# OpenPubkey: Augmenting OpenID Connect with User held Signing Keys

*Ethan Heilman* *Lucie Mugnier* *Athanasios Filippidis* *Sharon Goldberg*
*Sebastien Lipman* *Yuval Marcus* *Mike Milano* *Sidhartha Premkumar*
*Chad Unrein*
*BastionZero, Inc.*

## Abstract

OpenPubkey makes a client-side modification to OpenID Connect so that an ID Token issued by an OpenID Provider commits to a user held public key, $PK_u$. This transforms an ID Token into a certificate that cryptographically binds an OpenID Connect identity to a public key. We call such an ID Token a *PK Token*. The user can then sign messages with their signing key, $SK_u$, and these signatures can be authenticated and attributed to the user's OpenID Connect identity. This allows OpenPubkey to upgrade OpenID Connect from *Bearer Authentication* to *Proof-of-Possession*, eliminating trust assumptions in OpenID Connect and defeating entire categories of attacks present in OpenID Connect. OpenPubkey was designed to satisfy a decade-long need for this functionality. Prior to OpenPubkey, OpenID Connect did not have a secure way for users to sign statements under their OpenID identities.

OpenPubkey is transparent to users and OpenID Providers. An OpenID Provider can not even determine that OpenPubkey is being used. This makes OpenPubkey fully compatible with existing OpenID Providers. In fact a variant of OpenPubkey is currently deployed and used to authenticate signed messages and identities for users with accounts on Google, Microsoft, Okta, and Onelogin. OpenPubkey does not add new trusted parties to OpenID Connect and reduces preexisting trust assumptions. If used in tandem with our MFA-cosigner, OpenPubkey can maintain security even against a malicious OpenID Provider (the most trusted party in OpenID Connect).

## 1 Introduction

OpenID Connect [35] is the dominant Single Sign On (SSO) identity authentication protocol on the web. It is supported by all major identity providers including Google, Microsoft, Facebook, and Okta. With OpenID Connect, users sign into their OpenID Provider (OP) *e.g.,* Google. Once signed in, the user can authenticate with other entities by having their OP attest to their identity. This Single Sign On (SSO) functionality provides substantial convenience and security benefits to users, since by using OpenID Connect a user need only manage the one set of credentials at their OP, rather than having to manage a plethora of accounts and passwords at every site they use.

**Bearer Authentication** In OpenID Connect when a user wants to authenticate their identity to an *audience i.e.,* a website or other entity that supports OpenID Connect, the *audience* requests that the user's software, called the *Client Instance*, supply an ID Token. This ID Token is a cryptographically verifiable attestation, from the OpenID Provider (OP) to the audience, of the user's identity. The audience checks the OP's signature on the ID Token and grants the user access to the account corresponding to the identity attested by the ID Token. This form of authentication is called *Bearer Authentication*.

Bearer Authentication is when a user authenticates to a party by "bearing", that is revealing to that party, the user's authentication secret. In OpenID Connect, the secret revealed is the ID Token. Anyone "bearing" the ID token can authenticate as the identity attested to in the ID token.

The main benefit of Bearer Authentication is in its simplicity: you just reveal a secret in order to authenticate. This simplicity comes at the cost of inherent security vulnerabilities resulting from the necessity of exposing the authentication secret (the ID token) to authenticate. Thus, a server at an audience receiving an ID Token, or any party intercepting the ID Token, can replay that token to other servers at that same audience and thereby impersonate the user. This is called a *Token Replay Attack*. OAuth2 and OpenID Connect tokens are also vulnerable to *Token Export Attacks* where a malicious, compromised or misconfigured Client Instance leaks tokens to an attacker enabling that attacker to impersonate the user.

The security implications of *Token Replay Attacks* (Section 4.1) and *Token Export Attacks* (Section 4.2) have motivated a decade-long standardization effort (Section 5) to reduce these vulnerabilities. These efforts aim to upgrade OAuth2 and OpenID Connect to a more secure authentication archetype: Proof-of-Possession.

**Proof-of-Possession:** Instead of revealing the authentica-

tion secret, in Proof-of-Possession the user just proves that they *hold* the authentication secret. Proof-of-Possession typically uses digital signatures [11] where the authentication secret is a signing key and the user signs a random challenge from an audience, to prove they "possess" the signing key. Proof-of-Possession prevents Token Replay Attacks because the user no longer exposes their authentication secret to authenticate. Proof-of-Possession also addresses Token Export Attacks as it enables the use of non-extractable keys (see Section 4.2), where software is trusted to request signatures but is not trusted to read or "extract" the signing key.

OpenPubkey makes a client-side modification to OpenID Connect so that an ID Token issued by an OP commits to a public key, $PK_u$, held by the user. We call an ID Token generated in this way, a *PK Token*. The user can then produce and sign messages with their signing key, $SK_u$, and these signatures can be authenticated and attributed to the user's identity via the PK Token. This enables us to move OpenID Connect from Bearer Authentication to Proof-of-Possession. Using our new signing capability we create OpenPubkey Signed Messages (OSM) which are JSON Web Signatures which can be publicly verified, authenticated and attributed to the user identity attested to in the ID Token.

OpenPubkey's PK Token design is highly extensible and enables the use of *Cosigners i.e.,* third parties that provide additional validation of claims in the ID Token. In Section 3.4, we show how a cosigner that performs an independent MFA authentication of the user, allows OpenPubkey to maintain security even against a fully malicious OP. OpenID Connect, as currently used without a cosigner, suffers a complete loss of security if an OP becomes fully malicious (See Section 4.4). The use of cosigners does not require any changes at the OP and is not required for OpenPubkey.

OpenPubkey achieves this without adding trusted parties, without requiring any modifications to OpenID Providers (OP) and while reducing trust assumptions currently made by OpenID Connect. It is fully compatible with current OpenID Providers. To users, OpenPubkey looks just like OpenID Connect. No additional steps or actions are required. As user signing keys are ephemeral, we avoid the complexities and burdens of key management; Users don't need to transfer or manage keys. As discussed in Appendix A a variant of OpenPubkey is actively being used with ID Tokens issued by Google, Microsoft, Okta, and OneLogin.

Our paper is structured as follows: In Section 2.1, we provide background on JSON Web Signatures and JSON Web Key Sets. Section 2.2 introduces the parts of OpenID Connect needed for OpenPubkey. Section 3 describes OpenPubkey, which we follow with discussion of the security advantages of OpenPubkey (Section 4) and Related Work (Section 5).

## 1.1 Contributions

OpenPubkey's main contribution is achieving the following desiderata using simple changes that maintain compatibility with existing OPs, current user workflows and JSON Web cryptography standards:

- **Proof-of-Possession:** Users can prove to a verifier *e.g.,* an audience, that they hold a signing key associated with their OpenID identity.
- **Signed Messages:** Users can generate signed messages bound to the their identity.
- **Extensibility:** Developers can extend and add functionality to OpenPubkey without requiring changes to OPs or breaking backwards compatibility with OpenPubkey.
- **Trust minimization:** OpenPubkey must not add trusted parties to OpenID Connect and should minimize and eliminate preexisting trust assumptions.

An additional contribution of this work is the design of an MFA-Cosigner. This enables an entity other than the OP to provide an MFA authentication of the user such that the OP is no longer a Single Point of Compromise (SPoC).

A full manifest of the software projects and protocols that benefit from OpenPubkey is beyond the scope of this paper. Instead we provide two instructive examples. Zoom's End-to-End Encryption proposal [8] assumes functionality in OpenID Connect which does not exist in the OpenID Connect standard. This missing functionality is provided by OpenPubkey. SigStore [37] "Keyless Signatures" protocol requires trusting a party to map public keys and OpenID Connect identities, OpenPubkey could remove the trust placed in that party. For full details see Related Work (Section 5).

# 2 Background

## 2.1 JSON Web Signatures (JWS)

JSON Web Signatures (JWS) [20] are a standardized JSON-based signed message format. A JWS a bundle of objects including the message *aka,* the payload, a set of signatures and signature metadata. All the signatures in a JWS share the same payload but each has their own *protected header*.

```
{
  "payload": payload,
  "signatures":[
  {
    "protected": h₁ Protected header,
    "signature": σ₁ ← SIGN(SK₁,(payload,h₁))
  },
  {
    "protected": h₂ Protected header,
    "signature": σ₂ ← SIGN(SK₂,(payload,h₂))
```

```
    }]
}
```

Each signature is over both the payload and that signature's protected header.[1] As shown above, the signature, $\sigma_1$, is computed as $\texttt{SIGN}(SK_1, (\text{payload}, \text{h}_1))$, where $\text{h}_1$ denotes the protected header.

Protected headers include JOSE (JSON Object Signing and Encryption) [29] signature metadata parameters such as `kid` (Key ID) and `alg` (algorithm). `kid` is an identifier which is used to determine which key in a set of keys should be used to verify the signature. `alg` specifies the algorithm that must be used to verify the signature.

JSON Web Tokens (JWT) [21] are subtype of JSON Web Signatures (JWS)[2] in which the payload of the JWS is a set of standardized claims encoded in JSON *e.g.,* the claims `iss`, `sub`, and `aud`, identify the issuer, the subject and intended audience of the JWT respectively. JWTs is designed to enable an issuer to make authenticated claims about a subject, where these claims bound to the identity of the issuer.

JSON Web Key (JWK) [19] is a standardized JSON representation of a cryptographic key. In this paper we use public keys encoded as JWKs to verify digital signatures encoded as JWS. The JWK standard defines JSON Web Key Sets (JWKS), pronounced Jay-wicks, as sets of JWK keys.

Important to OAuth2, OpenID Connect and OpenPubkey is the notion of a JWKS endpoint *aka,* JWKS URI. A JWKS endpoint is a web service identified by a HTTPS URI which hosts a JWKS (JSON Web Key Set) containing the public keys for a particular entity[3]. Because HTTPS/TLS is used, the contents of the endpoint are authenticated. A party who is configured with the URI of the JWKS endpoint can download Google's public keys and use these public keys to verify digital signatures purporting to be signed by Google.

JWKS endpoints allow signing authorities in protocols like OpenID Connect to easily disseminate new signing keys, rotate old keys and rapidly recover from a compromise. Rotating signing keys is done by adding new keys and removing the old keys from the JWKS endpoint. Relying parties frequently update their JWKS caches and will remove the rotated out keys on their next download. A consequence of this design is that a compromise of a JWKS endpoint lets an attacker add their public key to the key set and thereby generate signatures under the identity associated with that JWKS endpoint.

## 2.2 OpenID Connect

In this section we provide the necessary background on OpenID Connect. We focus only on the subset of functionality used in OpenPubkey. This includes how ID Tokens are issued, used and what information they contain.

OpenID Connect (OIDC) [35] is a ubiquitous web identity protocol that built on OAuth2 [16]. OAuth2 allows users to delegate access and authorization to third parties via Access Tokens and Refresh Tokens. OpenID Connect extends OAuth2 by adding a new type of token, an ID Token. This allows users to prove their identity to entities called audiences. In this paper we follow the OpenID Connect terminology and refer to the party that authenticates users and issues ID tokens, Access Tokens and Refresh Tokens as the OpenID Provider (OP). This paper is specifically concerned with how an OP grants tokens using the PKCE (Proof Key for Code Exchange) Authorization Code Flow [9]. We do not consider other authorization flows in this paper. As Access Tokens are not used in OpenPubkey we exclude them from protocol descriptions.

In OpenID Connect there are five parties: the user, the audience, the OP, the Client Instance and OIDC-RP.

**The user** is the party who wants to prove their identity to an audience by revealing their ID token.

**The audience** is the party that to whom the user presents their ID Token.

**The Client Instance** is an OpenID Connect user-agent *i.e.,* software that acts on the user's behalf to enable the user to participate in the protocol. The Client Instance requests, receives, stores, sends and refreshes the user's ID Token.

**The OP (OpenID Provider)** is an identity provider, such as Google, Microsoft or Okta, who is trusted to authenticate the user and then sign and issue an ID Token which attests to a user's identity. This ID token is scoped to a particular set of audiences.

There is one other party that is relevant called the OIDC-RP (OpenID Connect Relying Party). While the RP stands for Relying Party, the purpose of this party is not exactly the same as a traditional relying party used in the context of other systems (*e.g.,* X.509 certificates). For clarity we recommend that the reader not think of the OIDC-RP as a traditional relying party.[4] Before explaining the role the OIDC-RP performs, we must define some other important aspects of OpenID Connect.

**The Client ID** is the identifier of the OpenID Connect SSO integration. For instance if the organization `example.org` wishes to let users sign into website `audience.com` using the Google OP, `example.org` would perform an SSO integration with the Google OpenID Provider and be granted a Client ID *e.g.,* `12345`.

**The Redirect-URI** is the URI that the Client Instance instructs the OP to send the Auth-Code after a user successfully authenticates to an OP. The Auth-Code is a secret generated by an OP that functions as proof that the user authenticated to the OP. The Client Instance receives the Auth-Code at the Redirect-URI, then sends it back to the OP with some other

[1] JWS objects can also have *headers* which unlike *protected headers* are not covered by the signature. In this paper we only use protected headers.

[2] JWTs can be JSON Web Encryptions (JWE) but that isn't relevant here.

[3] For instance Google's JWKS endpoint used for OAuth2 and OpenID Connect is: https://www.googleapis.com/oauth2/v3/certs.

[4] In OpenID Connect documentation the OIDC-RP is also sometimes called *the Client* and sometimes called the Relying Party.

secrets and in exchange is sent the user's ID Token and Refresh Token (See Section 2.2.4). What is important to note here is that we do not want malicious parties to be able to read what is sent to the Redirect-URI, because the redirect-URI is used communicate the security sensitive Auth-Code to the Client Instance.

