Improved PKEX

Steve Meireles Lopes[*]
steve.meireles.001@student.uni.lu
University of Luxembourg
Esch-sur-Alzette, Luxembourg

Marjan Skrobot[†]
marjan.skrobot@uni.lu
University of Luxembourg
Esch-sur-Alzette, Luxembourg

Abstract

Keywords

PAKE, Protocol, PKEX

1 Introduction

The secure exchange of public keys over an insecure network is a fundamental challenge in cryptography. Public Key Exchange (PKEX) is a password-authenticated protocol proposed by Dan Harkins [2] addresses this challenge by allowing two entities to exchange their public keys without prior trust. It is also generally used in IoT environments since it is in the specification of the widely used Easy-Connect framework as one of the authentication protocols [5]. PKEX ensures that the public keys are cryptographically bound to their respective entities, protecting against Man-in-the-Middle (MITM) attacks. Additionally, the protocol includes a proof of possession mechanism, ensuring that each entity demonstrates ownership of the private key corresponding to its public key.

PKEX operates in two distinct phases:

- **Authentication Phase:** This phase uses ephemeral public keys generated by both entities. These keys are encrypted and exchanged using a shared secret, typically derived from a password. The exchanged keys are then used to derive a strong shared secret key. The phase employs SPAKE2, a secure password-authenticated key exchange protocol proposed by Michel Abdalla and Manuel Barbosa [1].
- **Reveal Phase:** In this phase, the two entities commit to exchanging their public keys and "reveal" them to each other. Proof of possession is established by signing a combination of the public key, the ephemeral public key, the entity's identity, and the ephemeral public key of the other party.

By splitting the process into these two phases, PKEX achieves secure key exchange and robust authentication. However, like all cryptographic protocols, PKEX has potential vulnerabilities and limitations that leave room for improvement. This paper examines these limitations and proposes an improved version.

2 Analysis

2.1 Detailed Definition

As described above PKEX has two phases.

2.1.1 Authentication phase. The next steps are done by both parties, Alice and Bob. Values known to each party: pw, Pi, Pr Alice picks random x and calculates X, Q_a, M:

$$x, X = x * G$$

$$Q\_a = H(pw) * Pi$$

$$M = X + Q\_a$$

Bob picks random y and calculates Y, Q_b:

$$y, Y = y * G$$

$$Q\_b = H(pw) * Pr$$

Alice sends $Alice, M$ to Bob. Bob derives values and generates an ephemeral secret key.

$$Q\_a = H(pw) * Pi$$

$$X' = M - Q\_a$$

$$N = Y + Q\_b$$

$$z = KDF - n(F(y * X', Alice|Bob|F(M)|F(N)|pw))$$

Bob sends Bob, N to Bob and Alice derives values and an ephemeral secret key.

$$Q\_b = H(pw) * Pr$$

$$Y' = N - Q\_b$$

$$z = KDF - n(F(x * Y', Alice|Bob|F(M)|F(N)|pw))$$

2.1.2 Reveal phase. Here $a = F(B)$ is a function that takes an element and returns a scalar. Alice picks random a and calculates A, then derives u.

$$a, A = a * G$$

$$u = HMAC(F(a * Y'), Alice|F(A)|F(Y')|F(X))$$

Alice sends A, u using authentication encryption with the first half of the key naming it z_0. Bob checks the encryption and derives values.

$$if(AE - decryptreturnsfail)fail$$

$$if(Anotvalidelement)fail$$

$$u' = HMAC(F(y * A), Alice|F(A)|F(Y)|F(X'))$$

$$if(u'! = u)fail$$

$$v = HMAC(F(b * X'), Bob|F(B)|F(X')|F(Y))$$

Bob sends B, v using authentication encryption with the second half of the key naming it z_1. Alice checks the encryption and derives values.

$$if(AE - decryptreturnsfail)fail$$

$$if(Bnotvalidelement)fail$$

$$v = HMAC(F(x * B), Bob|F(B)|F(X)|F(Y'))$$

$$if(v'! = v)fail$$

[*]Bachelor Student
[†]Tutor of this project

2.2 Limitations

PKEX has several limitations. First, the security of the protocol has not yet been formally proven. There are several reasons for this. One reason, as mentioned by the author, is that he is not an academic expert and may not have the resources or expertise to conduct formal proofs. However, the protocol's design has inherent challenges to proving its security. Specifically, the reuse of ephemeral keys in both the authentication and reveal phases complicates the proof for several reasons:

- The reuse of ephemeral keys
  - the constructor __init__() which initializes the all the necessary values
  - start() which generates the ephemeral public key
  - finish(inbound_message) which authenticates the opposite entity and
    generates an ephemeral secret key

  creates dependencies between the two phases, making it more difficult to prove their security independently. Ideally, each phase should be isolated so that the security proof can be constructed without accounting for interactions between the phases.
- This dependency increases the potential for unforeseen vulnerabilities, as the interactions between the phases may introduce complex attack vectors. As a result, constructing and verifying a formal proof of security becomes significantly harder.
- The reuse of ephemeral keys might expose patterns in the two phases that an attacker could exploit, leading to the potential leakage of private key information. Consequently, the proof must account for how the Proof of Possession (PoP) guarantees hold in such scenarios.

The lack of modularity in PKEX further limits its flexibility. If the protocol were structured in a more modular and independent way, each phase could be modified or improved without impacting the others. Additionally, if a vulnerability were discovered in one phase, it could be replaced or updated without affecting the overall system. However, the current design ties the phases together, making it difficult to implement changes or fixes without disrupting the entire protocol.

3 Our Proposal

Our proposal aims to address the limitations identified in the PKEX protocol by proposing a more modular and flexible approach. Our protocol maintains the same security properties as PKEX while allowing for greater adaptability and ease of implementation.

3.1 Properties

Our proposal preserves the core security properties of PKEX:

- An adversary cannot subvert the exchange without knowing the password.
- An adversary cannot obtain the password through a passive attack.
- The protocol detects whether a guess of the password is correct.
- Proof of possession of the private key is ensured.
- At the end of the exchange, trust is established in the entity's public key, which is cryptographically bound to the entity's identity. The exchange fails if this binding is not confirmed.

3.2 Notation

For clarity, we introduce the following notation:

- Password or shared secret key: pw
- Private key: sk
- Public key: pk

3.3 Protocol Definition

Our protocol follows a two-phase structure, similar to PKEX, but with the flexibility to use different PAKE protocols and modular components.

3.3.1 Authentication Phase. In this phase, the protocol is very similar to the authentication phase in PKEX. However, unlike PKEX, which uses SPAKE2, our proposal allows using any secure Password-Authenticated Key Exchange (PAKE) protocol. This provides greater flexibility, enabling the use of more secure or efficient PAKE protocols as needed.

Let's assume two parties: Alice and Bob. They engage in a PAKE protocol to derive a shared secret and public key. The result is the following:

$$\text{Alice} \xrightarrow{\text{PAKE}} \text{Bob} \rightarrow (\text{shared secret } sk, \text{public key } pk)$$

3.3.2 Proof of Possession Phase. In the second phase, the Proof of Possession (PoP) mechanism is incorporated to verify the ownership of the private key corresponding to the public key exchanged in the previous phase. This is achieved by having the entities prove they possess the private key without revealing it. This ensures that only the entity with the correct private key can complete the exchange, providing additional security against impersonation and man-in-the-middle attacks.

The protocol's modularity allows for the replacement or enhancement of individual components, such as the PAKE or PoP methods, without requiring changes to the entire protocol. This design makes our protocol more adaptable and easier to update with newer cryptographic techniques as they become available.

Formal Definition:

Definition 3.1 (Key generation and proof of possession scheme). A key generation and proof of possession (KGPOP) scheme consists of:

- $PoP.KG() \rightarrow (pk, sk)$:A probabilistic key generation algorithm that outputs a public, secret key pair.
- $PoP.PG(pk, sk, attrs) \rightarrow \pi$: A probabilistic proof generation algorithm that takes as input a public key, secret key, and attributes $attrs \in \{0,1\}^*$, and outputs a proof string. (In our application, $attrs$ could be the body of a certificate signing request, for example.) In this case could be binding id with PAKE.
- $PoP.Vf(pk, attrs, \pi) \rightarrow \{0,1\}$: A deterministic verification algorithm that takes as input a public key pk, attributes $attrs$, and proof π, and outputs 1 if the proof is valid, and 0 otherwise.

3.4 Example implementation

In our implementation, we decided to use SPAKE2 as PAKE protocol, and for the proof of possession, we use Schnorr signature.

3.4.1 SPAKE2 Implementation. For the implementation we use the implementation of Warner [4]. The implementation provides a spake2.py file. Inside the file, there are several classes. We use the class "SPAKE2_A" which has following main methods.

