a struct that contains key types and algorithm ids, which are interpreted by the different PAKE implementations. There are several options here too:
  - The structure is implementation defined, there is a single setter function, the calling order decides what is being set.
  - The structure is implementation defined, there is a single setter function with a step type parameter for setting up the cipher suite
  - The structure is defined in the spec, user sets it up by hand
  - The structure is implementation defined, there is a single constructor
- Don't have a separate cipher suite param, config the operation directly. It is slightly simpler, but prevents pre-defined cipher suites which is quite handy in the case of some PAKEs (like SPAKE2+ for example)

Going with the single constructor, as it seems to be the simplest option.

Scoping

Out of scope for the end of April deadline:

- Investigating the missing 4 schemes
- Key confirmation API (interaction with OPAQUE can be tricky)
- Getting explicit keys out
- Getting session keys out (as opposed to key material)
- Key extraction (where the standard mandates key confirmation, outputting implicitly confirmed keys is considered key extraction as well)

- Decrypting and getting encrypted peer information out
- Getting export key out

Literature

Explicit and implicit key confirmation and their relationship:

```
- Precise definition and proof: https://eprint.iacr.org/2019/1203.pdf
- Intuition about their nature: Stinson, D. "Cryptography: Theory and Practice"
```