OpenID Connect must solve two security problems here. First it must ensure that a malicious Client Instance can't read the Auth-Code sent to the user's Client Instance Redirect-URI. This is addressed by the enforcement of *Same-Origin Policy* [36] in web browsers. Same-Origin Policy prevents malicious javascript running on the website `evil.com` from reading cookies for `example.org` or in OpenID Connect's case preventing javascript at `evil.com` from reading a URI with the `example.org` domain. Second, because a Client Instance tells the OP what Redirect-URI to use, OpenID Connect must prevent an OP from using unauthorized, and potentially malicious, Redirect-URI that was specified by a malicious Client Instance. This is solved by setting up an ACL (Access Control List) at the OP that maps Client IDs to Redirect-URIs. For example, an OP's Redirect-URI ACL might contain an entry that says Client ID `12345` can only use `https://example.org/redir/auth`. This ensures that an ID Token issued for Client ID `12345` must use Redirect-URI `https://example.org/redir/auth`. As an ID Token always includes the Client ID that it were issued for, audiences can check that the ID Token was issued using a Redirect-URI authorized by Client ID they trust.

**The OIDC-RP (OpenID Connect Relying Party)** is the entity that is issued the Client ID and manages the corresponding entry at the Redirect-URI ACL at the OP. When an audience is trusting an Client ID, that they are really trusting is the OIDC-RP that controls that Client ID.

#### 2.2.1 ID Tokens

The ID Tokens are JWTs (JSON Web Tokens) [21] and thus JSON Web Signatures (JWS). They are signed under a signing key held by an OP. In OpenID Connect, an OP's public keys are published on a JWKS endpoint (see Section 2.1), allowing anyone to download the public keys and verify that a given ID Token is validly signed by the OP. As a JWT, an ID Token contains claims that the OP is making about the user. The following claims are required for an ID Token to be considered valid:

**`iss`** (issuer) the URI of the OpenId Provider (OP) that issued this token *e.g.,* `https://accounts.google.com`.

**`sub`** (subject) a string that uniquely identifies the user account at the OP that the token is issued for.

**`aud`**, intended audience(s) of this ID Token, must include the OIDC-RP's Client ID.

**`iat`** (issued at), the time at which the token was issued.

**`exp`** (expiration) the time after which the token is expired.

**`nonce`** random value specified by the Client Instance in an Authentication Request. This field is only required if specified in an Authentication Request.

#### 2.2.2 Using an ID Token

A user whose Client Instance has been granted an ID Token, can authenticate to an audience by having the Client Instance send the user's ID Token to that audience.

The audience validates the ID Token by performing the following checks. First, the audience checks that the OP which generated the ID Token is an OP which the audience trusts to authenticate user identities. Second, the audience performs JWT (JSON Web Token) validation on the ID Token. This includes downloading the OP's public keys from the OP's JWKS endpoint. Third, the audience checks that the Client ID specified in the `aud` claim is one that the audience trusts, that the `aud` contains the audience and that the `aud` does not contain any other audiences which the audience does not trust. Finally, the audience ensures that ID Token is not expired based on the expiration claim in the ID Token.

#### 2.2.3 ID Token Refresh

Refresh Tokens are used to obtain new ID Tokens when they expire or become invalid. Refresh tokens are expected to be long-lived and do not have a set expiration but they can be invalidated by OP. The purpose of Refresh Tokens is to authenticate Refresh Requests made by the Client Instance to the OP to request a refreshed/unexpired ID Token. ID Token's typically expire in a few hours.

The purpose of ID Token expiration and Token Refresh in OpenID Connect is two fold. First, it limits the time an attacker can abuse a stolen ID Token. Once the ID Token expires audiences will reject them as invalid and they will provide no benefit to an attacker. This assumes the attacker does not steal the Refresh Token as well. Secondly, the Refresh Request provides a convenient place for an OP to revoke an authentication after tokens have been issued. To do this the OP internally marks the Refresh Token associated with that session as invalid so that the Refresh Token can no longer be used by a Client Instance to get fresh ID Tokens. Once the current ID Token held by the Client Instance expires, the Client Instance no longer has valid tokens.

To refresh an ID Token:

**1. Refresh Request:** The Client Instance makes a Refresh Request to the OP by sending the user's current Refresh Token to OP's Token Endpoint.

**2. Refresh Response:** If the Refresh Token sent by the Client Instance is valid *i.e.,* was issued by the OP and not revoked by the OP, then the OP replies to the Client Instance with a new Refresh Token and ID Token.

**3. Validation:** Returned tokens checked by Client Instance.

#### 2.2.4 Token Grant via PKCE Authorization Code Flow

As shown in Figure 2, the PKCE Authorization Code Flow is one way that OpenID Connect authenticates users and issues tokens. At a high level the Authorization Code Flow works as follows. The user needs an ID Token to prove their identity to an audience. To do this the user's Client Instance sends the user's web browser to the OP's Authorization Endpoint. The user authenticates with their credentials. If the OP accepts the user's credentials, the OP redirects the user's browser to the Redirect-URI specified to communicate the Auth-Code to the user's Client Instance. The Client Instance uses the Auth-Code to request tokens from the OP and receives an ID Token, an Access Token and a Refresh Token. This ends the Authorization Code Flow; the Client Instance can now send the ID Token to the audience and prove the user's identity. The main idea here is that the user proves their identity to the OP with their credentials and then the OP issues a signed ID Token which functions as an attestation that the user supplied valid credentials to the OP. Let us now walk through this flow in greater detail:

**1. Setup:** The user requests that the Client Instance perform the Authorization-Code Flow. The Client Instance prepares to initiate the flow by generating a random `nonce`, a random PKCE Code Verifier (`CV`) and a PKCE Code Challenge (`CC`) which is set to the `SHA256` hash of the Code Verifier. We use $\lambda$ to denote the security parameter used, we assume $\lambda \geq 256$.

$$\texttt{nonce} \xleftarrow{r} \{0,1\}^{\lambda}$$

$$\texttt{CV} \xleftarrow{r} \{0,1\}^{\lambda}$$

$$\texttt{CC} \leftarrow \texttt{SHA256}(\texttt{CV})$$

**2. Authentication Request:** The Client Instance initiates the Authorization-Code Flow by opening the user's web browser to the OP's Authorization Endpoint. This Authentication Request includes the following parameters set by the Client Instance `nonce`, `CC`, Client ID, and Redirect-URI. The OP reads the parameters from the web request sent to the Authorization Endpoint. If the request passes validation, the OP asks the user to consent and authenticate.

**3. User Consents/Authenticates:** The OP displays the consent and authentication page on the user's web browser. This page asks the user if they consent to have tokens issued by the OP to a Client Instance associated with a particular RP-OIDC. If the user consents they enter their credentials and other authentication factors into the web page.

**4. Authentication Response:** If the user successfully authenticates, the OP then verifies that the Redirect-URI specified in the Authentication Request is on the Redirect-URI ACL (Access Control List) for the Client ID given. If the Redirect-URI is on the ACL then the OP performs an Authentication Response. To do this the OP first generates an Auth-Code and records the user, `nonce` and `CC` associated with that Auth-Code. Then the OP transmits that Auth-Code to the Client Instance by redirecting the user's web browser to the Redirect-URI with the Auth-Code as a parameter.

**5. Token Request:** The Client Instance receives the Auth-Code when the user's web browser is redirected to Redirect-URI. The Client Instance then sends the Auth-Code along with the Code Verifier (`CV`) to the OP's Token Request Endpoint.

**6. Token Grant:** The OP verifies that the Code Verifier (`CV`) sent by the Client Instance is the SHA256 preimage of the Code Challenge (`CC`) sent in the Authentication Request, $\texttt{CC} = \texttt{SHA256}(\texttt{CV})$, that the Code Challenge (`CC`) corresponds to the Auth-Code, and that the Auth-Code is valid. If all of these checks pass, the OP responds to the Token Request with an ID Token and Refresh Token.

**7. Token Validation:** The Client Instance validates the tokens it received from the OP. The Client Instance checks that the claims in the ID Token match the claims the Client Instance expected. Specifically it checks that the `nonce` the Client Instance generated matches the value in the ID Token's `nonce` claim, that the OIDC-RP's Client ID is in the `aud` claim and that the `iss` claim matches the issuer identity for the expected OP. Note the Client Instance does not check that the `sub` claim matches the user's identity because the Client Instance does not know the identity of the user except through the ID Token.

If any of these steps fail, the Client Instance aborts and deletes all values computed for this session including the tokens. Otherwise the Client Instance accepts the ID Token.

## 3 The OpenPubkey Protocol

OpenPubkey is built around a token called a PK Token. A PK Token, as shown in Figure 1, is an ID Token constructed in such a way as to bind the user's identity in the ID Token to the user's public key, $PK_u$, under the signature of the OP which issued the ID Token. Audiences are able to use the PK Token to authenticate that a particular public key is held by a particular user identity. In this way a PK Token functions as a certificate. The PK token being a JWS can have more than one signature. The PK Token shown in Figure 1 has three signatures: the signature of the OP, the user's signature, and an MFA-cosigner signature.

In a PK Token the ID Token payload, `idT`.claims, has been constructed such that the `nonce` claim is not just a random value but a `SHA-3` hash of a set of claims made by the Client Instance. We call these claims the `cic` *aka,* the Client Instance Claims. The `cic` must include the user's public key, $PK_u$, the algorithm used to verify signatures with the user's public key, `alg`, and a random value, $rz$, chosen by the Client Instance. We allow the Client Instance to specify additional claims for extendability purposes, but here we assume a minimal `cic` containing only three claims. Thus, we repurpose the `nonce` claim to function as a randomized commitment that can opened under $rz$, to the values in `cic`. The second

```
pkT ← {
  "payload":idT.claims ←{
    "nonce":SHA-3(cic ← (PK_u,alg,rz)),
    "iss":issuer,
    "exp":expires on,
    "iat":issued at,
    "aud":audience,
    "sub":subject ID,
    "email":subject email
  },
  "signatures":
  [{
    "signature":σ_o ← SIGN(SK_o,(idT.claims,h_o)),
    "protected":h_o ←{"alg":"RS256",
      "typ": "JWT", "kid":"123"},
  },
  {
    "signature":σ_u ← SIGN(SK_u,(idT.claims,h_u)),
    "protected":h_u ←{
      "upk":PK_u,
      "alg":alg,
      "rz":rz
    },
  },
  {
    "signature":σ_m ← SIGN(SK_m,(idT.claims,h_m)),
    "protected":h_m ←{
      "alg":"EC256",
      "kid":"456",
      "csid":"https://mfa.io",
      "eid": User Auth event ID,
      "auth_time": MFA Auth time,
      "iat":issued at,
      "exp":expires on,
      "mfa":"Webauthn",
      "ruri":"https://acme.co/"
    }
  }]
}
```

Figure 1: Anatomy of the PK Token, pkT. The payload is the payload of the ID Token, idT. $(\sigma_o, h_o)$ is the signature and protected header the OP generated to grant and sign the ID token. $(\sigma_u, h_u)$, was added to the ID Token by the Client Instance to store the Client Instance Claims (cic) committed to in the nonce. $(\sigma_m, h_m)$, was added by the MFA-cosigner (Section 3.3) to show the user authenticated via MFA.

signature object, $(h_u, \sigma_u)$, which is generated by the Client Instance, provides the values of cic in the protected header, $h_u$, allowing any party to open and verify that the cic is committed to in the nonce. A PK Token is only required to have these two pairs of signatures and protected headers.[5]

The third signature, protected header pair, $(\sigma_m, h_m)$, shown in Figure 1 is not required for a PK Token. It is an optional feature of OpenPubkey that enables third party cosigners to add attestations by signing the ID Token. In this case, the cosigner is attesting that they performed an independent authentication of the user's identity via an MFA device. We describe the MFA-cosigner in in Section 3.3.

None of the claims in the cic are checked or even seen by the OP. The cic is merely proof that a particular user successfully authenticated to the OP under an identity attested to by the OP in the ID Token and the Client Instance used in that authentication flow claims that a particular public key is held by the user. As OpenID Connect trusts the Client Instance to not impersonate the user or leak a user's tokens, trusting the Client Instance to honestly specify the user's public key in the cic does not add trust assumptions.

### 3.1 Creating a PK Token

We will now show how OpenPubkey creates a PK Token by injecting a commitment to the user's public key into the nonce claim of an ID Token during OpenID Connect PKCE Authorization Code Flow. Our protocol leverages the fact that the Authorization Code Flow has the Client Instance generate and send the nonce to the OP and the OP must include this nonce in the ID Token it issues when the user completes the flow. Our description will not repeat all steps in the Authorization Code Flow as our protocol does not change any of these steps but the first step. In the Appendix D.1-D.2 we provide a step-by-step description of our protocol for native application and browser based Client Instances.

To do this we amend how the nonce is generated by the Client Instance in Step 1 of the Authorization Code Flow (Section 2.2.4). In OpenPubkey the Client Instance first generates the user's key pair, which we denote $(PK_u, SK_u)$. Then the Client Instance chooses a random value $rz$.

$$(PK_u, SK_u) \leftarrow GenKey(1^\lambda)$$

$$rz \leftarrow^r \{0,1\}^\lambda$$

These values along with user's signature algorithm, alg, constitute the Client Instance Claim, $\texttt{cic} \leftarrow (PK_u, \texttt{alg}, rz)$. Instead of randomly generating the nonce, the Client Instance turns the nonce into a SHA-3 commitment to the cic.

$$\texttt{nonce} \leftarrow \texttt{SHA-3}(\texttt{cic})$$

[5] We were inspired to use protected headers to specify additional signer claims by Section 5.3 of RFC-7519 [21] which specifies that JWT claims can be replicated to the JWS header.

The Client Instance then makes the standard OpenID Connect Authentication Request using this nonce. Steps 2-6 of the Authorization Code Flow are unchanged. If the flow completes successfully the ID Token will contain a nonce which commits to the user's public key.

