PKAP: Public Key Authentication Protocol

James Parker[*]

PKAuth, LLC

July 27, 2018

v0.0[†]

1 Introduction

This document defines Public Key Authentication Protocol (PKAP), a federated authentication protocol based on public key cryptography. PKAP provides 1) a standard for establishing cryptographic identities and 2) a protocol that authenticates users with web services via their cryptographic identities. PKAP's primary goal is to improve the security and usability of authentication.

To create a cryptographic identity, an individual generates public key pairs for each device and securely stores the corresponding private keys. The individual then generates a special key pair called the master key, which is used to digitally sign the set of device public keys. To share the newly created cryptographic identity, the individual hosts the signed set of public keys online. To authenticate an individual, web services only need to know the individual's master public key and the URL of the signed set. The individual can log in by digitally signing a challenge on any device included in the signed set.

Cryptographic identities can delegate access to other (child) cryptographic identities by referencing the child's URL and master public key. This enables groups or organizations to grant access to its members. For example, an employer may want to authorize its employees to unlock an office's PKAP enabled smart lock. The employer creates its cryptographic identity with references to the authorized employees' cryptographic identities. Now any authorized employee can authenticate with the smart lock and unlock the office door. If an employee retires, revocation is simple. The employer removes the retiree from the organization's cryptographic identity, and the retiree no longer has access.

PKAP has many practical applications in addition to allowing users to authenticate to websites using public key cryptography. Client-side applications on mobile and desktop can authenticate their users with PKAP. PKAP can be used alongside the Internet of Things (IoT) to provide access to network resources for groups of users. PKAP's signed sets contain encryption keys as well, so contacts can securely communicate via end-to-end encryption.

The rest of this document describes PKAP's standards and protocol in detail. Section 2 defines a standardized encoding format for the cryptographic algorithms used. Section 3 defines how cryptographic identities are encoded. Section 4 details how PKAP clients and servers interact to authenticate users and share cryptographic identities.

1.1 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 [2].

[*]`jp@pkauth.com`

[†]Working draft

Symmetric Authentication Algorithm	Key Length (bits)	Algorithm Identifier
HMAC with SHA256	256	sa-hmacsha256

Table 1: Summary of symmetric authentication algorithms.

2 Algorithms

PKAP uses various cryptographic algorithms to perform operations including encryption and authentication. This section describes the algorithms supported, as well as standards to perform tasks like serialization. Multiple precision integers, `mpint`, are used as defined in RFC 4251 [12]. Base64Url, `base64url`, is used to serialize binary data as defined in RFC 4648 [7]. PKAP Standard Form (PSF) is a standardized serialization format for various objects like public keys, private keys, and digital signatures. PSF will be defined for various objects throughout this section.

```
type PKAAlgorithm = String
```

To distinguish different cryptographic algorithms, PKAP defines unique algorithm identifiers throughout this section. These unique identifiers have type `PKAAlgorithm`, which is a type alias for strings.

```
type PKAIdentifier = String

toIdentifier( data : Data) : PKAIdentifier =
    let digest = ripeMD160( sha256( data)) in
    let checksum = firstFourBytes( sha256( sha256( digest))) in
    return base58( digest || checksum)
```

There are times when a cryptographic key needs to be identified without revealing the key itself. This identifier has type `PKAIdentifier`, which is a type alias for strings. To help construct identifiers, we define the function ‘toIdentifier‘ above. Functions `ripeMD160` and `sha256` respectively are the RIPEMD160 and SHA256 hash algorithms [3, 4]. The function `firstFourBytes` take the first four bytes of input and drops the rest. The function `base58` encodes binary data using the bitcoin alphabet [1]. Concatenation is represented by `||`. `PKAIdentifier`’s are similar to Bitcoin addresses, and they have checksums to help determine whether an identifier is valid.

