# WebAuthn Under Attack: How Trojans Can Compromise Your Credentials

- Aditya Mitra (adityamitra5102@gmail.com)
- Anisha Ghosh (ghoshanisha2002@gmail.com)

## Introduction

Trojan horse is defined as a computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes the program.

FIDO2 is an authentication standard that aims to provide phishing-resistant authentication over cryptographic protocols. The standard uses the FIDO Alliance Client to Authenticator Protocol (CTAP) specification and W3C Web Authentication (WebAuthn) together to bring cryptographic authentication. WebAuthn is claimed to be secure because it uses public-key cryptography to authenticate users. It further verifies the transaction details before signing them and it is compatible with multiple authenticators, both platform and roaming. It is a standard web API that is supported by most modern browsers and platforms.

## Authentication Process

Let us focus on how authentication takes place over FIDO standard. The parties involved are the claimant and the verifier. The verifier is also known as Relying Party (RP) and is responsible for verifying the identity of the claimant based on cryptographic approaches.

Let us assume, the user is already registered with the RP server with an authenticator. This means the claimant already has a keypair, of which the private key is securely stored in his authenticator while the public key and the credential ID is with the RP server.

The claimant first requests to authenticate himself to the RP Server. The RP Server retrieves the credential ids registered for the claimant and creates the 'public key options' that are passed on to the browser of the claimant as a response to an API call over secured context (https).

The options contain the following:

- Challenge: A nonce or cryptographic challenge of size 16 bytes or more.
- RP ID: The ID of the RP Server. The RP ID must be equal to the origin's effective domain or a registrable domain suffix of the origin's effective domain.
- Allow-credentials: This contains a list of the credential IDs of the user. The user can use any one of the credentials from this list.
- User verification: This ensures whether the user must enter some additional credentials apart from just the activation factor of the authenticator. For example, for some authenticators, the activation factor might be just a touch, but the additional credential might be a PIN.

These options are usually sent in CBOR encoding from the RP Server to the claimant's browser, however, it can also be sent in the form of JSON, or any other encoding as required by the implementation. The browser converts it into the following format:

- challenge: ArrayBuffer
- rpId: String
- allowCredentials: array of the following
  - type: String

   - id: ArrayBuffer
   - transports: Array of String
- userVerification: String
- extensions: Map (optional)
- timeout: Int (optional)

For example, when it is generated by Google when attempted to use FIDO Credentials for logging in, the options look like:

```
1.  publicKey:
      1.  allowCredentials: Array(5)
            1.  0: {type: 'public-key', id: ArrayBuffer(32)}
            2.  1: {type: 'public-key', id: ArrayBuffer(32), transports: Array(1)}
            3.  2: {type: 'public-key', id: ArrayBuffer(32), transports: Array(1)}
            4.  3: {type: 'public-key', id: ArrayBuffer(32), transports: Array(1)}
            5.  4: {type: 'public-key', id: ArrayBuffer(64), transports: Array(2)}
            6.  length: 5
            7.  [[Prototype]]: Array(0)
      2.  challenge: ArrayBuffer(6808)
      3.  extensions: {appid: 'https://www.gstatic.com/securitykey/origins.json'}
      4.  rpId: "google.com"
      5.  timeout: 180000
      6.  userVerification: "preferred"
      7.  [[Prototype]]: Object
```

The origin of the request is ‘accounts.google.com’, which is a registrable domain suffix of the RP ID ‘google.com’.

Once the browser verifies the RP ID against the Origin, it calls the navigator.credentials.get() function which is under the Web Authentication API.

The navigator.credentials.get() function passes on the information to the authenticator over CTAP protocol. The authenticator uses the private key against the given authenticator id and the RP ID.

It computes the Authenticator Data which is an array buffer of at least 37 bytes containing the hash of the RP ID (SHA-256 Hash) (32 bytes), flags (1 byte), and a signature counter (4 bytes). The client data is computed as a URL safe Base64 encoding of the challenge received, the origin and whether the request is cross origin. The client data is represented as a JSON in ArrayBuffer, and the hash (SHA-256) is calculated. The Client Data hash is appended to the authenticator data, and both is signed together with the private key in the authenticator.

Usually, the ECDSA algorithm is used to sign the challenge.