Once the ID Token is granted (Step 6) and validated (Step 7), the Client Instance creates a protected header, $\mathrm{h}_u$ consisting of the values in the cic.

$$\mathrm{h}_u \leftarrow \texttt{cic}$$

and signs the ID Token under the user's signing key, $SK_u$.

$$\sigma_u \leftarrow \texttt{SIGN}(SK_u, (\texttt{idT}.\text{claims}, \mathrm{h}_u))$$

The Client Instance then completes the creation of the PK Token by adding $(\sigma_u, \mathrm{h}_u)$ to the ID Token.

As shown in Figure 2 the only difference between OpenPubkey and standard OpenID Connect is how the nonce is computed. As we do not require changes to be made outside of the Client Instance, PK Token creation is fully compatible with, and transparent to, OpenID Connect OPs.

There is no danger of OPs modifying the nonce and breaking our protocol. As OpenID Connect spec clearly states: "If present in the Authentication Request, [OP] Authorization Servers MUST include a nonce Claim in the ID Token with the Claim Value being the nonce value sent in the Authentication Request. Authorization Servers SHOULD perform no other processing on nonce values" [35] As our nonce is a randomized under *rz*, it still functions as a nonce and the OP can not distinguish it from a simple random nonce.

Creating this signature, $\sigma_u$, prevents Identity Misbinding Attacks [7, 22] (sometimes called Unknown Key-Share(UKS) attacks), in which a malicious party associates their identity with another party's public key. In an Identity Misbinding Attack the malicious party doesn't control the signing key, associated with the public key, to which they are attempting to bind their identity. By requiring that the Client Instance sign the ID Token, it proves it knows the user's signing key and thus can not be performing an Identity Misbinding Attack.

## 3.2 Verifying a PK Token

The procedure for verifying a PK Token, pkT, is as follows:

**(1).** The verifier ensures that the PK Token has Client ID, aud and iss claims it has been configured to expect.

**(2).** The verifier then extracts the kid from the OP's protected header, $\mathrm{h}_o$, in the PK Token, uses the kid to find the matching public key, $PK_o$, in the verifier's list of OP public keys. It uses $PK_o$ to verify the OP's signature, $\sigma_o$ on the PK Token: $\texttt{VER}(PK_o, (\texttt{pkT}.\text{payload}, \mathrm{h}_o), \sigma_o) = 1$

**(3).** Having validated the ID Token portion of the PK Token, the verifier next checks that the nonce claim is equal to the SHA-3 hash of the cic in user's protected header, $\mathrm{h}_o$. This ensures that the nonce claim commits to the cic.

**(4).** Finally the verifier checks that the cic is well formed and contains the user's public key $PK_u$, the user's algorithm alg, and the random value *rz*. It uses these values to verify the user's signature over the PK Token: $\texttt{VER}(PK_u, (\texttt{pkT}.\text{payload}, \mathrm{h}_u), \sigma_u) = 1$

The verifier accepts the PK Token if and only if all of these checks pass. PK Token expiration is handled outside this procedure depending on the needs of the verifier.

## 3.3 Cosigners

OpenPubkey enables third parties to cosign a PK Token and make additional claims *i.e., cosigner claims*. Cosigning is uncomplicated, a cosigner puts their cosigner claims into a protected header, signs the PK Token payload and the protected header, and then adds the protected header and resultant signature to the PK Token JWS. Because the signatures are JWS compliant they can be verified by any JWS verifier!

We require that all cosigners identify themselves by setting a cosigner ID claim csid. This is done for the convenience of the validating party. Cosigner IDs are prearranged by the cosigner and validating party. Figure 1 presents an example PK Token with a cosigner claim "csid" (cosigner ID) set to "https://mfa.io". We assume that any party attempting to validate a cosigner's signature is configured with the JWKS URI of the cosigner.

## 3.4 MFA-Cosigner

We employ this cosigner mechanism to build a MFA-cosigner who authenticates a user via an MFA (Multi-Factor Authentication) device. The MFA-cosigner then attests to this authentication by cosigning the user's PK Token. This provides an OP independent authentication of a user and allows an audience and a user to maintain security against a malicious OP. We define two protocols for the MFA cosigner: MFA-Auth and MFA-Refresh. MFA-Auth allows a user to perform authentication with their registered MFA device and have their PK Token signed by MFA-cosigner. MFA-Refresh is used by the Client Instance to refresh the MFA-cosigner signature on the user's PK Token.

MFA-Auth follows the Authorization-Code Flow pattern including the use of Redirect-URI to securely communicate an Auth-Code to the Client Instance. Unlike OpenID Connect the MFA-cosigner records the Redirect-URI used as a cosigner claim in the PK Token. This is done to empower audiences and verifiers to choose which Client Instances they wish to trust by enforcing a Redirect-URI allow lists at the audience/verifier.

Similar to refresh and expiration in ID Tokens, as discussed in Section 2.2.3, the MFA-cosigner includes an expiration claim to force Client Instances to regularly refresh their MFA signature via the MFA-Refresh protocol. This allows the MFA-cosigner to to refuse to refresh revoked MFA devices,

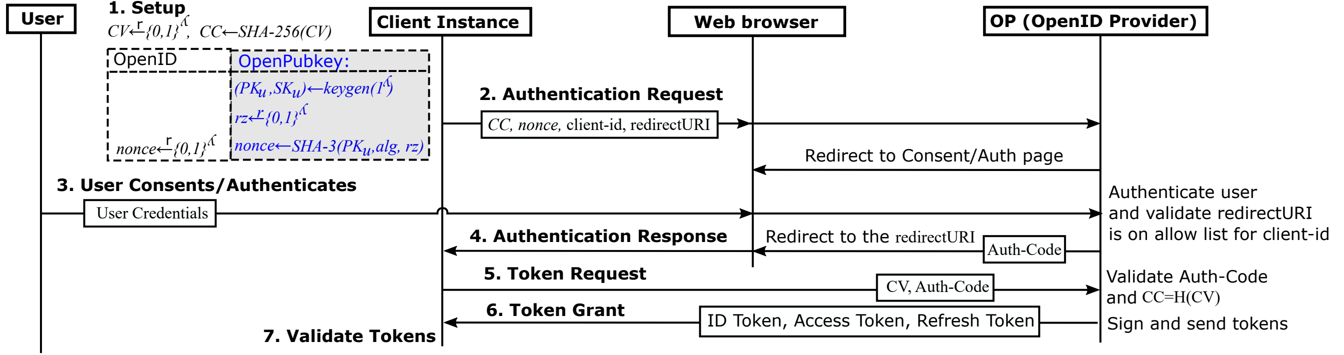


Figure 2: Text in black is the standard OpenID Connect PKCE Authorization Code Flow. Text in blue is our change which commit to a user's public key in an ID token. Our change is invisible to the OP since it only takes place on the Client Instance.

sessions and signatures. The MFA-Refresh protocol is happens in the background and does not require the user perform an MFA. It follows the Proof-of-Possession pattern. The MFA-cosigner sends a random challenge to the Client Instance. The Client Instance signs this challenge, along with some authentication metadata, under the user's public key. In response the MFA-cosigner issues a new signature for the PK Token with an updated expiration claim.

We omit a full description here as MFA-Auth and MFA-Refresh are straightforward protocols. In Appendix C we provide full description of these protocols and all the claims made by the MFA-cosigner.

#### 3.4.1 Verifying a MFA-Cosigner Signature

We assume the verifier has been preconfigured with an Redirect-URI allow list and list of MFA-cosigner public keys. Remember the cosigner claims are stored in the MFA-cosigner's protected header, $h_m$. The procedure for verifying a MFA-cosigner signature and claims on a PK Token, `pkT`, is as follows:

**(1).** The verifier ensures that the MFA-Cosigner's Redirect-URI claim, which attests to the Redirect-URI of the Client Instance to which this signature was issued, is on the verifiers allow list of Redirect-URIs.

**(2).** The verifier uses the MFA-cosigner `kid` claim to find the MFA-Cosigner's public key, $PK_m$, in the verifier's list of MFA-Cosigner public keys. It then uses $PK_m$ to verifies the MFA-cosigner's signature, $\sigma_m$ on the PK Token: $\texttt{VER}(PK_m, (\texttt{pkT}.\text{payload}, h_m), \sigma_m) = 1$

The verifier accepts the PK Token if and only if all checks pass. If signature expiration is desired by the verifier, it is enforced outside this procedure.

### 3.5 OpenPubkey Signed Messages

In this section we show how we use PK Tokens to generate and verify OpenPubkey Signed Messages (OSM). These are messages signed under the user's public key and attributable to the user's identity in the corresponding PK Token. To provide concrete examples we define two different scenarios for the verification of OSMs: Archival Verification (Section 3.5.3)) and PoP (Proof-of-Possession) Authentication (Section 3.5.4).

Our OpenPubkey Signed Messages (OSM) are just a subtype of JSON Web Signatures (JWS). What distinguishes them are a set rules with regards to the parameters set in the protected header. As shown in Figure 3, these rules are: (a). The `typ` must be set to "osm" to identify it as an OSM. (b). The `alg` must match the `alg` in the `cic` *aka,* user's protected header in the PK Token. This prevents an attacker from switching the algorithm.[6] (c). The `kid` matches the `SHA-3` hash of the user's PK Token. This binds the OSM to a particular PK Token and user identity and also identifies the public key need to verify the OSM.

As the OSM is a JWS it can support more than one signer. In Appendix F we define a protocol that makes use of multiple signatures. In this section we only deal with one signature on an OSM, the user's signature.

#### 3.5.1 Verifying OpenPubkey Signed Messages (OSM)

On receiving an OSM, `osm`, and a PK Token, `pkT`, the verifier runs the following procedure to verify the `osm`:

**(1). Challenge:** The verifier checks that the `typ` claim is set to "osm", that `osm.kid` commits to the PK Token: $\texttt{osm.h.kid} = \texttt{SHA-3}(\texttt{pkT})$, and that the algorithm in the OSM

[6] While we could technically infer the algorithm from the PK Token, `alg` is a required parameter in the JWS standard (RFC-7515) [20]. By including the `alg` we can make use of existing JWS libraries for OSM signature verification.

```
osm ← {
  "payload": m,
  "signatures":[{
    "protected": h {
      "alg": pkT.cic.alg,
      "kid": SHA-3(pkT),
      "typ": "osm",
    },
    "signature": SIGN(pkT.PK_u,(m,h))
  }]
}
```

Figure 3: JWS representation of our OpenPubkey Signed Messages (OSM). `pkT` is the user's PK Token.

matches the algorithm given the in `cic` (Client Instance Claims) in the PK Token: $\texttt{osm}.\text{h}.\texttt{alg} = \texttt{pkT.cic.alg}$

**(2). Challenge Response:** To ensure that the PK Token, `pkT`, is valid, the verifier runs the PK Token verification procedure given in Section 3.2. If the verifier wishes to enforce MFA-cosigner signatures, it also runs the MFA-cosigner verification (Section 3.4.1).

**(3). Verify:** The verifier then checks that the signature on `osm` verifies under the user's public key attested to in the `pkT`: $\texttt{VER}(\texttt{pkT}.PK_u, (\texttt{osm}.\text{payload}, \texttt{osm}.\text{h}), \sigma_u) = 1$

The verifier accepts the OSM as valid and attributable to the user identity given in the PK Token if, and only if, all of these checks pass. A verifier may wish to enforce expiration checks on the PK Token or MFA-cosigner signature and reject expired PK Tokens. We look at how expiration works next.

#### 3.5.2 Expiration enforcement

In the next two sections (Section 3.5.3 and Section 3.5.4) we will introduce two scenarios for verifying OSMs. These scenarios include a choice by the verifier to enforce or not enforce expiration on a PK Token. To set the stage for this, we will now specify our expiration enforcement mechanism.

ID Tokens have short expiration times and must to be refreshed frequently by making Refresh Requests to the OP *e.g.,* Google's OP sets a one hour expiration on ID Tokens [12]. This presents a problem for us as the `nonce` claim only appears in ID Tokens granted via the Authorization Code Flow. An ID Token granted via Refresh Requests does not have a `nonce` claim and can't be used to construct a PK Token. To enable verifying parties to enforce expiration if they so wish while avoiding the bad user experience of forcing the user to run the Authorization Code Flow every time the ID Token in their PK Token expires, we use the following alternative expiration mechanism. We do not expire a PK Token when the underlying ID Token expires according to its `exp` claim. Instead verifiers wishing to enforce expiration inspect the `iat` (issued at) claim, which specifies when the OP issued the ID Token, and reject the PK Token if it is older than two weeks. That is, if expiration is enforced, a PK Token expires two weeks after it is issued.[7]

#### 3.5.3 Archival Verification

In many cases a verifier may only care about verifying OSMs and PK Tokens which have be recently generated. In fact if a verifier has chosen to enforce expiration that verifier explicitly wants to reject OSMs which depends on an older PK Token. Yet there are many cases, such as signatures on software artifacts or verifying signed audit logs, in which a verifier can not require that the signatures be recent. Archival verification is designed for these usecases and enables the authentication of OSMs (OpenPubkey Signed Messages) years or even decades after the message was signed.

The main challenge facing archival verifiers is that the OP public key necessary to verify the PK Token may not longer be available at the OP's JWKS endpoint. Most OP's rotate their signing keys out of JWKS endpoint between every two weeks to a four times a year [12, 13, 23, 34]. To ensure the verifier has the OP's public key for the PK Token sent with the OSMs, the verifier must create and maintain an archival log of OP public keys.[8] This archival log must cover the time period that verifier wishes to verify PK Tokens over. The verifier builds this archival log by regularly downloading the public keys from the OP's JWKS endpoint. The verifier not only records the public keys but also the time at which the public keys were downloaded.

An archival verifier seeks to answer the question: "would this OSM and PK Token verify when they were originally created?" To answer this question the verifier determines when the PK Token was created by inspecting the token's `iat` (issued-at) claim. Using this it populates a list of OP public keys from the archival log that have a download time immediately before and immediately after the issued-at time. It then uses this list of OP public keys to verify the PK Token and then the OSM under the PK Token. If the verifier is configured to require a MFA-cosigner signatures it must use the same processes as used for OP signatures: maintain a log of MFA-cosigner public keys and then use that log to perform MFA-cosigner verification.