2.1 Symmetric Authentication

Symmetric authentication is used to verify the integrity of data using a secret key. Table 1 summarizes the supported algorithms.

```
type PKAMacKey = {
    "key" : MacKey
  , "algorithm" : PKAAlgorithm
}
```

`PKAMacKey` is the record type used to encode symmetric authentication secret keys. The `algorithm` key states which symmetric authentication algorithm the key corresponds to. The `key` key is the Base64Url encoding of the PSF encoded secret key, where the PSF encoding is dependent on the algorithm used.

```
type PKAMac a = {
    "content" : a
  , "tag" : MacTag
  , "algorithm" : PKAAlgorithm
  , "identifier" : PKAIdentifier
}
```

`PKAMac a` is the record type used to encode symmetrically authenticated messages. The key `content` holds the Base64Url encoded message. Any content of type `a` can be authenticated as long as `a` is encodable

```
{
    "key":"qzzq7ZEpLCy8akbLuLSpkk0dmVgS_PwMHEQFXZe005Y=",
    "algorithm":"sa-hmacsha256"
}
```

Figure 1: JSON encoded `PKAMacKey`.

```
{
    "tag":"CHGpaQxpbsJR5dipozuDPqtaz1Vt3T61MNkoRulRalA=",
    "identifier":"N2zo2ur8NNbVcT1hpXmSps3vaxUxwuPBp",
    "content":"eyJ1c2VybmFtZSI6ImpwIiwic2VydmljZV9pZ...",
    "algorithm":"sa-hmacsha256"
}
```

Figure 2: JSON encoded `PKAMac PKAChallenge`.

in binary. Unless otherwise specified, this specification's default binary encoding is JSON. The `tag` key is the Base64Url encoding of the PSF encoded MAC tag, where the PSF encoding is dependent on the algorithm used. The key `algorithm` is the identifier of the symmetric authentication algorithm used. The key `identifier` is the unique identifier of the secret key used to authenticate the message.

When verifying a `PKAMac`, one MUST verify that the algorithm of the secret key matches the algorithm in the `PKAMac`, the secret key's identifier matches the identifier in the `PKAMac`, and the tag is valid.

JP: Should the identifier be optional? Is it needed/useful?

2.1.1 HMAC with SHA256

HMAC with SHA256 is a cryptographic algorithm used for symmetric authentication [10]. The algorithm identifier (`PKAAlgorithm`) for HMAC with SHA256 is `"sa-hmacsha256"`. The PSF encoding for HMAC with SHA256 secret keys (`MacKey`) is simply the 256 bit binary secret key. The PSF encoding for HMAC with SHA256 authentication tags (`MacTag`) is the 256 bit binary SHA256 output. The secret key identifier (`PKAIdentifier`) for HMAC with SHA256 is the output of `toIdentifier` where the input is the PSF encoded secret key.

An example JSON encoded `PKAMacKey` is shown in Figure 1. An example JSON encoded `PKAMac` is shown in Figure 2. The authenticated content has type `PKAChallenge`, which is defined in later sections.

2.2 Symmetric Encryption

Symmetric encryption is used to encrypt data using a secret key. Table 2 summarizes the symmetric encryption algorithms supported by PKAP.

```
type PKAEncSecretKey = {
    "key" : EncSecretKey
  , "algorithm": PKAAlgorithm
}
```

`PKAEncSecretKey` is the record type used to encode symmetric encryption secret keys. The key `algorithm` states which symmetric encryption algorithm the secret key corresponds to. The `key` key is the Base64Url

Symmetric Encryption Algorithm	Key Length (bits)	Algorithm Identifier
AES GCM	256	se-aesgcm256

Table 2: Summary of symmetric encryption algorithms.

Asymmetric Authentication Algorithm	Public Key Length (bits)	Algorithm Identifier
ED25519	256	aa-ed25519
RSA PSS with SHA256	2048	aa-rsa2048pss256

Table 3: Summary of asymmetric authentication algorithms.

encoding of the PSF encoded secret key, where the PSF encoding is dependent of the algorithm used.

```
type PKASymEncrypted a = {
    "ciphertext" : SymEncrypted a
  , "identifier" : PKAIdentifier
  , "algorithm" : PKAAlgorithm
}
```

`PKASymEncrypted` is the record type used to encode data encrypted using symmetric encryption. The `ciphertext` key holds the Base64Url encoding of the PSF encoded cipthertext. Any content of type a can be encrypted as long as `a` is encodable in binary. Unless otherwise specified, this specification's default binary encoding is JSON. The key `algorithm` states which symmetric encryption algorithm was used. The key `identifier` is the unique identifier of the secret key used to encrypt the content.