It is then returned as a publicKeyCredential object which contains the following:

- authenticatorAttachment: Whether the authenticator used was platform or roaming (cross-platform.)
- rawId: The credential id of the used credential
- id: URL Safe Base64 encoding of the rawId.
- type: The type of credential, usually ‘public-key’.
- response: Object of type AuthenticatorAssertionResponse, holding the following data:
   - authenticatorData: ArrayBuffer of size at least 37 as discussed above.
   - clientDataJSON: The client data as discussed above.
   - signature: The signed client data hash and authenticator data as discussed above.

- userHandle: The user id.

The response on signing the options shown above are:

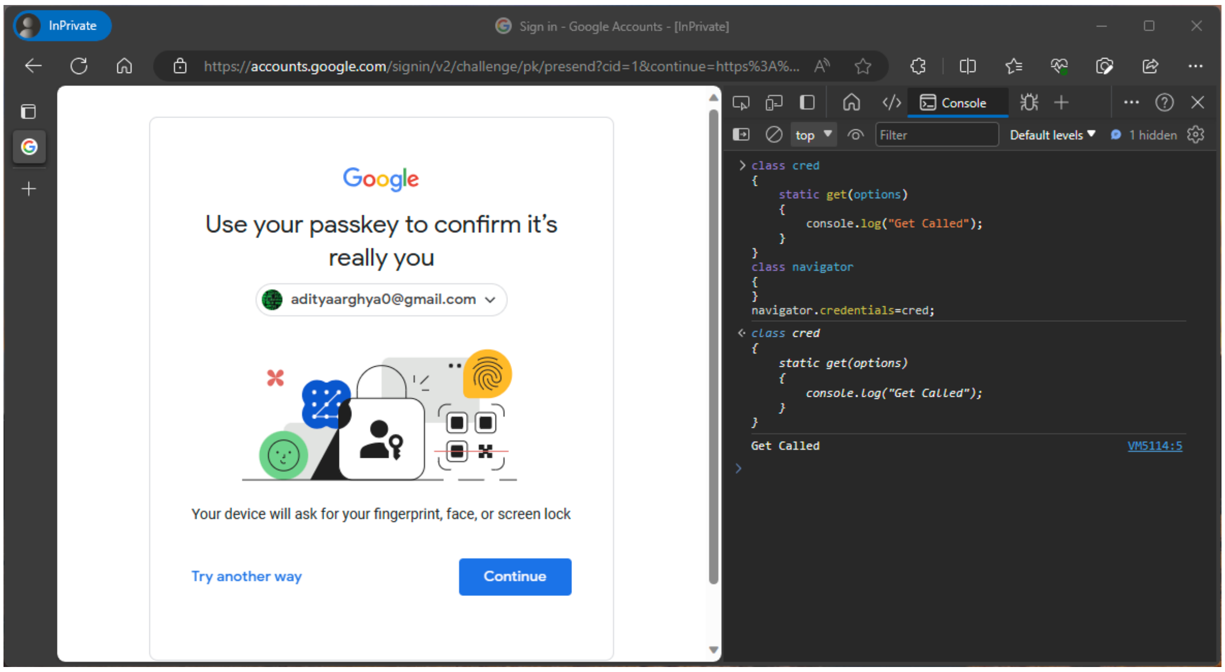
```
PublicKeyCredential {authenticatorAttachment: 'platform', id: 'Rt5CPeeMPxwo_-4XLx0Ftqld-8DdBjZqEkJcH7RK_Bo', rawId: ArrayBuffer(32), response: AuthenticatorAssertionResponse, type: 'public-key'}
    authenticatorAttachment: "platform"
    id: "Rt5CPeeMPxwo_-4XLx0Ftqld-8DdBjZqEkJcH7RK_Bo"
    rawId: ArrayBuffer(32)
    response: AuthenticatorAssertionResponse
        authenticatorData: ArrayBuffer(37)
        clientDataJSON: ArrayBuffer(9415)
        signature: ArrayBuffer(72)
        userHandle: ArrayBuffer(36)
        [[Prototype]]: Object
    type: "public-key"
    [[Prototype]]: Object
```

This response is returned by the navigator.credentials.get() function and is then encoded to CBOR or any other encoding and sent back to the RP Server. The RP Server verifies the authenticator data and client data first. Then, it verifies the signature. If the verifications are successful, the claimant successfully authenticates.