#### 3.5.4 OpenPubkey PoP Authentication

As shown in Figure 4, PoP (Proof-of-Possession) Authentication is designed as a drop in upgrade for OpenID Connect user authentication but with the enhanced security of message integrity and Proof-of-Possession (PoP). It allows a verifier who

[7] If OPs didn't strip the `nonce` claim from the ID Token during a Refresh Request we would just use the ID Token's native expiration.

[8] Note that even if OPs maintained authenticated records of past public keys, a verifier can not rely on these records being eternally available. On a time scale long enough the probability an OP shutdowns or suffers unrecoverable data loss approaches 1.

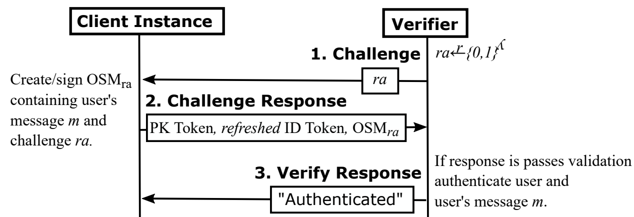


Figure 4: OpenPubkey PoP Authentication

is communicating interactively with another party to authenticate that the other the party is a user with a particular OpenID Connect identity and also that the messages sent by this party were signed by that user as part of this interactive communication session. This is contrasted with Archival Verification in which the verifier is only concerned with authenticating that a message was signed by a user. To illustrate this difference consider these two examples, in PoP Authentication the verifier asks "Am I talking live with Alice and is this what Alice is saying?" vs Archival Authentication which asks "I found a message in a log file, is it signed by Alice?".

As it is intended to be used like OpenID Connect, PoP Authentication must maintain all the security features of OpenID Connect. To achieve this PoP Authentication performs an standard OpenID Connect authentication in parallel with a PK Token authentication.

This parallel OpenID Connect authentication is especially important to enable OPs to revoke ID Tokens. In OpenID Connect, ID Token's have short expiry times which forces Client Instances to make frequent Refresh Requests to the OP to get new ID Tokens issued when the current ID Token expires. As discussed in Section 2.2.3, OpenID Connect uses these frequent Refresh Request to enable OPs to revoke authentication by revoking the Refresh Token.

The PoP Authentication protocol works as follows:

**(1). Challenge** The verifier sends the user's Client Instance a random challenge value, *ra*. This value will be used in Proof-of-Possession.

**(2). Response** The Client Instance responds to this challenge by including *ra* in the protected header of the OSM. The Client Instance adds the message content it wants to authenticate, and signs the OSM with the user's signing key, $SK_u$. It then sends the OSM, the user's PK Token and a *refreshed* ID Token to the verifier. This refreshed ID Token is an unexpired ID Token the Client Instance was issued from a recent Refresh Request.

**(3). Verify Response** The verifier receives the OSM, PK Token and refreshed ID Token. To check the OP's signatures on the refreshed ID Token and the PK Token, the verifier downloads the OP's public keys from the OP's JWKS endpoint. Then it verifies the PK Token and the OSM according to Section 3.5.1.

In addition to this the verifier checks that the challenge value *ra* is set in the protected header of the OSM. This demonstrates Proof-of-Possession of the user's signing key to the verifier. Next the verifier enforces expiration on the PK Token using the `iat` claim as detailed in Section 3.5.2. If the verifier enforces a MFA-cosigner, it verifies the MFA-cosigner's signature and rejects expired MFA-cosigner signatures.

The verifier, in parallel, performs a standard OpenID Connect authentication on the refreshed ID Token. To prevent someone using two ID Tokens that don't match the verifier checks that the refreshed ID Token is for the same user, audience, Client ID, and issuer as the ID token in the PK Token *e.g.,* the verifier must reject an authentication attempt in which a Client Instance provides a refreshed ID Token for Alice and a PK Token for Bob.

## 4 Security Discussion

In this section we provide a discussion of the security of OpenID Connect and OpenPubkey. Our approach is to step through each of the trusted parties in OpenID Connect and look at the resulting security loss if the trust assumptions placed in that party is violated by that party being compromised. We then show when and how OpenPubkey and the MFA-Cosigner can restore security in the presence of malicious OpenID Connect trusted parties. We take this approach as OpenID Connect has been shown to maintain security [15] as long as its trust assumptions are not violated.

Other than the user, the four trusted parties in OpenID Connect are: the audience, the Client Instance, the OIDC-RP and the OP. In OpenID Connect an individual compromise of any of these parties results in some loss of security. We start with a discussion of the audience and Token Replay Attacks. We follow this we the impact of a compromised Client Instance such as Token Export Attacks. Then we look at a compromised OIDC-RP how that enables Counterfeit Client Instances. Finally we examine the security consequences of a compromise of the most trusted party in OpenID Connect, the OP. In Table 1, we summarize this discussion.

### 4.1 Compromise of Audience

OpenID Connect is built on Bearer Authentication in which a user authenticates by revealing their authentication secret *i.e.,* their ID Token, to the audience. A consequence is that OpenID Connect is vulnerable to Token Replay Attacks in which an audience or party that intercepts an ID Token, can replay that token to impersonate the user. OpenID Connect limits the parties impacted by such replay attacks by scoping ID Token to a Client ID and an audience set in the ID Token claims. This means that an ID Token scoped to a set of audiences will be rejected by an audience not in that set. As stated in the OpenID Connect specification [35]: "The ID Token MUST be rejected if the ID Token does not list the Client as a valid audience, or if it contains additional audiences not

| | Replay Attacks | Export Attacks | Counterfeit Client Instance | Compromise of OP's Signing keys | Compromise of Client Instance |
|---|---|---|---|---|---|
| **OpenID Connect** | ⚠ | ⚠ | ⚠ | ⚠ | ⚠ |
| **OpenPubkey** | 🛡 | 🛡 | ⚠ | ⚠ | ↑ |
| **OpenPubkey + MFAcos** | 🛡 | 🛡 | 🛡 | 🛡 | ↑ |

Table 1: Enumerates the security impact of different trusted parties being compromised in OpenID Connect. We denote security via symbols as: ⚠- Vulnerable, ↑- Security increased but attack still possible, 🛡 - (Attack prevented).

trusted by the Client." Client here refers to the OIDC-RP. In practice most audiences protect themselves from Token Replay Attacks from other audiences by running their own OIDC-RP and only trusting ID Tokens associated with the client-id of their own OIDC-RP. In this case the OIDC-RP and the audience would be the same entity.

While having each audience trust and use a different client-id mitigates replay attacks between audiences, it does not address replay attacks within an audience *aka,* intra-audience replay attacks. In practice most audiences represent a panoply of servers and services. Bearer's Authentication creates an anti-pattern where every system within that audience which authenticates a user becomes an audience-wide Single Point of Compromise (SPoC) that defeats the audience's entire authentication enforcement system. Consider a vulnerability in an unimportant web server that leaks the content of web requests containing ID Tokens. An attacker can replay the leaked ID Tokens to that entire audience's set of services and servers.

OpenPubkey is not vulnerable to replay attacks as it enables Client Instances to perform Proof-of-Possession with the user's signing key as shown in Section 3.5.4. The user's signing key, *aka,* their authentication secret, is never sent outside the user's host.

## 4.2 Compromise of Client Instance

In OpenID Connect the Client Instance is trusted with total control over a user's tokens. If compromised or misconfigured it can perform Token Export Attacks. That is, it can read the user's tokens and send them *aka,* "export" them, to an attacker. These attacks are a serious security issue in OAuth2 and OpenID Connect, and as discussed in Related Work (Section 5), have motivated a decade of efforts to mitigate this risk. As RFC-8471 [31] states about OAuth2 bearer tokens "any party in possession of bearer security tokens gains access to certain protected resource(s). Attackers take advantage of this by exporting bearer tokens from a user's application connections or machines, presenting them to application servers, and impersonating authenticated users."

*Non-extractable keys* present a very compelling solution to this problem. Non-extractable keys are a functionality provided by modern web browsers [33], operating systems [28] and mobile devices [1, 3] which allows authorized programs to be trusted to request signatures from a signing key but not be not trusted to read or export the signing key. Unfortunately bearer tokens, such as those used by OpenID Connect, can not take advantage of non-extractable keys as the Client Instance must read the tokens and transmit them to an audience to authenticate.

Since OpenPubkey allows the use of signature-based Proof-of-Possession, it can leverage non-extractable keys to protect the user's signing key. For example a Client Instance can use the "Web Cryptography API" [33] avaliable to modern web browsers to generate the user's key pair via a call to the `generateKey` method with the `extractable` attribute set to false. As `extractable` is false, the web browser will not allow the Client Instance to learn the user's signing key $SK_u$ but will allow the Client Instance, and only the Client Instance, to request signatures.

OpenPubkey with non-extractable keys provides some protections against a compromised Client Instance, since unlike OpenID Connect a compromised OpenPubkey Client Instance can not exfiltrate the authentication secret. Yet this protection is not unlimited, a compromised OpenPubkey Client Instance can still be used by an attacker to request signatures on the attacker's behalf and use these signatures to pass Proof-of-Possession challenges.

Limiting the attacker to only requesting signatures from the Client Instance greatly restricts an attacker. First, the attacker must now route communications through the compromised Client Instance requiring that they modify the Client Instance to be accept commands from the attacker and return the results of these commands. Additionally by forcing the attacker to route communications through the user's computer, they are will leave evidence of their means, methods and intentions on a device more likely to be available to investigators than the attacker own infrastructure. Finally, if the user closes their browser window the attacker will no longer be able to request signatures. This means that even a successful attack could be defeated by a single click.

An additional limitation is that the protection offered by non-extractable keys does not apply to keys generated after a compromised. This is because the Client Instance is the party that instructs the browser to generate a key as a non-extractable key. To do this a compromised Client Instance can wait until the next time a user runs the Authorization Code Flow then replace the user's public key in the `nonce`

with an public key generated by the attacker. This attack can not be done retroactively and all keys generated prior to the compromise are safe from being exfiltrated/exported. In Appendix F we show how to use web browser security features to provide non-extractability even for keys generated after an attacker has successfully compromised a Client Instance via XSS (Cross-Site-Scripting) attack.

This type of protection is very similar to the protection provided by HSMs (Hardware Security Modules) and for this reason non-extractable keys are sometimes referred to as SSMs (Software Security Modules). We argue that despite these limitations non-extractable keys provide a significant security improvement.

### 4.3 Compromise of OIDC-RP

As discussed in Section 2.2, the Redirect-URI is specified by a Client Instance to the OP and then is used by the OP to send the Auth-Code from the OP to the Client Instance. Sending the Auth-Code to a particular Redirect-URI ensures that only Client Instances loaded from javascript origins associated with the Redirect-URI, receive the Auth-Code from the OP. This is because the ability of javascript to read from a particular Redirect-URI implies a same-origin relationship. Redirect-URIs enable similar enforcement mechanisms for Client Instance running as mobile or native apps [40].

Despite the importance of the Redirect-URI, OpenID Connect does not include the Redirect-URI to which a ID Token was issued as a claim in the ID Token. Instead OpenID Connect uses a Redirect-URI ACL (Access Control List) at the OP to determine which Redirect-URIs have been authorized for a Client ID. An OIDC-RP (OpenID Connect Relying Party) is trusted to configure the allowed Redirect-URIs for their Client ID at the OP's Redirect-URIs ACL.

A *Counterfeit Client Instance Attack* is an attack where a user is tricked into performing an OpenID Connect Authorization-Code Flow with an attacker controller Client Instance, such that the user's tokens are issued to this "Counterfeit" Client Instance. We call such an attacker controlled Client Instance a Counterfeit Client Instance, because it isn't the Client Instance the user or audience trusts but has the same Client ID as the Client Instance the user and audience trusts. The Redirect-URI ACL prevents Counterfeit Client Instance Attack because the OP will see that the Counterfeit Client Instance Redirect-URI is not authorized for that Client ID.

If the OIDC-RP is compromised, this protection no longer holds because the attacker can use the OIDC-RP's access to add the Redirect-URI of the attacker controlled Client Instance to the ACL. Since the OP will see the Counterfeit Client Instance's Redirect-URI as authorized for the Client ID, it will send the Auth-Code to the Counterfeit Client Instance's Redirect-URI enabling the Counterfeit Client Instance to learn the Auth-Code and use it to successfully request the user's tokens from the OP.

A Counterfeit Client Instance Attack is different from a compromised Client Instance as the counterfeit Client Instance is not the same Client Instance which the user trusts, it merely has the same Client ID. As a result the counterfeit Client Instance can not steal tokens already issued to the user's Client Instance. This attack only impacts audiences which trust the Client ID associated the compromised OIDC-RP. In Appendix B we provide a detailed example of the counterfeit Client Attack.

While OpenID Connect and OpenPubkey without an MFA-Cosigner are both equally vulnerable to Counterfeit Client Instance Attacks, adding an MFA-cosigner to OpenPubkey provides a strong defense against this attack. This is because when the MFA-cosigner signs the PK Token, it includes a claim to the Redirect-URI the MFA-cosigner used to privately communicate the MFA-Auth-Code (see Appendix C) to the Client Instance. Thus, the MFA-cosigned PK Token attests to the Redirect-URI used by the Client Instance. We enable audiences and other verifying parties to remove this trust assumption on OIDC-RP. Instead audiences and verifiers can maintain their own Redirect-URI ACLs and reject a Client Instance using a Redirect-URI they do not trust and have not internally authorized. If a verifier does not want to maintain a Redirect-URI ACL they can simply ignore the Redirect-URI specified by the MFA-cosigner and keep OpenID Connect's trust assumption on the OIDC-RP.