When decrypting a `PKASymEncrypted`, one MUST verify that the algorithm in the secret key, `PKAEncSecretKey`, matches the algorithm in the `PKASymEncrypted a` and the key's identifier matches the identifier in the `PKASymEncrypted a`.

Each algorithm has a standard key derivation function (`PKAKDF`) that produces a symmetric encryption key with inputs of a password, salt, and number of iterations.

2.2.1 AES GCM (256)

AES GCM is a cryptographic algorithm used for symmetric encryption [5]. Its algorithm identifier (`PKAAlgorithm`) is `"se-aesgcm256"`. The PSF encoding for AES GCM secret keys (`EncSecretKey`) is the 256 bit binary secret key. The PSF encoding for AES GCM ciphertexts (`SymEncrypted a`) is the concatenation of a 96 bit initialization vector (IV), the GCM ciphertext, and a 128 bit tag. The secret key identifier (`PKAIdentifier`) for AES GCM is the output of `toIdentifier` where the input is the PSF encoded secret key.

The standard `PKAKDF` for AES GCM is PBKDF2 with SHA256 [9].

2.3 Asymmetric Authentication

Asymmetric authentication algorithms are used to digitally sign data. Table 3 summarizes the asymmetric authentication algorithms supported by PKAP.

```
type PKAAuthPublicKey = {
    "public_key" : AuthPublicKey
  , "algorithm" : PKAAlgorithm
}
```

`PKAAuthPublicKey` is the record type used to encode asymmetric authentication public keys. The `algorithm` key states which asymmetric authentication algorithm the key corresponds to. The `public_key` key is the Base64Url encoding of the PSF encoded public key, where the PSF encoding is dependent on the algorithm used.

```
type PKAAuthPrivateKey = {
    "private_key" : AuthPrivateKey
  , "algorithm" : PKAAlgorithm
}
```

`PKAAuthPrivateKey` is the record type used to encode asymmetric authentication private keys. The `algorithm` key states which asymmetric authentication algorithm the key corresponds to. The `private_key`

```
{
    "algorithm":"aa-ed25519",
    "signature":"GicCpubKVNbbVqOng9_g2_dd-...",
    "identifier":"54tNxiGsBYMWCjb3rqUUvyTYk4mA62UXS",
    "content":"eyJ0YWciOiJDSEdwYVF4cGJz..."
}
```

Figure 3: JSON encoded `PKASigned (PKAMac PKAChallenge)`.

key is the Base64Url encoding of the PSF encoded private key, where the PSF encoding is dependent on the algorithm used.

```
type PKASigned a = {
    "content" : a
  , "signature" : Signature
  , "algorithm" : PKAAlgorithm
  , "identifier" : PKAIdentifier
}
```

`PKASigned a` is the record type used to encode asymmetrically authenticated messages. The `content` key holds the Base64Url encoded message. Any content of type `a` can be digitally signed as long as `a` is encodable in binary. Unless otherwise specified, this specification's default binary encoding is JSON. The key `signature` is the Base64Url encoding of the PSF encoded digital signature, which is dependent on the algorithm used. The `algorithm` key states which asymmetric authentication algorithm the key is for. The key `identifier` is the unique identifier of the key pair used to sign the content (Corresponding public and private keys always share the same identifier).

When verifying a `PKASigned a`, one MUST verify that the algorithm of the public key matches the algorithm in the `PKASigned a`, the key's identifier matches the identifier in the `PKAMac`, and the signature is valid.

2.3.1 ED25519

ED25519 is one of the cryptographic algorithms used for asymmetric authentication [8]. Its algorithm identifer (`PKAAlgorithm`) is `"aa-ed25519"`. The PSF encoding for ED25519 public keys (`AuthPublicKey`) is `A`. The PSF encoding for ED25519 private keys (`AuthPrivateKey`) is `k` concatenated with `A`. Both `k` and `A` are 256 bits each. The identifier (`PKAIdentifier`) for both the public key and the private key is the output of `toIdentifier` where the input is the PSF encoded public key. The PSF encoded signature (`Signature`) for ED25519 is the 256 bit `R` concatenated with the 256 bit `S`.

2.3.2 RSA PSS with SHA256

Note: May drop RSA. Not currently in Rust implementation.