**Vulnerability assessment:**

However, web authentication, in the end is a JavaScript API and still follows all the rules of JavaScript. This includes features like Function Overriding. This implies, if we push a custom implementation of the navigator.credentials.get() as a XSS (Cross Site Scripting), it can override the native implementation of the same function and the WebAuthn calls will not function as it is supposed to. This can be verified by simply entering the following code into the console window before making a WebAuthn call:

```
class cred
{
	static get(options)
	{
		console.log("Get Called");
	}
}
class navigator
{
}
navigator.credentials=cred;
```


InPrivate
Sign in - Google Accounts - [InPrivate]
https://accounts.google.com/signin/v2/challenge/pk/presend?cid=1&continue=https%3A%...
Console
top
Filter
Default levels
1 hidden

Google

Use your passkey to confirm it's really you

adityaarghya0@gmail.com

Your device will ask for your fingerprint, face, or screen lock

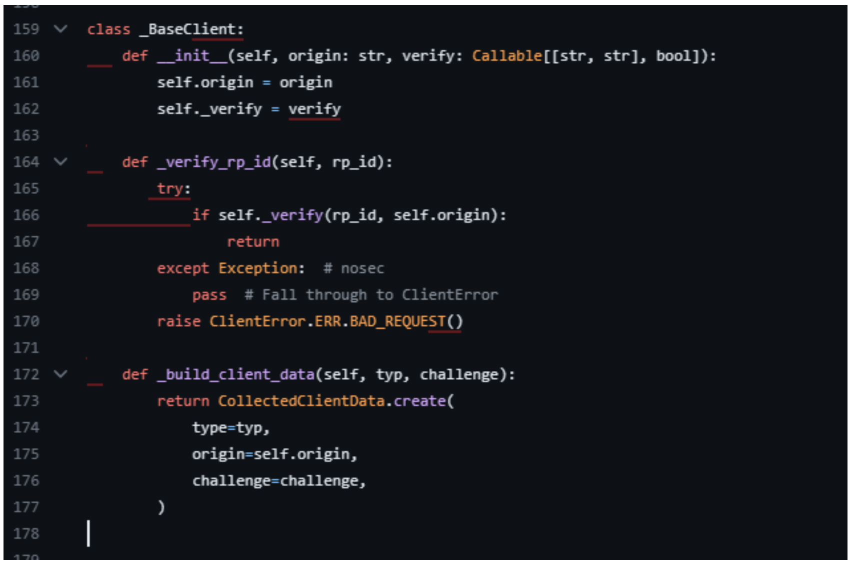
Try another way

Continue

```
> class cred
  {
      static get(options)
      {
          console.log("Get Called");
      }
  }
  class navigator
  {
  }
  navigator.credentials=cred;
<· class cred
  {
      static get(options)
      {
          console.log("Get Called");
      }
  }
  Get Called                                        VM5114:5
```

The adversarial actor can perform a Self-XSS on his own system to modify how WebAuthn works for him. He can then capture the 'options' and send the same to the legitimate user to sign. One major issue that might arise here is that even if the client is on a phished webpage, the RP ID verification fails. This is because the challenge arises from the legitimate RP Server and must be signed by the keys enrolled against that RP Server only.

This is overcome by not directing the legitimate user to his web browser. A trojan on the legitimate user's system might be able to access the Authenticator while impersonating the RP Server without verification. The trojan uses the python implementation of the FIDO2 library. This choice can be justified with a better look at the client.py file in the FIDO2 library.



```python
class _BaseClient:
    def __init__(self, origin: str, verify: Callable[[str, str], bool]):
        self.origin = origin
        self._verify = verify

    def _verify_rp_id(self, rp_id):
        try:
            if self._verify(rp_id, self.origin):
                return
        except Exception:  # nosec
            pass  # Fall through to ClientError
        raise ClientError.ERR.BAD_REQUEST()

    def _build_client_data(self, typ, challenge):
        return CollectedClientData.create(
            type=typ,
            origin=self.origin,
            challenge=challenge,
        )

```

The _verify_rp_id() function cryptographically verifies the RP ID against the origin from which the function is called. However, the origin can be specified as a string for the library, as seen in the __init__ constructor.

Hence, the attack model might be, the adversarial actor runs a Self-XSS to extract the 'options' sent by the RP Server. These options and the origin are now sent to Trojan running on the user's system.