### 4.4 Compromise of OP (OpenID Provider)

The OP (OpenID Provider) is trusted to authenticate users and attest to the identity of users. This creates a Single Point of Compromise (SPoC) in OpenID Connect since an attacker who controls the OP's signing keys can produce ID Tokens attesting to anything. The MFA-cosigner, if required by a verifier, removes this single point of compromise by providing a second and independent authorization and attestation to the user's identity. While the attacker can generate a valid PK Token, the attacker can not get it signed by the MFA-cosigner without the user's MFA credentials. Thus an audience that requires an MFA-cosigner is protected even if the OP is fully malicious. Additionally since audiences always check that the PK Token is signed by the OP, if the MFA-cosigner is compromised but the OP is not compromised the audience is protected as well. An attacker must compromise both the MFA-cosigner and OP at the same time to impersonate a user. This constitutes a substantial trust reduction and security improvement over OpenID Connect.

### 4.5 No new trusted parties

OpenPubkey does not use any trusted parties not already existing in OpenID Connect and reduces trust assumptions placed in the audience and the Client Instance. While the Client Instance in OpenPubkey generates a key pair for the user, this

does not increase the trust placed in the Client Instance as OpenID Connect already fully trusts the Client Instance to manage and store the user's authentication secrets.

The MFA-cosigner reduces trust placed in the OP but its use does not add any additional trust assumptions. This is because if the MFA-cosigner is compromised the security returns to the security provided if no MFA-cosigner was used.

Additionally OpenPubkey provides the following trust reductions in OpenID Connect. By employing Proof-of-Possession, OpenPubkey completely eliminates the risk of intra-audience Token Replay Attacks resulting from the compromise of an audience. While OpenPubkey does not eliminate the trust assumption on Client Instances, OpenPubkey is able to provide improved protection against Token Export Attacks via non-extractable keys. OpenPubkey if used with a MFA-Cosigner, reduces the trust placed in the OIDC-RP by enabling audiences to implement Redirect-URI allow lists and defeat the Counterfeit Client Instance attack. In OpenID Connect, the OP (OpenID Provider) is the sole authoritative source of identity and thus the compromise of the OP's signing keys is devastating. OpenPubkey with an MFA-cosigner provides a robust defense against this attack. While the attacker can use the OP's signing keys to generate a valid PK Token, the attacker can not get the PK Token signed by the MFA-cosigner.

## 5 Related Work

There has been significant standardization work over the last decade to defend against Token Replay and Token Export Attacks in OAuth2 (RFC 6749) [16]. *Token Binding* [30–32] was a standard which mitigated these attacks by constraining a token to a particular party. It failed because it was unable to get chromium/Google Chrome [17] to make the necessary changes to support Token Binding. Certificate Bound Tokens (RFC-8705) [10] is an alternative approach which modifies the OAuth2 protocol by introducing TLS Client Certificates to OAuth2 and thereby adding Proof-of-Possession. A third proposal is the IETF draft "Demonstration of Proof-of-Possession (DPoP)" [14] which adds a new token type to OAuth2 that attests to the token holder's public key, enabling Proof-of-Possession with this signing key. OpenPubkey's Proof-of-Possession was inspired by DPoP.

OpenPubkey differs from these approaches in several respects. While they are focused on OAuth2 authorization, OpenPubkey is focused on OpenID Connect authentication. Unlike these approaches, OpenPubkey supports the authentication of message integrity in addition to identity, this unlocks a new world of protocols built on user held signing keys. User signed messages are not currently possible in OAuth2 and OpenID Connect. Critically OpenPubkey does not require changes to token issuers so it can be used today (and is being used today).

OpenPubkey PoP's JWS challenge and sign pattern is very similar to the JWS challenge and sign Request Authentication used in RFC-8555 (ACME) [5]. We are currently investigating if we can reformulate OpenPubkey PoP as an extension of ACME's Request Authentication.

Verifiable Credentials [27, 39] are an ongoing standardization project with the aim of allowing users to custody and control their digital identity documents, called "credentials", such as University Diplomas, Passports and Non-Fungable Tokens (NFTs). Verifiable credentials issued to a user can attest to that user's public key. For instance if Boston University issues Alice a diploma in the form of a Verifiable Credential, she can sign a statement which is cryptographically verifiable as coming from a Boston University graduate. In the language of Verifiable Credentials, Alice's signed message would be called a *Verifiable Presentation* [41].

[25] is a draft standard which defines a API for OpenID Connect OPs to issue verifiable credentials. Of the major OPs, only the Azure (Microsoft) OP supports this and can issue verifiable credentials at part of OpenID Connect. Enabling this functionality at Azure requires additional configuration. OpenPubkey requires no modifications or special configurations by the OP and supports all major OPs.

The User Endpoint for OpenID Verifiable Credentials [2] introduces Signed JWKs. These Signed JWKs attest to the public keys of OPs and can be used to determine what public keys an OP used, even after those public keys have been rotated off of the JWKS endpoint. If Signed JWKs were widely supported, OpenPubkey archival verifiers (Section 3.5.3) would no longer need to maintain a log of past OP public keys.

Alternatively, the Verifiable Credential issuer and OP can be separate parties. In this setting the Verifiable Credential issuer issues the verifiable credential when a user authenticates by presenting an ID Token. This approach does not require changes to OPs but the verifiable credential issuer now represents an additional highly trusted party. OpenPubkey is carefully designed to avoid adding trusted parties.

Sigstore [37] is an open source project for signing and verifying software artifacts. Users can sign under their OpenID Connect identity by using the sigstore Fulcio Certificate Authority [38] which uses an immutable log to store a mapping between an OpenID Connect ID Token and a short lived public key enabling parties to attribute signatures to identities. The Fulcio CA (Certificate Authority) is trusted to create this mapping between an ID Token and a public key. Using OpenPubkey this trust assumption could be removed as OpenPubkey does not need a trusted party to map ID Tokens to public keys. The Fulcio CA could in turn help OpenPubkey by acting as a public OpenPubkey verifier and OP public key database.

The Zoom Cryptography Whitepaper [8] proposes adding a *zoom-identity-snapshot* claim to OpenID Connect ID Tokens as a "mechanism for IDPs to issue and for clients to verify—a signed attestation that binds a user's email address to their set of devices and keys". That is, they want a way to bind one

or more public keys to an identity in an ID Token. As their whitepaper notes, "[to do this] compatible identity providers will need to support a custom extension of the OpenID Connect (OIDC) protocol". This custom extension is necessary because adding new claims to ID Tokens requires modifications to the OP that issues those ID Tokens. This a natural fit for OpenPubkey, because the problem that OpenPubkey solves is binding public keys to email addresses via ID Tokens. To illustrate the power and generality of OpenPubkey, let's look at two ways OpenPubkey can enable the zoom-identity-snapshot to be used today.

One of the benefits of OpenPubkey's `cic` (Client Instance Claims) design is that, unlike adding new claims to the ID Token directly, new claims can be attested to in an ID Token using the `cic`, and this does not require any changes to the OP. The only limitation to this approach is that the claims added to the `cic` will be opaque to the OP. As the zoom-identity-snapshot as opaque to the OP, "the identity provider can treat this attribute as an opaque string and does not need to check its validity" [8], the zoom-identity-snapshot can easily be added as an additional claim to OpenPubkey's `cic` (Client Instance Claims) by a Client Instance. Alternatively, a user could attest to the validity of a zoom-identity-snapshot and bind it to their email/identity, by signing it as an OSM (OpenPubkey Signed Message) under their PK Token. Using either of these two strategies, OpenPubkey enables zoom-identity-snapshots to be deployed today and does require any changes to OPs (OpenID Providers).

The FIDO2 standard WebAuthn [4, 18] provides a mechanism for users to perform a Multifactor Authentication (MFA) via Proof-of-Possession using user held signing keys. WebAuthn is not designed to generate publicly verifiable authenticated messages, and is thus complementary to OpenPubkey. Our MFA-cosigner can reap the security benefits of WebAuthn by using it as an MFA authentication protocol.

[15] provides a formal security analysis of OpenID Connect, showing that it provides the security it promises within its trust assumptions. [24] looks for vulnerabilities in OpenID Connect implementations that use the Google OpenID Provider. Our security analysis of OpenID Connect differs from these two works in that they focus on security failures resulting from implementation errors, whereas we focus on security failures resulting from the violation of trust assumptions. [40] examines the security of OAuth2 against implementation failures and malicious trusted parties. [26] uses the notion of an malicious OP to perform automated security analysis of OpenID Connect implementations.

## 6 Conclusion

OpenPubkey extends OpenID Connect with user generated signatures whose authentication and integrity are protected by the security of OpenID Connect. OpenPubkey is transparent to users and OpenID Providers, is fully compatible with all currently deployed OpenID Providers such as Google, Microsoft, Okta and Onelogin. It is currently in use. This is achieved without the creation of trusted parties or trust assumptions not already present in OpenID Connect. As OpenPubkey upgrades OpenID Connect from Bearer Authentication to Proof-of-Possession Authentication, it eliminates existing trust assumptions in OpenID Connect resulting from OpenID Connect's inherent vulnerability to Token Replay and Token Export Attacks. We provide a substantial trust reduction as our MFA-cosigner enables security to be maintained even against a fully malicious OpenID Provider.

## 7 Acknowledgements:

We would like to thank John Merfeld and Mayank Varia for their feedback and comments.

## References

[1] Android. Android developers : android keystore system. https://developer.android.com/training/articles/keystore, 2022.

[2] M. Ansari, R. Barnes, P. Kasselman, and K. Yasuda. OpenID Connect UserInfo Verifiable Credentials 1.0. Specification, OpenID Foundation, 2022. https://openid.net/specs/openid-connect-userinfo-vc-1_0.html.

[3] Apple. Certificate, key, and trust services : storing keys in the keychain. https://developer.apple.com/documentation/security /certificate_key_and_trust_services/, 2018.

[4] D. Balfanz, A. Czeskis, J. Hodges, J.C. Jones, M. Jones, A. Kumar, A. Liao, R. Lindemann, E. Lundberg, V. Bharadwaj, A. Birgisson, H. Le Van Gong, C. Brand, A. Langley, G. Mandyam, M. West, and J. Yasskin. Web Authentication: An API for accessing Public Key Credentials. W3c recommendation, World Wide Web Consortium, 3 2019. https://www.w3.org/TR/webauthn-1/.

[5] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kaste)n. RFC 8555: Automatic certificate management environment (ACME. RFC 8555, RFC Editor, 3 2019. https://www.rfc-editor.org/rfc/rfc8555.

[6] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. https://ed25519.cr.yp.to/ed25519-20110926.pdf.

[7] Simon Blake-Wilson and Alfred Menezes. Unknown key-share attacks on the station-to-station (sts) protocol. In *Public Key Cryptography: Second International*

*Workshop on Practice and Theory in Public Key Cryptography, PKC'99 Kamakura, Japan, March 1–3, 1999 Proceedings 2*, pages 154–170. Springer, 1999.

[8] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Armin Namavari, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. Zoom cryptography whitepaper (version 4.0). *Zoom Video Commun., Inc., San Jose, CA, Tech. Rep. Version*, 11 2022.

[9] J. Bradley and N. Agarwal. RFC 7636: Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, RFC Editor, 9 2015. https://datatracker.ietf.org/doc/html/rfc7636.

[10] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt. RFC 8705: OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705, RFC Editor, 2 2020. https://datatracker.ietf.org/doc/html/rfc8705.

[11] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[12] Google Cloud Authentication Documentation. Token types. https://cloud.google.com/docs/authentication/token-types.

[13] Okta Documentation. Okta developer: key rotation. https://developer.okta.com/docs/concepts/key-rotation/.

[14] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite. OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP). Draft rfc, IETF, 8 2022. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop.

[15] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202. IEEE, 2017.

[16] Ed. D. Hardt. RFC 6749: The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, 8 2012. https://datatracker.ietf.org/doc/html/rfc6749.

[17] Nick Harper. Intent to remove: token binding (chromium). https://groups.google.com/a/chromium.org/g/blink-dev/c/OkdLUyYmY1E.

[18] Takanori Isobe and Ryoma Ito. Security analysis of end-to-end encryption for zoom meetings. *IACR Cryptol. ePrint Arch.*, 2021:486, 2021. https://eprint.iacr.org/2021/486.pdf.

[19] M Jones. RFC 7517: JSON Web Key (JWK). RFC 7517, RFC Editor, 5 2015. https://datatracker.ietf.org/doc/html/rfc7517.

[20] M Jones, J Bradley, and N Sakimura. RFC 7515: JSON Web Signature (JWS). RFC 7515, RFC Editor, 5 2015. https://datatracker.ietf.org/doc/html/rfc7515.

[21] M Jones, J Bradley, and N Sakimura. RFC 7519: JSON Web Token (JWT). RFC 7519, RFC Editor, 5 2015. https://datatracker.ietf.org/doc/html/rfc7519.

[22] Hugo Krawczyk. Sigma: The'sign-and-mac'approach to authenticated diffie-hellman and its use in the ike-protocols. In *Crypto*, volume 2729, pages 400–425. Springer, 2003.

[23] Microsoft Learn. Signing key rollover in the microsoft identity platform. https://learn.microsoft.com/en-us/azure/active-directory/develop/active-directory-signing-key-rollover.

[24] Wanpeng Li and Chris J Mitchell. Analysing the security of google's implementation of openid connect. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 357–376. Springer, 2016.

[25] T. Lodderstedt, K. Yasuda, and T. Looker. OpenID for Verifiable Credential Issuance. Standards track, OpenID Foundation, 10 2022. https://openid.net/specs/openid-4-verifiable-credential-issuance-1_0.html.

[26] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. Sok: Single sign-on security—an evaluation of openid connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 251–266. IEEE, 2017.

[27] S. McCarron, J. Andrieu, M. Stone, T. Siegman, G. Kellogg, T. Thibodeau, G. Kellogg, N. Otto, S. Lee, B. Sletten, D. Burnett, M. Sporny, and K. Ebert. Verifiable Credentials Use Cases. Working group note, World Wide Web Consortium, 9 2019. https://www.w3.org/TR/vc-use-cases/.