RSA PSS with SHA256 is a cryptographic algorithm used for asymmetric authentication [6]. Its algorithm identifier (`PKAAlgorithm`) is `"aa-rsa2048pss256"`. The PSF encoding for RSA public keys (`AuthPublicKey`) is the `mpint` of `n` and `e`:

```
mpint public_n
mpint public_e
```

The PSF encoding for RSA private keys (`AuthPrivateKey`) is the `mpint` of `n`, `e`, `d`, `p`, `q`, `dP`, `dQ`, `qInv`:

```
mpint public_n
mpint public_e
mpint private_d
mpint private_p
mpint private_q
```

Asymmetric Encryption Algorithm	Public Key Length (bits)	Algorithm Identifier
RSA OAEP with SHA256 and MGF1	2048	ae-rsa2048oaep256
X25519	256	ae-x25519

Table 4: Summary of asymmetric encryption algorithms.

```
mpint private_dP
mpint private_dQ
mpint private_qInv
```

The identifier (`PKAIdentifier`) for both the public key and the private key is the output of `toIdentifier` where the input is the PSF encoded public key. The PSF encoded signature (`Signature`) for RSA PSS is the digital signature generated using SHA256, MGF1, and a `0xbc` trailer field.

2.4 Asymmetric Encryption

Asymmetric encryption algorithms are used to encrypt messages. Table 4 summarizes the asymmetric encryption algorithms supported by PKAP.

```
type PKAEncPublicKey = {
    "public_key" : EncPublicKey
  , "algorithm" : PKAAlgorithm
}
```

`PKAEncPublicKey` is the record type used to encode asymmetric encryption public keys. The key `algorithm` states which asymmetric encryption algorithm the public key corresponds to. The `public_key` key is the Base64Url encoding of the PSF encoded public key, where the PSF encoding is dependent on the algorithm used.

```
type PKAEncPrivateKey = {
    "private_key" : EncPrivateKey
  , "algorithm" : PKAAlgorithm
}
```

`PKAEncPrivateKey` is the record type used to encode asymmetric encryption private keys. The key `algorithm` states which asymmetric encryption algorithm the private key corresponds to. The `private_key` key is the Base64Url encoding of the PSF encoded private key, where the PSF encoding is dependent on the algorithm used.

```
type PKAAsymEncrypted512 = {
    "ciphertext" : AsymEncrypted512
  , "algorithm" : PKAAlgorithm
}

type PKAAsymEncrypted a = {
    "keys" : {PKAIdentifier : PKAAsymEncrypted512}
    "ciphertext" : PKASymEncrypted a
}
```

JP: A single "key" option without identifiers?

Asymmetric encryption is typically limited in the length of messages which can be encrypted. Therefore, PKAP generates a symmetric secret key which is used to encrypt the message. The symmetric secret key is created by randomly generating a 512 bit seed and passing the seed to the respective `PKAKDF` as the password (the salt is the empty string and the number of iterations is one). Then the 512 bit seed is encrypted with each of the asymmetric public keys.

`PKAAsymEncrypted512` is the record type used to encode the 512 asymmetrically encrypted bits. The `ciphertext` key is the Base64Url encoding of the PSF encoded ciphertext, which is dependent of the algorithm used. The key `algorithm` states which asymmetric encryption algorithm was used.

`PKAAsymEncrypted a` is the record type that encodes the message ciphertext and the dictionary of encrypted symmetric secret keys. The `ciphertext` key contains the symmetrically encrypted message with type `PKASymEncrypted a` (Section 2.2). Any content of type `a` can be encrypted as long as `a` is encodable in binary. Unless otherwise specified, this specification's default binary encoding is JSON. The `keys` key contains a dictionary of public key identifiers to asymmetrically encrypted 512 bit seeds ({`PKAIdentifier : PKAAsymEncrypted512`}).

When decrypting a `PKAAsymEncrypted512`, one MUST verify that the algorithm of the public key matches the `algorithm` and the key's identifier matches the identifier from `keys`.