The trojan invokes the authenticator and might require user verification with Fingerprint or a Pin. The legitimate user is assumed to touch his fingerprint sensor or enter the pin as a part of a sophisticated social engineering attack. The signed challenge, in the form of the PublicKeyCredential is sent back to the adversarial actor, who then returns the same to the custom implementation of navigator.credentials.get(). It is then returned to the RP Server from the system of the adversarial actor. Thus, the RP Server authenticates the adversarial actor as a legitimate user.

It is to be noted that the custom implementation of navigator.credentials.get() needs to convert the ArrayBuffers in the 'options' to arrays and then to JSON so that it can be sent to the applications running on the machine of the adversarial actor. It also needs to encode the response back to the desired formats.

**Attack scenario**

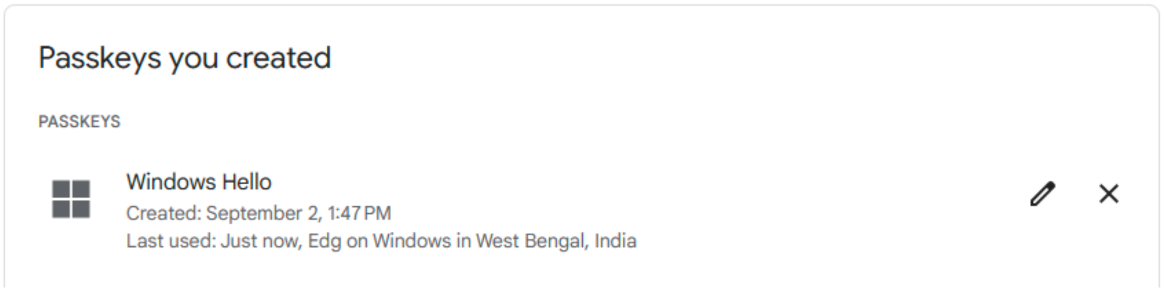
For the attack scenario, the adversarial actor has a cloud server that is used only to exchange the data between the system of the adversarial actor and the trojan. Any other messaging client could also be used instead of this cloud server. The adversarial actor further runs an application on his machine. The application first copies the script to be injected into the clipboard so that the adversarial actor can paste it in his browser console. The application then captures the 'options' from the malicious navigator.credentials.get() implementation and encodes it into Base64 and then sends it to the cloud server. The trojan running on the system of the legitimate user requests this Base64 data from the cloud server. It decodes the Base64 into proper WebAuthn format, performs the challenge signing and then encodes the results back into Base64 and sends it back to the cloud server. The cloud server returns the response back to the adversarial actor's system. The application decodes the Base64 and returns it to the malicious implementation of navigator.credentials.get(). The implementation then decodes it back to ArrayBuffers and returns it to the RP Server.

The codes used for the following is available here:

- Cloud side:
  - https://adityamitra5102.github.io/fidovulntest/cloud/flaskapp.py This application is served by Apache Webserver in the cloud.
  - https://adityamitra5102.github.io/fidovulntest/cloud/requirements.txt This is the dependencies for the application.
- Adversarial actor side:
  - https://adityamitra5102.github.io/fidovulntest/attacker/flaskapp.py This application runs on the machine of adversarial actor so that the malicious implementation can communicate to localhost:5000.
  - https://adityamitra5102.github.io/fidovulntest/attacker/inject.js This is the Self-XSS code to be pasted in the browser console before WebAuthn call.
  - https://adityamitra5102.github.io/fidovulntest/attacker/requirements.txt This is the dependencies for the adversarial side application.
- Trojan
  - https://adityamitra5102.github.io/fidovulntest/malware/malware.py This is the basic implementation of the Trojan.
  - https://adityamitra5102.github.io/fidovulntest/malware/requirements.txt This is the dependencies for the implementation of the Trojan.

The repository containing the code is not made public for obvious reasons. But the codes can be accessed from the above links.

The attack scenario was tested, and the adversarial actor was able to take over the Google account of the legitimate user in the process. The Google account was secured with Passkeys with Windows Hello.


Passkeys you created

PASSKEYS

Windows Hello
Created: September 2, 1:47 PM
Last used: Just now, Edg on Windows in West Bengal, India

Full video of the case study is available here (unlisted video): [https://youtu.be/Lh45VzZqQ60](https://youtu.be/Lh45VzZqQ60)

**End note**

This case study and related information is only for educational purposes and not for exploiting accounts without proper authorization.