[28] Microsoft. Cryptography api: Next generation (cng). https://learn.microsoft.com/en-us/windows/win32/seccng/cng-portal, 2010.

[29] M Miller. RFC 7520: JSON Object Signing and Encryption (JOSE). RFC 7520, RFC Editor, 5 2015. https://datatracker.ietf.org/doc/html/rfc7520.

[30] A. Popov, M. Nystroem, and D. Balfanz. RFC 8472: Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation. RFC 8472, RFC Editor, 10 2018. https://www.rfc-editor.org/rfc/rfc8472.html.

[31] A. Popov, M. Nystroem, D. Balfanz, and J. Hodges. RFC 8471: The Token Binding Protocol Version 1.0. RFC 8471, RFC Editor, 10 2018. https://datatracker.ietf.org/doc/html/rfc8471.

[32] A. Popov, M. Nystroem, D. Balfanz, and J. Hodges. RFC 8473: Token Binding over HTTP. RFC 8473, RFC Editor, 10 2018. https://www.rfc-editor.org/rfc/rfc8473.html.

[33] W3C Recommendation. Web cryptography api. https://www.w3.org/TR/WebCryptoAPI/, 2017.

[34] OneLogin OpenId Connect API Reference. Rotate signing key. https://developers.onelogin.com/openid-connect/api/rotate-key.

[35] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. Openid connect core 1.0 incorporating errata set 1. specification 7636, OpenID Foundation (OIDF), 9 2015.

[36] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *USENIX Security Symposium*, pages 713–727, 2017.

[37] Sigstore. A new standard for signing, verifying and protecting software. https://www.sigstore.dev/.

[38] Sigstore. Fulcio: A free-to-use ca for code signing. https://github.com/sigstore/fulcio.

[39] M. Sporny, G. Noble, D. Longley, D. Burnett, B. Zundel, K. Hartog, D. Longley, and D. Chadwick. Verifiable Credentials Data Model v1.1. W3c recommendation, World Wide Web Consortium, 9 2019. https://www.w3.org/TR/vc-data-model/.

[40] Ed. T. Lodderstedt, M. McGloin, and P. Hunt. RFC 6819: OAuth 2.0 Threat Model and Security Considerations. RFC 6819, RFC Editor, 1 2013. https://www.rfc-editor.org/rfc/rfc6819.

[41] O. Terbu, T. Lodderstedt, K. Yasuda, and A. Lemmon. OpenID for Verifiable Presentations. Standards track, OpenID Foundation, 9 2022. https://openid.net/specs/openid-4-verifiable-presentations-1_0.html.

# A Differences between the protocol BastionZero uses and OpenPubkey

We at BastionZero have deployed a variant of OpenPubkey and it is currently used in BastionZero (`https://bastionzero.com`) to secure remote access sessions. BastionZero is actively evolving this deployed protocol to bring it in fully in line with OpenPubkey as described in this paper. Currently there are several differences between the deployed software and the protocol described in this paper:

1. BastionZero does not use JSON Web Signatures (JWS) in signed messages or the PK Token format. Instead BastionZero uses a custom serialization format.
2. BastionZero don't use a MFA-cosigner, instead when a user performs an MFA authentication a session cookie is issued. This session cookie is checked by an in-path security system that drops messages that do not have a valid session cookie. This architecture provides equivalent security to having a cosigner in terms of MFA authentication. However as a result of this architecture, BastionZero can not take advantage of the Redirect-URI MFA-Cosigner claim and enforce Redirect-URI ACLs outside of the OP.
3. The only user-held signing key algorithm BastionZero supports is EdDSA [6]. Non-extractable keys in web browsers do not currently support EdDSA. For this reason BastionZero is unable to support non-extractable keys.
4. BastionZero currently only uses OpenPubkey to secure connections from a desktop application and does not use OpenPubkey in the browser.
5. BastionZero does not go directly from the Client Instance to the OP when creating a PK Token. Instead it first goes to a web page which enables the user to select which OP they wish to sign in with.
6. BastionZero does not currently support or use archival verification. BastionZero only uses PoP Authentication

# B Counterfeit Client Instance Attack

In this section we provide a detailed example of the Client Instance Counterfeit Attack [40]. As discussed in Section 4.3, Client Instance Counterfeit Attack is possible if an attacker compromises OIDC-RP and also succeeds, by some deceit, into tricking the user into initiating and completing an Authentication Request with the OP using the attacker crafted Counterfeit Client Instance. This attack is different from a compromised Client Instance as the Counterfeit Client Instance is not the same Client Instance which the user expects and trusts. Since the Counterfeit Client Instance does not have

access to the storage or memory of the user's Client Instance it can not steal tokens already issued to the user's Client Instance. However as the name suggests, the Counterfeit Client Instance is pretending to be the Client Instance that the user trusts by using the same Client ID. This attack only impacts users and audiences which trust the Client ID associated the compromised OIDC-RP.

This attack is enabled by the compromise of the OIDC-RP because the Redirect-URI ACL configured by the OIDC-RP and enforced by the OP, defends against the substitution of Counterfeit Client Instances. However if the OIDC-RP is compromised, this defense no longer holds as the attacker can abuse the authority of the OIDC-RP to add any Redirect-URI they want to the OP's Redirect-URI ACL associated with the OIDC-RP's Client ID. It is important to note that, because of our use of PKCE, if the honest Client Instance initiates the Authentication request, the attacker will not be able to learn the tokens issued to the user as they do not know the PKCE Code Verifier (`CV`) secret committed to in the Authentication request.

Before we step through a concrete example of this attack we first introduce the players:

A user Alice, who trusts a browser-based Client Instance with the Client ID `12345`, to authenticate to an audience `https://audience.com/`. The Client Instance is loaded from `https://example.org/` and thus has the javascript origin `https://example.org/`. The Client Instance receives the Auth-Code via the Redirect-URI, `https://example.org/redirect/`.

An OIDC-RP Bob, who is authorized to configure the Redirect-URI ACL at the OP for the Client ID `12345`. That is, Bob gets to decide which Redirect-URIs are approved for Client Instance that Alice uses. At some point in the past Bob has configured the OP's ACL with the Redirect-URI `https://example.org/redirect/` as the only authorized Redirect-URI for the Client ID `12345`.

An attacker Eve, who wants to trick Alice into using a Counterfeit Client Instance loaded from `https://evil.com/`. The Counterfeit Client Instance copies the same Client ID as the user's Client Instance *i.e.,* Client ID `12345`. Because browsers enforce same-origin policy, Eve's Counterfeit Client Instance with origin `https://evil.com/` can not read the Auth-Code sent to the Redirect-URI `https://example.org/redirect/`. Instead Eve's Counterfeit Client Instance uses the Redirect-URI `https://evil.com/redirect/`. Because Eve's Counterfeit Client Instance uses the Client ID `12345`, the OP will not send the Auth-Code to Redirect-URI `https://evil.com/redirect/` because that URI is not on the OP's ACL for Client ID `12345`.

We will now show how Eve can trick Alice into using Eve's Counterfeit Client Instance if Eve compromises the OIDC-RP Bob.

1. Eve compromises Bob *i.e.,* the OIDC-RP who is trusted to configure the OP's Redirect-URI ACL for the Client-ID `12345`.

2. Eve leverages Bob's access to add the Redirect-URI `https://evil.com/redirect/` to the OP's Redirect-URI ACL for the Client-ID `12345`. This means that her Counterfeit Client Instance can now receive an Auth-Code for Client-ID `12345`.

3. Eve now needs to trick Alice into performing an Authorization-Code Flow with the Counterfeit Client Instance at `https://evil.com`. To do this Eve sends Alice a phishing email with a link to `https://evil.com` asking Alice to sign into her SSO.

4. Alice clicks this link and does not notice that the URI in the email is not the URI of her Client Instance (`https://example.org/`) but rather the URL of the Counterfeit Client Instance (`https://evil.com`).

5. Alice's browser loads `https://evil.com`, downloads and runs the Counterfeit Client Instance. The Counterfeit Client Instance initiates a Authentication Request to the OP with an attacker chosen `nonce`, PKCE `CC`, Redirect-URI (`https://evil.com/redirect/`) and Client ID `12345`. This redirects Alice's browser window to the OP's Consent and Authentication page. Alice is unlikely to notice that the URI that first loaded, `https://evil.com`, is not what she expects, since her browser was immediately redirected to OP's Consent and Authentication page that she expects and trusts. All of this happens in a fraction of a second from Alice clicking the link in the email.

6. The OP presents Alice with the Consent and Authentication page. Alice provides her credentials and the OP redirects Alice's browser to the attacker chosen Redirect-URI (`https://evil.com/redirect`) with the Auth-Code.

7. The Counterfeit Client Instance learns the Auth-Code when the Redirect-URI loads and makes a Token Request using the Auth-Code and the PKCE `CV`. The attacker knows the `CV` because the Counterfeit Client Instance initiated the Authentication request and was able to choose the `CV`.

8. The OP responds by issuing the Counterfeit Client Instance the ID Token, Access Token and the Refresh Token for Alice with the Client ID `12345`.

9. The Counterfeit Client Instance can then send Alice's ID Token and Refresh Token to Eve. Eve can then use the ID Token to impersonate Alice to any audience which trusts ID Tokens with Client ID `12345`.

OpenPubkey with the MFA-Cosigner defeats this attack by having the MFA-Cosigner attest to the Redirect-URI used to send the MFA-Auth-Code to the Client Instance. Thus, if OpenPubkey with the MFA-Cosigner was used, the PK Token would specify that the Redirect-URI was `https://evil.com/redirect`. The audience, if it required an MFA-Cosigner signature, could inspect this value. If `https://evil.com/redirect` is not on the audience's Redirect-URI ACL, the audience will reject this ID Token even if the Redirect-URI is on the OP's Redirect-URI ACL.

# C MFA-Cosigner Protocols

Our MFA-cosigner uses two protocols: MFA-Auth and MFA-Refresh. MFA-Auth allows a user to perform authentication with their registered MFA device and have their PK Token cosigned. Refresh is used by the Client Instance to refresh the MFA-cosigner signature on their PK Token. We finish this section by presenting our MFA-cosigner validation procedure.

We assume that any party attempting to validate the MFA-cosigners attestation is configured with the JWKS URI of the MFA-cosigner. Our MFA-cosigner protocol borrows heavily from the Authorization Code Flow including the notion of a redirect-URI. Similar to OpenID Connect this Redirect-URI is used to securely communicate with a Client Instance, unlike OpenID Connect the Redirect-URI used is also included as a cosigner claim. This is done to empower validating parties to enforce Redirect-URI ACLs (Access Control Lists). Our MFA-Auth and MFA-Refresh protocols are intentionally modeled on OAuth2's Authentication-Code Flow to leverage the familiarity practitioners have with the much loved Authorization-Code Flow.

## C.1 MFA-cosigner claims

There are several required cosigner claims that an MFA-cosigner must set in the protected header of the cosigner's signature in the PK Token. The MFA-cosigner must specify the following cosigner claims:

**`csid`**, the cosigner ID. This is required for all cosigners.

**`kid`**, key ID that the MFA-cosigner used to sign and issue the cosigner claims.

**`alg`**, algorithm that should be used to verify the signature.

**`eid`**, unique id of the MFA authentication the user performed. This useful for linking a particular user authentication event across refreshed MFA signatures. Additionally this provides a useful handle if an authorized party wants to query the MFA-cosigner for more granular and sensitive details about a particular authentication session.

**`auth_time`**, the date time at which the user performed the MFA authentication.

**`iat`**, the date time at which this particular MFA signature was signed and issued. This differs from the `auth_time` which records the time at which the user MFA authentication took place because the `iat` will record the date time the at which the refreshed MFA signature was issued. This follows the pattern established by JWT claims.

**`exp`**, the date time at which this signature expires.

**`mfa`**, MFA protocol and device employed by the user *e.g.,* WebAuthn, ToTP.

**`ruri`**, the Redirect-URI that was used to communicate the MFA-Auth-Code to the Client Instance.

The MFA-cosigner claims do not include values such as the client-id or the user. This because these claims as these are already present in the PK Token and the MFA-cosigner's signature covers and thus already commits to these values.

## C.2 MFA-Auth

The MFA-Auth protocol allows a user to perform an MFA authentication to the MFA-Cosigner and get the MFA-Cosigner's signature and claims added to the user's PK Token. This signature functions as an attestation by the MFA-Cosigner, that the user authenticated successfully to the MFA-Cosigner.

**1. Setup:** The Client Instance initiates an OpenPubkey PoP Authentication with the MFA-Cosigner by requesting and receiving a *ra* challenge (See Section 3.5.4). The Client Instance creates and signs a OSM, denoted `osm`, which contains the *ra* challenge value and the Redirect-URI that the Client Instance wishes to use.

**2. MFA-Request:** The user's Client Instance opens the user's web browser to the MFA-cosigner's MFA-Request endpoint. In this request the Client Instance sends all the values needed to complete a PoP Authentication. That is, the `osm`, PK Token, and refreshed ID Token `idT`'. The MFA-Cosigner performs the PoP Authentication verification.

**3. User Consents/Authenticates:** If verification completes successfully, the MFA-cosigner requests the user perform an authentication with their registered MFA hardware device.

**4. MFA-Response:** If user successfully authenticates to the MFA-cosigner. The MFA-cosigner generates an MFA-Auth-Code and associates it with the PK Token of the user and the details of the MFA authentication. The MFA-Cosigner communicates the MFA-Auth-Code to the Client Instance by redirects the user's web browser to the Redirect-URI specified in `osm` sent in the MFA-Request.

**5. Sign-Request:** The Client Instance receives the MFA-Auth-Code via the Redirect-URI, then initiates a second OpenPubkey PoP Authentication. This time the Client Instance uses the MFA-Auth-Codeas the random challenge value *ra*. The Client Instance then sends the OSM containg the MFA-Auth-Code, PK Token, and ID Token' to the Cosigner's Sign-Request endpoint.