2.4.1 X25519

X25519 is a cryptographic algorithm used for asymmetric encryption [11]. Its algorithm identifier (`PKAAlgorithm`) is `"ae-x25519"`. The PSF encoding for X25519 public keys (`EncPublicKey`) and private keys (`EncPrivateKey`) are both 256 bits. The identifier (`PKAIdentifier`) for both the public key and the private key is the output of `toIdentifier` where the input is the PSF encoded public key. The 512 bit seed is asymmetrically encrypted by generating an ephemeral key pair and performing Diffie-Hellman key exchange to generate a shared secret. Take the SHA512 of the shared secret and xor it with the 512 bit seed. The PSF encoded ciphertext (`AsymEncrypted512`) for X25519 is 256 bits of the ephemeral public key concatenated with 512 bits of the encrypted seed.

2.4.2 RSA OAEP with SHA256 and MGF1

Note: May drop RSA. Not currently in Rust implementation.

RSA OAEP with SHA256 and MGF1 is a cryptographic algorithm used for asymmetric encryption [6]. Its algorithm identifier (`PKAAlgorithm`) is `"ae-rsa2048oaep256"`. The PSF encodings for RSA public keys (`EncPublicKey`), private keys (`EncPublicKey`), and the identifier (`PKAIdentifier`) for both public and private keys are the same as previously defined in Section 2.3.2. The PSF encoding for ciphertexts (`Encrypted a`) is the encrypted 512 bit seed.

3 Cryptographic Identities

PKAP cryptographic identities distribute keys using a recursive, self-signing public key infrastructure. Specifically, a recursive tree structure (with type `PKASigned PKATree`) grants access to child entities and provides the public keys of entities in the tree. These public keys are used for authentication, encryption, and digital signatures. The recursive tree structure is encoded in JSON with type `PKASigned PKATree` in a file (with extension `.pkt` by convention). Identities are self-signing since each node has a master key which is used to sign that node.

The definition of type `PKATree` is shown in Figure 4. `PKATree` is a record type that consists of keys `authentication`, `signature`, `encryption`, `children`, `master`, `ttl`, `expiration`, and `updated`. `authentication` indicates the set of public keys that can be used for authentication. `signature` is the set of public keys that can be used to verify digital signatures. `encryption` is the set of public keys that can be used for encryption. `children` indicates a reference to child nodes of the current node along with their meta data. `master` is the current node's master key. `ttl` indicates the *time to live* in seconds. `expiration` is an optional date-time field of when the current node expires. `updated` is the date-time when the current node was last updated or created.

Type `PKAChild` is a record that describes meta data about a child node. `key` is the master key of the child node. `location` is URL reference to the child node's JSON encoded `PKASigned PKATree` (or `.pkt` file). `roles` grants additional privileges to the child node. `expiration` is an optional date-time field of when delegation to the child expires. `depth` is an optional attribute that indicates the maximum depth of the child's tree to limit further delegation. For instance, a `depth` of 0 means that the child node cannot have children, while a `depth` of 1 means the child node can have children, but no grandchildren.

Type `PKARole` is an enum of strings. The only currently valid strings are `"admin"`, `"write"`, and `"read"`.

Type `URL a` is a URL string that references a JSON encoded object of type `a`.

An example of a JSON encoded `PKATree` is in Figure 5.

```
type PKATree = {
    "authentication" : [PKAAuthPublicKey]
  , "signature" : Maybe [PKAAuthPublicKey]
  , "encryption" : Maybe [PKAEncKey]
  , "master" : PKAAuthPublicKey
  , "ttl" : Int
  , "expiration" : Maybe DateTime
  , "updated" : DateTime
  , "children" : Maybe [PKAChild]
}

type PKAChild = {
    "key" : PKAAuthPublicKey
  , "location" : URL (PKASigned PKATree)
  , "roles" : [PKARole]
  , "expiration" : Maybe DateTime
  , "depth" : Maybe Int
}

type PKARole = "admin" | "write" | "read"

type URL a = String
```

Figure 4: Definition of type `PKATree`.

When parsing a JSON encode `PKASigned PKATree`, there are a few constraints that MUST be checked. The digital signature of `PKASigned PKATree` MUST be cryptographically verified. The public key used to sign the `PKASigned PKATree` MUST equal the `master` public key in `PKATree`. For all children in a node, the `key` MUST equal the public key used to sign the child's `PKASigned PKATree` at the child's `location`. A node's *computed roles* MUST be the set intersect of the roles of all of the node's ancestors. A node's *computed expiration* MUST be the earliest expiration of all of the node's ancestors. A node's *computed updated time* MUST be the latest updated time of all of the node's ancestors. If `depth` is present in any of a node's ancestors, the node's *computed remaining depth* MUST decrement for every subsequent recursive layer in the tree, and it MUST be the minimum of the current remaining depth and the current node's `depth`.