- the constructor __init__() which initializes the all the necessary values
- start() which generates the ephemeral public key
- finish(inbound_message) which authenticates the opposite entity and generates an ephemeral secret key

Code Snippet:

```
    class SPAKE2_Base:
  "This class manages one side of a SPAKE2 key
  ↪ negotiation."

  side = None # set by the subclass

  def __init__(self, password,
               params=DefaultParams,
               ↪ entropy_f=os.urandom):
    assert isinstance(password, bytes)
    self.pw = password
    self.pw_scalar =
    ↪ params.group.password_to_scalar(password)

    assert isinstance(params, _Params),
    ↪ repr(params)
    self.params = params
    self.entropy_f = entropy_f

    self._started = False
    self._finished = False

  def start(self):
    if self._started:
      raise OnlyCallStartOnce("start() can only
      ↪ be called once")
    self._started = True

    g = self.params.group
    self.xy_scalar =
    ↪ g.random_scalar(self.entropy_f)
    self.xy_elem =
    ↪ g.Base.scalarmult(self.xy_scalar)
    self.compute_outbound_message()
    # Guard against both sides using the same
    ↪ side= by adding a side byte
    # to the message. This is not included in the
    ↪ transcript hash at the
    # end.
    outbound_side_and_message = self.side +
    ↪ self.outbound_message
    return outbound_side_and_message

  def finish(self, inbound_side_and_message):
    if self._finished:
      raise OnlyCallFinishOnce("finish() can
      ↪ only be called once")
    self._finished = True

    g = self.params.group
    inbound_elem =
    ↪ g.bytes_to_element(self.inbound_message)
    if inbound_elem.to_bytes() ==
    ↪ self.outbound_message:
      raise ReflectionThwarted
    #K_elem = (inbound_elem +
    ↪ (self.my_unblinding() * -self.pw_scalar)
    #          ) * self.xy_scalar
    pw_unblinding =
    ↪ self.my_unblinding().scalarmult(-self.pw_scalar)
    K_elem =
    ↪ inbound_elem.add(pw_unblinding).scalarmult(self.xy_sca
    K_bytes = K_elem.to_bytes()
    key = self._finalize(K_bytes)
    return key

class SPAKE2_A(SPAKE2_Asymmetric):
    side = SideA
    def my_blinding(self): return self.params.M
    def my_unblinding(self): return self.params.N
    def X_msg(self): return self.outbound_message
    def Y_msg(self): return self.inbound_message

class SPAKE2_B(SPAKE2_Asymmetric):
    side = SideB
    def my_blinding(self): return self.params.N
    def my_unblinding(self): return self.params.M
    def X_msg(self): return self.inbound_message
    def Y_msg(self): return self.outbound_message
```

3.4.2 POP. For the proof of possession, the Schnorr Signature is used. The POP phase consists of two parts.

The first part consists of key generation and signing: We use a hash key derivation function which we then split into three keys.

$$sk1, sk2, sk3 = HKDF(ephermal\_key)$$

Pick random scalar a (or b) and Compute A (or B)

$$A = G * a$$

or

$$B = G * b$$

Pick random scalar k and compute K:

$$K = G * k$$

Compute e and s such that:

$$e = H(K||A(orB)||PakeID)$$

$$s = k + e * (-a(orb))$$

Outbound message to send:

$$(A, s, e)$$

or

$$(B, s, e)$$

The second part consists of verification: Upon receiving the previous output you have:

$$A', s', e'$$

Re-calculate e:

$$e'' = H(K||A(orB)||PakeID)$$

Output:

$$True, if e'' == e'$$

$$False, if otherwise$$

Code snippet:

```python
class I_PKEX(SPAKE2_Asymmetric):

    def start_pkex(self, key):
        if not self._finished:
            raise RunSpakeFirst("start_pkex() may only be called when SPAKE2 protocol is finished running")

        g = self.params.group

        sk = HKDF(
                algorithm=hashes.SHA256(),
                length=len(key)*3,
                salt=None,
                info=None,
                ).derive(key)

        split_size = len(sk) // 3
        sk1 = sk[:split_size]
        assert len(sk1) == split_size, len(sk1)
        sk2 = sk[split_size:2*split_size]
        assert len(sk2) == split_size, len(sk2)
        self.pake_id = sk[2*split_size:]
        assert len(self.pake_id) == split_size, len(self.pake_id)

        # inbound_element is public key A (pA)
        # inbound_elem = g.bytes_to_element(self.inbound_message)

        # sk = a, pk = g * a = A
        self.ab_scalar = g.random_scalar(self.entropy_f)
        self.AB_element = g.Base.scalarmult(self.ab_scalar)

        # random k
        self.k = g.random_scalar(self.entropy_f)
        # print("ab_scalar: ", self.ab_scalar)
        # print("k", self.k)
        self.K_element = g.Base.scalarmult(self.k)

        self.e = hashes.Hash(hashes.SHA256())
        self.e.update(self.K_element.to_bytes() + self.AB_element.to_bytes() + self.pake_id)
        self.e = self.e.finalize()

        print("To be checked: ", self.e)
        self.e = g.password_to_scalar(self.e)

        print(self.k)
        self.s = (self.k + self.e * (-self.ab_scalar)) % L

        return (self.AB_element.to_bytes(), self.s, self.e)

    def finalize(self, key, data):
        g = self.params.group

        (AB_element, s, e) = data
        AB_element = g.bytes_to_element(AB_element)

        g_s = g.Base.scalarmult(s)
        pk_e = AB_element.scalarmult(e)

        K_element_check = g_s.add(pk_e)
        e_check = hashes.Hash(hashes.SHA256())
        e_check.update(K_element_check.to_bytes() + AB_element.to_bytes() + self.pake_id)
        e_check = e_check.finalize()
        e_check = g.password_to_scalar(e_check)
        print("To be checked: ", e)
        print("Checking e: ", e_check)

        return e_check == e
```

3.5 Benchmarks

Benchmarks	Average time for one exchange
PKEX	0,03060 seconds
Improved PKEX	0,03666

Table 1: Average time for one exchange

Benchmarks	Average time for one exchange
PKEX	0,35223 seconds
Improved PKEX	0,31128

Table 2: Average time for 10 exchanges

While the improved PKEX has a slightly higher cost for a single exchange maybe due to its modular components. It has a better performance over multiple exchanges. The reason may be because of the optimizations in key reuse.

4 Conclusion

In conclusion, while PKEX [2] is widely used and included in the Easy-Connect Specification, it exhibits several limitations. This paper thoroughly analyzes the PKEX protocol, identifies its shortcomings, and proposes an improved solution to address these issues.

The primary limitations of PKEX highlighted in this work are the lack of modularity and the reuse of ephemeral keys, which could impact both flexibility and security. The PKEX protocol is structured in two phases: the authentication phase and the reveal phase. The first phase employs the SPAKE2 protocol proposed by Abdalla [**?**], while the second utilizes values from the initial phase to achieve proof of possession and bind a private key to an entity.

Our proposed protocol addresses these limitations by ensuring that the second phase does not reuse values directly from the first phase, instead relying only on the output of the authentication phase. This approach improves modularity and security. In the authentication phase, any secure PAKE protocol such as SPAKE2 can be used. Similarly, any proof-of-possession mechanism such as the Schnorr Signature can be integrated into the second phase. This modularity allows for greater flexibility and the potential to adopt newer cryptographic primitives as they emerge.

To validate our proposal, we implemented and compared both protocols [3]. Benchmarking results show that while our protocol is slightly slower for single exchanges, it is faster when performing multiple exchanges, demonstrating its practical advantages in scenarios requiring repeated key exchanges.

4.0.1 Future Improvements. Even if our approach has obvious advantages, there is still room for improvement in a few areas. The protocol should have a formal proof of security since it would theoretically offer more robust security than PKEX.

References

[1] Michel Abdalla and David Pointcheval. 2005. Simple Password-Based Encrypted Key Exchange Protocols. In *Topics in Cryptology – CT-RSA 2005 (Lecture Notes in Computer Science, Vol. 3376)*. Springer, 191–208. https://www.di.ens.fr/~mabdalla/papers/AbPo05a-letter.pdf
[2] Dan Harkins. 2018. *Public Key Exchange*. Internet-Draft draft-harkins-pkex-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-harkins-pkex/06/ Work in Progress.
[3] Steve Meireles. 2025. I-PKEX: Proposal to improve PKEX.
[4] Brian Warner. 2025. python-spake2: A Python Implementation of SPAKE2.
[5] Wi-Fi Alliance. 2025. Wi-Fi Easy Connect. https://www.wi-fi.org/discover-wi-fi/wi-fi-easy-connect Accessed: 2025-01-06.