**6. Sign-Grant:** The MFA-Cosigner receives and performs a PoP Authentication verification on these values. In addition it checks that the PK Token sent is the same PK Token used in the first PoP sent in the MFA-Request. If these checks pass,

the MFA-cosigner writes its cosigner claims to a protected header and then signs the ID Token in the PK Token, generating the signature, $\sigma_m$. This signature covers the ID Token claims and the protected header, $h_m$. The MFA-cosigner then responds to the Client Instance Sign-Request by sending the signature, $\sigma_m$, and the protected header. The Client Instance updates the ID Token in the PK Token with the addition of this new signature object consisting of the protected header and signature pair $(h_m, \sigma_m)$.

The first PoP Authentication (MFA-Request) is used by the MFA-Cosigner to ensure that an attacker who steals a user's MFA device can not perform any action without also compromising the user's Client Instance. The second PoP Authentication (Sign-Request) performs the same role as the PKCE Code Challenge and Code Verifier in OAuth2 and OpenID Connect with the user's signing key taking the place of the Code Verifier. That is, it ensures the Client Instance which starts the MFA-Request is the only Client Instance which can use the MFA-Auth-Code to redeem the MFA-cosigner's signature.

## C.3 MFA-Refresh

The MFA-Cosigner claims include an expiration time for the signature to force Client Instances to regularly refresh their MFA signature. Audiences may reject expired MFA signatures. The MFA-Cosigner maintains a list of revoked MFA devices and signatures. It then uses this revocation list to refuse to refresh revoked sessions or devices. To refresh the MFA-Cosigner signature a Client Instance performs a PoP authentication with the MFA-Cosigner.

**1. Challenge:** MFA-Cosigner sends the user a random challenge value, `ra`.

**2. Refresh Request:** The Client Instance creates a OSM containing, ("MFA refresh request", `csid`, `authid`, PK Token, `ra`), and signs it. The Client Instance sends this OSM, the user's PK Token, and a refreshed ID Token' to the MFA-cosigner's Refresh endpoint.

The MFA-Cosigner receives and verifies the OSM using PoP Authentication. It checks that the PK Token in the message contains the MFA-cosigner's signature. The MFA-cosigner checks that the `eid` in the MFA-Cosigner claims in the PK Token is not on the cosigner's revocation list. If the `eid` is on the revocation list it refuses to sign and aborts the protocol. Otherwise the MFA-cosigner produces a new signature object with the same claims as the previous signature object but with the `iat` and `exp` updated.

**3. Refresh Response:** The MFA-cosigner responds with the new signature and protected header containing the updated claims. The Client Instance updates their PK Token with the new signature, protected header pair.

## C.4 MFA-Cosigner validation

To validate a PK Token signed by the MFA-cosigner the verifier performs the following actions. First, the verifier ensures that the MFA-cosigner claim `ruri` which attests to the Client Instance using a redirect-URI is on the verifier's Redirect-URI ACL. This enables the verifier to make a decision about what Client Instances it is willing to trust. The verifier then checks that the MFA-cosigner's signature verifies under the set of public keys the verifier associates with the MFA-cosigner. We expect the verifier to build this set from the MFA-cosigners JWKS endpoint.

Like with PK Tokens, expiration is handled outside the validation procedure as not all of OpenPubkey's authentication scenarios reject expired MFA-cosigner signatures *e.g.,* Archival Verification. If the verifier wishes to enforce expiration, it rejects any MFA-cosigner signature whose expiration claim is in the past: $\texttt{exp} < \texttt{datetime.now()}$.

# D OpenPubkey Auth-Code Flows

In this section we provide the exact steps for creating an PK Token using a native application (*aka,* desktop app) Client Instance or a web browser-based Client Instance.

## D.1 Full Authorization Code Flow using a Client Instance running as a native application

In this protocol description we assume that the Client Instance is a local application, *aka,* a native app, that runs outside of the user's web browser. This Client Instance is configured with four different localhost Redirect URI each with a different port. These match localhost Redirect URIs that are configured at the OP's Redirect URIs ACL by the OIDC-RP.

**1. Setup:** The user requests that the Client Instance perform the Authorization-Code Flow to generate a PK Token. The Client Instance prepares to initiate the flow by generating user key pair $(PK_u, SK_u)$, a random value $rz$:

$$(PK_u, SK_u) \leftarrow GenKey(1^\lambda)$$
$$rz \leftarrow^r \{0,1\}^\lambda$$

These values along with signature algorithm used by the user's public key, $PK_u$, constitute the Client Instance Claim, $\texttt{cic} \leftarrow (PK_u, \texttt{alg}, rz)$. Rather than randomly generating the `nonce`, the Client Instance instead commits to the `cic` in the `nonce` by setting the `nonce` to the `SHA-3` hash of the `cic`.

$$\texttt{nonce} \leftarrow \texttt{SHA-3}(\texttt{cic})$$

Then as is in OpenID Connect, the Client Instance creates a random PKCE Code Verifier (`CV`) and a PKCE Code Challenge (`CC`) which is set to the `SHA256` hash of the Code Verifier.

$$\texttt{CV} \xleftarrow{r} \{0,1\}^{\lambda}$$

$$\texttt{CC} \leftarrow \texttt{SHA256}(\texttt{CV})$$

The Client Instance binds an HTTP server to a port on localhost. This localhost port is where the redirectURI will be used to send the Auth-Code and the port must be the same port as the OIDC-RP configured on the Redirect URI allow list. If this port is in use it attempts one of the other four ports. If all four ports are in use it fails. If it successfully binds sets the redirectURI to the URI which matches the port the Client Instance successfully bound to.

**2. Authentication Request:** The Client Instance initiates the Authorization-Code Flow by performing Authentication Request that opens the user's web browser to the OP's Authorization Endpoint. This request includes the following parameters set by the Client Instance nonce, CC, client-id, and redirectURI. The redirectURI is set to https://localhost:<port>/callback, where the port is the localhost port that the client is listening on. If the request passes validation, the OP asks the user to consent and authenticate.

**3. User Consents/Authenticates:** The OP displays the consent and authentication page on the user's web browser. This page asks the user if they consent to have tokens issued by the OP to a Client Instance associated with a particular RP-OIDC. If the user consents they enter their credentials and other authentication factors into the web page.

**4. Authentication Response:** If the user successfully authenticates, the OP then verifies that the redirectURI specified in the Authentication Request is on the allow list of redirect URLs for the client-id given. If redirectURI is on the allow list then the OP performs an Authentication Response. To do this the OP first generates an Auth-Code and records the user, nonce and CC associated with that Auth-Code. Then the OP transmits that Auth-Code to the Client Instance by redirecting the user's web browser to the redirectURI and specifying the Auth-Code as a parameter in that URI.

**5. Token request:** The redirectURI is set to the localhost port that the Client Instance is listening. The Client Instance receives Auth-Code when the user's web browser is redirected to the redirectURI. The Client Instance then sends the Auth-Code along with the Code Verifier (CV) to the OP's Token Request Endpoint. Typically this is done by the Client Instance making a behind the scenes HTTPS request directly to the OP's Token endpoint.

**6. Token Grant:** The OP verifies that the Code Verifier (CV) sent by the Client Instance is the SHA256 preimage of the Code Challenge (CC) sent in the Authentication Request, $\texttt{CC} = \texttt{SHA256}(\texttt{CV})$, that the Code Challenge (CC) corresponds to the Auth-Code, and that the Auth-Code is valid. If all of these checks pass, the OP responds to the Token Request with an ID Token and Refresh Token.

**7. Token Validation:** The Client Instance validates the tokens it received from the OP. The Client Instance checks that the claims in the ID Token match the claims the Client Instance expected. Specifically it checks that the nonce the Client Instance generated matches the value in the ID Token's nonce claim, that the OIDC-RP's client-id is in the aud claim and that the iss claim matches the issuer identity for the expected OP. Note the Client Instance does not check that the sub claim matches the user's identity because the Client Instance does not know the identity of the user except through the ID Token.

**8. PK Token Creation:** The Client Instance creates a protected header, $\mathrm{h}_u$ consisting of the values in the cic.

$$\mathrm{h}_u \leftarrow \texttt{cic}$$

and signs the ID Token under the user's signing key, $SK_u$.

$$\sigma_u \leftarrow \texttt{SIGN}(SK_u, (\texttt{idT}.\text{claims}, \mathrm{h}_u))$$

The Client Instance then constructs the PK Token by adding $(\sigma_u, \mathrm{h}_u)$ to the signature list of the JWS representation of the ID Token.

If any of these steps fail, the Client Instance aborts and deletes all values computed for this session including the tokens. Otherwise the Client Instance accepts the ID Token uses it to authenticate the user's identity to the intended audience.

## D.2 Full Authorization Code Flow using a Client Instance running in a Web Browser

In this protocol description the Client Instance is running inside a web browser as a javascript application. The Client Instance is configured with a single Redirect URI, redirectURI, which is the same as a Redirect URI configured at the OP by the OIDC-RP. This Redirect URI represents the javascript origin of Client Instance.

**1. Setup:** The user requests that the Client Instance perform the Authorization-Code Flow to generate a PK Token. The Client Instance prepares to initiate the flow by generating user key pair $(PK_u, SK_u)$, a random value $rz$:

$$(PK_u, SK_u) \leftarrow GenKey(1^{\lambda})$$

$$rz \xleftarrow{r} \{0,1\}^{\lambda}$$

These values along with signature algorithm used by the user's public key, $PK_u$, constitute the Client Instance Claim, $\texttt{cic} \leftarrow (PK_u, \texttt{alg}, rz)$. Rather than randomly generating the nonce, the Client Instance instead commits to the cic in the nonce by setting the nonce to the SHA-3 hash of the cic.

$$\texttt{nonce} \leftarrow \texttt{SHA-3}(\texttt{cic})$$

Then as is in OpenID Connect, the Client Instance creates a random PKCE Code Verifier (CV) and a PKCE Code

Challenge (CC) which is set to the SHA256 hash of the Code Verifier.

$$\texttt{CV} \xleftarrow{r} \{0,1\}^{\lambda}$$

$$\texttt{CC} \leftarrow \texttt{SHA256}(\texttt{CV})$$

**2. Authentication Request:** The Client Instance initiates the Authorization-Code Flow by performing Authentication Request that opens the user's web browser to the OP's Authorization Endpoint. This request includes the following parameters set by the Client Instance `nonce`, `CC`, client-id, and `redirectURI`. The `redirectURI` is set to `https://localhost:<port>/callback`, where the port is the localhost port that the client is listening on. If the request passes validation, the OP asks the user to consent and authenticate.

**3. User Consents/Authenticates:** The OP displays the consent and authentication page on the user's web browser. This page asks the user if they consent to have tokens issued by the OP to a Client Instance associated with a particular RP-OIDC. If the user consents they enter their credentials and other authentication factors into the web page.

**4. Authentication Response:** If the user successfully authenticates, the OP then verifies that the `redirectURI` specified in the Authentication Request is on the allow list of redirect URLs for the client-id given. If `redirectURI` is on the allow list then the OP performs an Authentication Response. To do this the OP first generates an Auth-Code and records the user, `nonce` and `CC` associated with that Auth-Code. Then the OP transmits that Auth-Code to the Client Instance by redirecting the user's web browser to the `redirectURI` and specifying the Auth-Code as a parameter in that URI.

**5. Token request:** The Client Instance receives Auth-Code when the user's web browser is redirected to the `redirectURI`. This is because the `redirectURI` is within the origin of the Client Instance and thus triggers a javascript event which informs the Client Instance of the Auth-Code. The Client Instance then sends the Auth-Code along with the Code Verifier (CV) to the OP's Token Request Endpoint. Typically this is done by the Client Instance making a behind the scenes HTTPS request directly to the OP's Token endpoint.

**6. Token Grant:** The OP verifies that the Code Verifier (CV) sent by the Client Instance is the SHA256 preimage of the Code Challenge (CC) sent in the Authentication Request, $\texttt{CC} = \texttt{SHA256}(\texttt{CV})$, that the Code Challenge (CC) corresponds to the Auth-Code, and that the Auth-Code is valid. If all of these checks pass, the OP responds to the Token Request with an ID Token and Refresh Token.

**7. Token Validation:** The Client Instance validates the tokens it received from the OP. The Client Instance checks that the claims in the ID Token match the claims the Client Instance expected. Specifically it checks that the `nonce` the Client Instance generated matches the value in the ID Token's `nonce` claim, that the OIDC-RP's client-id is in the `aud` claim and that the `iss` claim matches the issuer identity for the expected OP. Note the Client Instance does not check that the `sub` claim matches the user's identity because the Client Instance does not know the identity of the user except through the ID Token.

**8. PK Token Creation:** The Client Instance creates a protected header, $\mathrm{h}_u$ consisting of the values in the `cic`.

$$\mathrm{h}_u \leftarrow \texttt{cic}$$

and signs the ID Token under the user's signing key, $SK_u$.

$$\sigma_u \leftarrow \texttt{SIGN}(SK_u, (\texttt{idT}.\text{claims}, \mathrm{h}_u))$$

The Client Instance then constructs the PK Token by adding $(\sigma_u, \mathrm{h}_u)$ to the signature list of the JWS representation of the ID Token.

If any of these steps fail, the Client Instance aborts and deletes all values computed for this session including the tokens. Otherwise the Client Instance accepts the ID Token uses it to authenticate the user's identity to the intended audience.

## E Load Balancers and OpenPubkey

In this section we specify a design for deploying OpenPubkey as an authentication mechanism for users of a modern scalable web application. That is, the user's Client Instance instance is a web browser-based and it is using OpenPubkey to authenticate communication with the web application. We begin by introducing the setting and our high level design. Then we take a closer look and discuss where we adapt our protocols to accommodate the load balancing architecture used by scalable web applications.