Type `PKALink` (Figure 6) contains an entity's master public key (`master_key`) and a URL reference to the entity's `.pkt` file (`location`). By convention, a JSON encoded `PKALink` file has extension `.pkl`.

4 PKAP API

PKAP is a public key authentication protocol used to authenticate end users. The protocol requires that the client and service MUST establish a secure channel where the service's identity is authenticated. It also requires that the service's internal clock time MUST be monotonic and have integrity. In addition, the client and service MUST agree upon a unique `username` that identifies the client.

4.1 PKAP for Websites

While the PKAP authentication protocol can be used over any secure channel, special considerations must be taken into account when authenticating over `https` in the browser. In particular, PKAP compliant websites

```
{
  "authentication": [{
    "public_key": "_JJ9LujZL7TjIUCoTQFposnQ2t6eozeAoW9A-LjFzAw=",
    "algorithm": "aa-ed25519"
  }],
  "signature": [],
  "encryption": [],
  "children": [{
    "key": {
      "public_key": "AAABAQCoe6rDgIkHks48j9-TottFftZ...",
      "algorithm": "aa-rsa2048pss256"
    },
    "location": "http://domain.name/name.pkt",
    "roles": ["admin"],
    "depth": 1
  }],
  "master": {
    "public_key": "7sqbfLVUo9Atr13rrWu5zdkeenlosdHNMJcNdS83V1w=",
    "algorithm": "aa-ed25519"
  },
  "ttl": 3600,
  "expiration": "2017-01-01T01:00:00.000Z",
  "updated": "2015-01-01T01:00:00.000Z"
}
```

Figure 5: A JSON encoded `PKATree`.

```
type PKALink = {
    "location": URL (PKASigned PKATree)
  , "master_key": PKAAuthPublicKey
}
```

Figure 6: Definition of type `PKALink`.

MUST embed the html header tag `pkap`. The `pkap` tag MUST always contain the attributes `href` and `token`. The `href` attribute MUST point to a URL on the same domain of the current webpage that will handle PKAP requests. This URL is referred to as `authUrl`. The `token` attribute MUST be a random sequence of characters to prevent CSRF attacks. If the user is not logged into the website, `href` and `token` are the only required attributes. If the user is logged into the website, the `authenticated` attribute MUST be included where the value is the user's username. If the website knows the user's master public key and `PKASigned PKATree` URL, the `pkinfo` attribute MUST be included where the value is the identifier (`PKAIdentifier`) of the user's master public key. Examples of `pkap` tags are shown in Figure 7.

```
<pkap href="https://domain.name/pkap" token="95eYX2IN" />
<pkap href="https://domain.name/pkap" token="95eYX2IN"
        authenticated="user63" />
<pkap href="https://domain.name/pkap" token="95eYX2IN"
        authenticated="user63" pkinfo="MiD7NiyZrhj4qF69C..." />
```

Figure 7: Example `pkap` html header tags.

```
verb: initiate
username: user63
identifier_pk: 14E2FED8E57F47BCC4D70A90EFCD765C97040F5A
tree_path: ["http://domain.name/child.pkt"]
```

Figure 8: Initiating authentication POST request.

```
type DomainName = String
type Nonce = String

type PKAChallenge = {
    "username": String
  , "public_key": PKAAuthPublicKey
  , "timestamp": UTCTime
  , "service_identifier": DomainName
  , "nonce": Nonce
  , "challenge": Maybe ExternalChallenge
}
```

Figure 9: Definition of a `PKAChallenge`.

4.1.1 Login

When initiating an authentication request, the client first verifies that the domain of the `authUrl` is the same as the current website's. It then sends a POST request to the `authUrl` where the `verb` key has value `initiate`, the `username` is the user's username, and the key `identifier_pk` is the `PKAIdentifier` of the public key to be used for authentication. The key `tree_path` is an array of `PKASigned PKATree` URLs that indicate the path of children in the PKATree to reach the public key to be used for authentication. The `tree_path` MUST not include the root `PKASigned PKATree` URL, and the URLs in the `tree_path` MUST equal the URL locations in the tree. An example request is shown in Figure 8.

When the service receives the initiate request, it MUST check that the username exists and that the user's public key set contains a public key that matches the identifier. It is a failure if the public key is not in the `tree_path` of the PKATree. The service MUST generate a uniformly random nonce with a length of at least 64 bits. The service MUST also get the current time. Using this information, the service MUST create a `PKAChallenge` (defined in Figure 9 and a JSON encoded example is in Figure 11). The service MUST then authenticate the challenge using a MAC, and the MAC key used MAY be unique for each user. The service MUST return the JSON encoded `PKAMac PKAChallenge` in a `200 Success` response. An example of a JSON encoded `PKAMac PKAChallenge` is shown in Figure 2. For added security, the service MAY store the nonce internally to protect against replay attacks that occur before the challenge expires. On any failure, the service MUST return a `400 Bad Request` response with content of a JSON encoded `PKAError` (Figure 10).

When the client receives the challenge, it MUST check that the `username` and `identifier_pk` in the challenge match what was in the origin request. The client MUST also check that the `service_identifier` matches the service's domain name and that the `timestamp` is within a relatively recent period of time. By default this period is two minutes. If any of these checks fail, the client MAY restart the authentication process by sending another `initiate` request or MAY stop the authentication attempt. Upon success, the client MAY continue the process by digitally signing the `PKAMac PKAChallenge`, producing a `PKASigned (PKAMac PKAChallenge)`. The client provides the service the digitally signed challenge by sending a POST request to the `authUrl` where key `verb` has value `authenticate`, key `token` is the token provided by the pkap header, and key `challenge` is the JSON encoded `PKASigned (PKAMac PKAChallenge)`. An example of a JSON encoded `PKASigned (PKAMac PKAChallenge)` is shown in Figure 3. An example of the authenticating POST request is shown in Figure 12.

When the service receives the `authenticate` request, it MUST check that the username exists, that the

```
type PKAError = {
    "error": ErrorCode
  , "success": Bool
}
```

`ErrorCode`	Description
0	General error
1	Invalid token
2	Invalid verb
3	Invalid parameters (missing or improperly encoded parameters)
4	Cryptographic identity expired
5	Could not verify identity
6	Invalid identity (no cryptographic identity for user or supplied public key is not valid)
7	Challenge expired
8	Invalid challenge (nonce is wrong, domain name is wrong, signed challenge is not verified)
9	Rate limit timeout

Figure 10: Definition of a `PKAError`. `ErrorCode` is an integer error code that details what error occurred. The `success` key is a boolean set to `false`.

```
{
    "username":"user63",
    "service_identifier":"domain.name",
    "public_key": {
        "algorithm":"aa-ed25519",
        "public_key":"YSwAWjrhYF0Q4QTEZcZczgVd5K..."
    },
    "timestamp":"2016-02-18T18:03:47.813Z",
    "nonce":"T0AIBo5flVq6FvP0jCKNPClWEWa71..."
}
```

Figure 11: JSON encoded `PKAChallenge`.

```
verb: authenticate
token: 95eYX2IN
challenge: {
    "algorithm":"aa-ed25519",
    "signature":"GicCpubKVNbbVqOng9_g2_dd-...",
    "identifier":"54tNxiGsBYMWCjb3rqUUvyTYk4mA62UXS",
    "content":"eyJ0YWciOiJDSEdwYVF4cGJz..."
}
```

Figure 12: An example authenticating POST request.

```
type PKAResponse = {
    "success" : Bool
  , "redirect" : Maybe URL
  , "extra" : Maybe JSON
}
```

Figure 13: Definition of a `PKAResponse`.

```
{
    "success" : true,
    "redirect" : "https://domain.name"
}
```

Figure 14: A successful AJAX response indicating a browser redirect.