Consider a user Alice who employs a web browser-based OpenPubkey Client Instance loaded from `example.org` to securely authenticate and communicate with a web application also hosted at `example.org`. OpenPubkey's web authentication would work as follows:

1. Alice visits `example.org`, this loads the Client Instance which checks if Alice is signed in and has a PK Token. As Alice is not signed in, Alice and her Client Instance run the OpenPubkey protocol to create PK Token. More specifically what happens is:

    (a) The Client Instance generates a new non-extractable user key pair, $(PK_u, SK_u)$, puts public key of this key pair in the `cic`, and initiates the Authorization Code Flow with the OP.

    (b) Alice Consents and Authenticates, the Client Instance is granted an ID Token by the OP. This ID Token becomes the PK Token.

2. As Alice's Client Instance now has a PK Token, the Client Instance authenticates to `example.org` using

OpenPubkey PoP Authentication (Section 3.5.4). To send the PoP challenge value, *ra*, `example.org` creates a web browser cookie with the name `ra-cookie`. After authenticating Alice is now signed into `example.org`.

3. Each time Alice's Client Instance wishes to make an authenticated request to `example.org`, the Client Instance performs a OpenPubkey PoP Authentication in the request using the value `ra-cookie` as the challenge *ra*. The server receiving the authenticated request at `example.org` request, performs the OpenPubkey PoP authentication verification.

In the next section we are going to look how compute this *ra* value in the `ra-cookie` in a modern web application in which there are many servers.

## E.1 Computing the `ra-cookie`

Our OpenPubkey PoP Authentication, in Section 3.5.4, assumes that the Client Instance is authenticating to a specific party acting as the verifier. It is this verification party that chooses the challenge *ra* and checks that this challenge *ra* is returned in the response. To achieve scale and handle a large volume of requests, web applications often stochastically[9] load balance requests across a large pool of servers. This presents a difficulty because the server that sets an *ra* in a `ra-cookie` may not be the same server that is will be verifying the request. How can the verifying server know which *ra* in the `ra-cookie` it should expect, how can it be sure that this *ra* hasn't been used before? In the proceeding paragraphs we discuss this problem in more detail and show how we can adopt OpenPubkey PoP authentication to overcome this difficulty while allowing OpenPubkey to be easily deployed within scalable architectures currently in use.

To illustrate the problem, let us consider the following "obvious" approach and the problems it creates: The *ra* value in the `ra-cookie` is changed after each authenticated request made by the Client Instance.

The first problem is that such an approach creates is that it prevents the Client Instance from making requests in parallel. The Client Instance must wait for the server to set a new *ra* value in the `ra-cookie` before making a another request. This forces the Client Instance to perform only one request at time, and would introduce unacceptably high latency in complex web applications where Client Instance is likely to make a large volume of parallel requests. This is a solvable problem, for instance by setting more than one *ra* value in the `ra-cookie` or using the `ra-cookie` as a seed to a hash function let the Client Instance generate new *ra* values as it needs them. We choose not to use such a solution because it doesn't address the second problem.

The second problem a much harder to overcome obstacle. If server in a pool of servers receives a request with a particular *ra* value how does it know this *ra* hasn't been used before in a request to another server? That is, how does it prevent a malicious party from using *i.e.,* replaying, the same *ra* twice? A malicious party could make two parallel requests to the different servers with the same *ra* at exactly the same moment. The servers in the pool could solve this by having shared state across the pool. This state would record *ra* values that have already been used. While in theory this solves the problem, the performance cost of a transactional write per authenticated user requests would be significant and would necessitate the web application to deploy a complex distributed system. For this reason, we do not see this as an effective solution and do not advocate it.

To overcome these two problems in this setting we relax our replay protection. Instead of requiring that a *ra* value has never been used before as we do elsewhere we instead allow *ra* values that can be used more than once, but these *ra* are limited in use to short window of time. We see this compromise as worth making because we want OpenPubkey to be easy for implementers to deploy in high-scale modern web applications.

To do this, a server when it sets the *ra* value in the `ra-cookie` computes *ra* as:

$$ra \leftarrow \texttt{HMAC}(key_{ra}, \texttt{timestamp}) || \texttt{timestamp}$$

where `timestamp` is the current time in seconds and $key_{ra}$ is a secret key known to all the servers in the server pool. Servers then reject any PoP authentication with a *ra* whose `timestamp` is more than 15 different than the current time. Servers update the *ra* value in the `ra-cookie` on every request. The $key_{ra}$ is used so that servers can authenticate that an *ra* was generated by another server in the pool. This allows servers to reject *ra* values computed by a malicious user as the *ra* value will not be authenticated.

The benefits of this system are that the Client Instance can make as many PoP authenticated requests as it wants in parallel. The signatures of these requests can only be replayed within a 30 second window. The servers that authenticate these requests do not require any shared state, all they require is the secret $key_{ra}$.

If an attacker managed to the learn the secret $key_{ra}$, it would not allow the attacker to impersonate users or break authentication. The security impact would only be that an attacker could compute *ra* futures values. By itself this does not have any security impact. If the attacker also compromised a Client Instance and knew the $key_{ra}$ they could compute a signatures on an *ra* values for a `timestamp` that hasn't happened yet. In the attacker managed to exfiltrate those signatures with future *ra* values off the Client Instance, then they could use them later even if the compromised Client Instance was no

[9] Load Balancing strategies exist in which a user session is bonded to a particular server. This means the user gets the same server over multiple requests. Load balancers configured in this way, can do not require this new approach and can just set many *ra* in the `ra-cookie`.

longer online. The $key_{ra}$ secret is simple to rotate and could be rotated daily or hourly by having all the servers in the pool pull it from a secrets manager.

If an even greater level of security is desired, instead of an HMAC, a signed random beacon could be used to compute a new *ra* every 15 seconds. The signing key used to sign *ra* by the random beacon could then be stored in an HSM (Hardware Security Module) as a non-extractable key. Each server would know the pubkey of the random beacon and could verify the validity of *ra* sent by users. A transparency log of the random beacon could be maintained of all past values and their timestamps. If the servers kept a log of *ra* received, at the end of the day they compare their logs with the transparency log and detect if the random beacons signing key has been compromised. The advantage of this scheme over HMAC is that servers don't need to be trusted with the secret that authenticates new *ra* values.

# F Device Signing Keys and XSS Attacks

In this section we will look at how to constrain an attacker who has compromised a web browser-based Client Instance via a XSS (Cross Site Scripting) attack. XSS (Cross Site Scripting) attacks represent one of the main avenues by which an attacker might compromise a web browser-based Client Instance. As such we wish to carefully consider the ways in which we might counter this threat. To address the threat of a Client Instance that has been compromised by a XSS attacker, we present an extension to OpenPubkey aimed at constraining such an attacker. We do this by introducing a second signing key, which we call the Device Signing Key.

**Threat Model:** We assume an attacker who can compromise the Client Instance at will via a XSS vulnerability, but can not compromise the user's Operating System, the user's web browser. Consequently the attacker can not bypass security boundaries enforced by the web browser including same-origin policy. We do not address the topic of how to prevent XSS vulnerabilities. We also assume that the attacker does not compromise Client Instance the first time it runs on that user's browser.

We now ask the question: If an attacker has compromised a Client Instance with via XSS, how can we constrain the attacker to limit the security impact of such a compromise? Our approach is to make it very difficult for compromised Client Instance instance to read and exfiltrate all the secrets necessary to sign messages as the user. That is, our approach here aims to force the attacker to only be able to impersonate the user via the compromised Client Instance. If the compromised Client Instance goes offline, the attacker can no longer request signatures. To do this we introduce a second signing key held used by the Client Instance called the device signing key and make use of our MFA-Cosigner as an enforcement mechanism to prevent a compromised Client Instance from replacing or regenerating the device signing key.

We note that alternative solution would be to have the Load Balancer perform all the authentication of the OSMs sent. This is very similar to how Load Balancers are used to manage and terminate TLS connections. We do not oppose implementers using OpenPubkey in this fashion but we note that the Load Balancers are now a Single Point of Compromise (SPoC). We have opted to present a more radical and higher security approach, where each server performs verification, to illustrate the power of OpenPubkey to remove Single Points of Compromise.

## F.1 Device Signing Keys Explained

In this section we explain our device signing key design and discuss the security benefits of this feature against an XSS attacker who has compromised a Client Instance. Use of this optional security feature assumes that verifiers requires an MFA-Cosigner. OpenPubkey Web Authentication can be used with a device signing key. Before we explain what the device signing key is and how we use it, we need to explain the problem it solves.

In Section 4.2 we show how to use non-extractable keys to prevent a compromised Client Instance exfiltrating the user's signing key $SK_u$. However the protection offered by that approach is only retroactive, it does not extend to protecting user signing keys generated after Client Instance is compromised. This is because the Client Instance is the party that is trusted to request that the browser generate non-extractable key pair. This means that a compromised Client Instance can simple generate a user signing key as extractable the next time the user signs in and is granted a PK Token. The device signing key is designed to prevent this attack.

The device key pair, $(PK_d, SK_d)$, consists of the device public key, $PK_d$, and device signing key, $SK_d$. This device pair key is generated the first time the Client Instance loads. On creation, the Client Instance immediately sends the device public key, $PK_d$, to the MFA-Cosigner to register it. As part of registration, the MFA-Cosigner sets the `dk-cookie` to the device public key $PK_d$ sent by the Client Instance. The `dk-cookie` is marked by the MFA-Cosigner as an `HTTP cookie` preventing the Client Instance or any other JavaScript running in the web browser from seeing or altering it. As a result, after this point if a compromised Client Instance attempts to replace the device signing key, the MFA-Cosigner will detect this change and refuse to sign the PK Token.

The device key pair generation and registration procedure is as follows:

1. The Client Instance is loaded by the web browser. On loading the Client Instance asks the web browser's crypto API if it has already generated a device key pair. If no device key pair exists for the Client Instance it moves to the next step.

2. The Client Instance asks the web browser to generate a

non-extractable key pair. This key pair will be the device key pair $(PK_d, SK_d)$.

3. Once the Client Instance has generated the device key pair, it registers the device public key, $PK_d$, with the MFA-Cosigner. It does this by making a request to an the Device Registration endpoint at the MFA-Cosigner. Note 0that this is an new endpoint created for the device signing key functionality and not part of the regular MFA-Cosigner protocol.

4. The MFA-Cosigner on receiving the device public key, $PK_d$, checks if the web request included a web browser cookie for the MFA-Cosigner's domain named `dk-cookie`. If this cookie already exists, the MFA-Cosigner aborts and triggers an alarms that an attack may have been attempted. If this cookie does not exist, the MFA-Cosigner sets a cookie named, `dk-cookie`, with the value of the device public key, $PK_d$. The `dk-cookie` is marked as a `HTTP cookie` so that it is only sent to servers and JavaScript running in the browser can not read or alter it. The device signing key has now been generated and registered.

Unlike the user signing key which is associated with the user, the device signing key is associated with the web browser. Multiple users could use the same device signing key if they share the same computer and web browser. When a user signs out, the user signing key is deleted and the user must sign in and authenticate to the OP to be issued a new PK Token with a fresh user signing key. The device signing key is never deleted. Once it is generated it exists in that browser forever (or until the MFA-Cosigner chooses to allow the Client Instance to rotate the device signing key).

If a Client Instance has a device signing key $SK_d$, it associates it with all PK Token it creates by including the device public key $PK_d$ alongside the user public key $PK_u$, in the `cic` in the PK Token. A verifier can then see the public key of the device and the user and require that all OSMs (OpenPubkey Signed Messages) have two signatures; A signature generated by the user signing key and a signature generated by the device signing key.

We modify OpenPubkey PK Token creation to accommodate the device signing key:

1. During Setup the Client Instance includes the device public key $PK_d$ alongside the user public key $PK_u$ in the `cic`. When the OP grants the ID Token, the `nonce` claim will commit to the both the $PK_u$ and the $PK_d$. The Client Instance then transforms the ID Token into a PK Token by signing it with the user signing key and the device signing key, adding two signatures.

2. Use of the device signing key requires a signature from the MFA-Cosigner. The Client Instance runs MFA-Auth with the MFA-Cosigner signing the messages with both the $SK_u$ and the $SK_d$. The MFA-Cosigner checks that `dk-cookie` matches the device public key in the PK Token's `cic` and also verifies the device signing keys signature on the signed messages used to authenticate during the MFA-Auth protocol.

3. If these device signing key used does not match expected device public key specified in the `dk-cookie` the MFA-Cosigner refuses to sign and triggers an alarm that the Client Instance might be compromised.

The security benefit of the device signing key is that a compromised Client Instance can not replace the already generated non-extractable device signing key with a new device signing key and still successfully get the MFA-Cosigner to accept and sign the PK Token. Since the compromised Client Instance can not change the device signing key, the Client Instance can not exfiltrate the device signing key. The compromised Client Instance can exfiltrate the user signing key, but verifiers requiring a device signing key signature will reject messages signed only by the user signing key. Thus we achieve our security goal of forcing the attacker to route all messages they want signed through the Client Instance.

An additional advantage of the use of device signing keys is that it provides an easy place to added a device approval process. For instance `example.org` could require than any web browser that Alice uses must first be approved by an administrator. The device public key could be approved by adding it to an allow list of approved devices. The MFA-Cosigner could then enforce this approval policy by checking if a `dk-cookie` value is on the device allow list.

This security provided by device signing keys has two limitations. Attackers with more powerful capabilities such as the ability to compromise the user's Operating System or Process Memory of the user's web browser can bypass these protections and exfiltrate the device signing keys. The security depends on the fact that an XSS attacker compromises the Client Instance but the browsers security enforcement mechanisms are still intact. Second, if an attacker compromises the Client Instance before the Client Instance has generated and registered the device signing key, then the compromised Client Instance can generate the device signing key as an extractable key. As the Client Instance generates and registers the device signing key in the first few seconds of the first time it is loaded, this presents extremely small window of time for an attacker to compromise it.