`token` key is the token provided by the PKAP html header tag, and that the public key used to sign the challenge belongs to the user being authenticated. The service MUST also check that the `service_identifier` matches the service's domain name. The service MUST check that the `timestamp` is within a relatively recent period of time. By default this period is two minutes. If the service previously stored nonce internally, the service MUST check that the nonce for that user is equal, and it MUST revoke the given nonce. The service MUST verify the MAC integrity of the challenge (`PKAMac PKAChallenge`). The service MUST verify the integrity of the digitally signed challenge (`PKASigned (PKAMac PKAChallenge)`). On any failure, the service MUST return a `400 Bad Request` response with content of a JSON encoded `PKAError` (Figure 10). Otherwise, the client's identity is now confirmed so the service can authorize the user as usual. Typically, this involves setting browser cookies and redirecting the client. The service MAY redirect the client by returning content of a JSON encoded `PKAResponse`.

The definition of type `PKAResponse` is shown in Figure 13. `PKAResponse` is a record type that consists of keys `success`, `redirect`, and `extra`. `success` is a boolean that indicates the success of a request. `redirect` is an optional URL to redirect the client to. `extra` is an optional field of any JSON value that may be application specific. An example with a redirect field is shown in Figure 14.

4.1.2 Logout

When the client wishes to logout, it checks the domain of the `authUrl`. Then the client sends the service an AJAX POST request where the `verb` key is `logout` and the `token` key is the token provided by the `pkap` html header tag. Figure 15 shows an example of this. After verifying the token, the service can log out the client as usual. Typically the service does this by revoking browser cookies and redirecting. The service MAY return content of type `PKAResponse` (Figure 13).

4.1.3 Establish Public Key Information

When the user is logged in and the service does not know the user's master public key, the `pkap` tag includes the `pkinfo` attribute. In this circumstance, the client MAY establish the user's master public key and `PKASigned PKATree` URL with the service. The client does this by first verifying the domain of `authUrl`. It then sends an AJAX POST request to `authUrl` where the `verb` key is the constant `pkinfo`, the `token` key is the CSRF token, the `username` key is the user's username, the `pkurl` key is the user's `PKASigned PKATree`

```
verb: logout
token: 95eYX2IN
```

Figure 15: An example of a logout AJAX POST request.

```
verb: pkinfo
token: 95eYX2IN
username: user63
pkurl: http://domain.name/name.pkt
pkmaster: {
  "public_key": "AAABAQCoe6rDgIkHks48j9-TottFftZ...",
  "algorithm": "aa-rsa2048pss256"
}
```

Figure 16: An example AJAX POST request to establish a user's public key information.

URL, and the `pkmaster` key is the user's master public key. An example is shown in Figure 16. On any failure, the service MAY return a `400 Bad Request` response with content of a JSON encoded `PKAError` (Figure 10).

The service MUST handle the `pkinfo` verb to associate the received public key information with the given user, and it MUST verify the CSRF token. The service SHOULD take precautions when authorizing public key information to access a user's account (similar to those taken when users edit their passwords). For example, confirmation emails could be sent to users or users could be redirected to a page where users needs to enter their passwords to approve the new public key information. The service MAY redirect the client using JSON as shown previously in Figure 14. The public key information MUST NOT be transported via GET variables.

References

[1] Bitcoin Wiki. *Base58Check encoding*, 2017 [accessed May 15, 2018]. `https://en.bitcoin.it/wiki/Base58Check_encoding#Base58_symbol_chart`.

[2] Bradner, S. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[3] Dobbertin, H., Bosselaers, A., and Preneel, B. Ripemd-160: A strengthened version of ripemd. In *Fast Software Encryption* (Berlin, Heidelberg, 1996), D. Gollmann, Ed., Springer Berlin Heidelberg, pp. 71–82.

[4] Donald, 3rd, E., and Hansen, T. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634, July 2006.

[5] Dworkin, M. J. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Tech. rep., Gaithersburg, MD, United States, 2007.

[6] Jonsson, J., and Kaliski, B. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1. RFC 3447, February 2003.

[7] Josefsson, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.

[8] Josefsson, S., and Liusvaara, I. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[9] Kaliski, B. Pkcs #5: Password-based cryptography specification. RFC 2898, September 2000.

[10] Kelly, S., and Frankel, S. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. RFC 4868, May 2007.

[11] Langley, A., Hamburg, M., and Turner, S. Elliptic curves for security. RFC 7748, January 2016.

[12] Ylonen, T. The Secure Shell (SSH) Protocol Architecture. RFC 4251, January 2006.