A Security Enhancement to the FIDO2 Protocol

Project Submission by: Kiley Carson
Student ID: 30009983

Supervised by: Dr. Rei Safavi-Naini

Due 12-04-2022 @ 23:59

A Final Report presented for the degree of
Bachelor of Science Honours (Computer Science)

Computer Science
University of Calgary
Canada
12-04-2022

Introduction

FIDO2 is the Fast IDentity Online (FIDO) Alliance's proposed standard for passwordless user authentication; the goal was to be both more secure than password (with MFA) authentication and easier to use. The standards defined formalize the World Wide Web (W3C) Consortium's Web Authentication (WebAuthn) and the FIDO Alliance's Client-to-Authenticator Protocol v2.0 (CTAP2). Understanding the FIDO2 authentication system; its cryptographic building blocks, security model and proof are all goals of the project. Additionally, the deliverables include a design of a security enhancement to the FIDO2 protocol using One-Time Passwords (as a form of Multi-Factor Authenticator), a proof of concept implementation of the design and an evaluation. Therefore, the relevant work in this project includes the "Provable Security Analysis of FIDO2" [1] and the specifications of both the WebAuthn [2] and CTAP2 sub-protocols [3]. Using One Time Passwords to introduce independent binding states, a modified design of the CTAP2 protocol that solved the security flaw discovered by Barbosa, Boldyreva, Chen, *et al.* [1] was achieved. Some time evaluation was done, but most of it was left for Future Work due to resource constraints.

Background

Provable Security of FIDO2

Four main things are done in the paper by Barbosa, Boldyreva, Chen, *et al.* [1]; general information about the FIDO2 protocol is provided, a modular-style security analysis of the protocol is completed, flaws are identified, and lastly, improvements for a stronger protocol via modifications of the CTAP2 protocol are suggested. This paper is the first provable security analysis of the new FIDO2 protocols for a standard of passwordless user authentication using the Bellare-Rogaway Model [4], a communication model that uses oracles for distributed security. The analysis covers the core components of FIDO2 - W3C's Web Authentication (WebAuthn) specification and the new Client-to-Authenticator Protocol (CTAP2). Strong adversarial capabilities are captured in the analysis as it completed using a computational model in the provable security approach. During the analysis "Strong Unforgeability" and "Unforgeability with Trusted-Binding" was defined as and discussed in the context of the trust implications of the client and the authenticator. These definitions allowed for the security goals to be considered in both the scenario where there are and where there are not assumptions of trust made. This revealed that an attack was possible in the CTAP2 protocol [1].

FIDO2 Subprotocol: WebAuthn

In *Web Authentication: An API for accessing Public Key Credentials - Level 2* [2], the specifications of the one of the cryptographic building blocks, Web Authentication (WebAuthn), of the FIDO2 system is provided. Since one of the goals of the project is to understand the cryptographic building blocks of FIDO2, these specifications are relevant to the project. An API that enables the creation and use of strong, scoped, and verified public-key credentials by web applications in order to authenticate users is defined in this specification. The web application requests authenticators to create and bind one or more public key credentials, each scoped to a given WebAuthn Relying Party. In order to upload user privacy, the user agent has control over access to authenticators and their public key credentials. Authenticators are responsible for ensuring that no operation is performed without user consent. Authenticators provide cryptographic proof of their properties to Relying Parties via attestation. The presence of a *Security Considerations* section in the WebAuthn specifications is a strong quality of this document. It contained a description about how to design a system where an authenticator does not need to be physically close to the client or where the authenticator and client do not communicate directly. The specifications state that physical proximity as a key strength for "something you have" and that designing a solution without this requirement would require consideration of the strength of the authenticator. The section also spoke to how the authenticator provides key management and cryptographic signatures and the fact that it could be a separate device or embedded in a WebAuthn client. However, the specifications do not have security models and proofs that would be needed in order to verify the security of the protocols.

FIDO2 Subprotocol: CTAP2

One of the cryptographic building blocks of the FIDO2 system is the FIDO Alliance's Client to Authenticator (CTAP) Protocol v2; CTAP2 is the protocol that project will be focusing on since the design will be working to improve the vulnerability identified by Barbosa, Boldyreva, Chen, *et al.* [1]. these specifications are therefore relevant to the project. Two CTAP protocol versions (CTAP1/UAF and CTAP2) are referred to in the document, specifically an application layer protocol for communication between a client/platform and a roaming authenticator. Additionally, the bindings between this protocol and other transport protocols using different physical media forms are described and requirements are defined. Based on the requirements of the application layer protocol, each transport binding defines how a transport layer connection should be created. The project will be focusing on the CTAP2 version as authenticators implementing CTAP2 are referred to as FIDO2 authenticators. These specifications do not explicitly include the security considerations for the protocol, nor security models and a proof, rather a reference to the FIDO Security Reference Document [5] is attached at the bottom of the specification. In that document, the CTAP2 protocol is discussed with security threats such as a malicious device with direct communication access to FIDO Authenticator, a hostile or compromised ASM/FIDO Client or sniffing occuring between ASM/-FIDO Client and the Authenticator. Consequences and possible mitigations are provided. Additionally, a note exists in the specification stating that "for other requirements than those specified in this specification[,] for example... security and privacy requirements ... [one] can refer to the applicable certification documents (e.g. the FIDO Alliance, FIPS, Common Criteria, etc)". [3]

FIDO2 Security Proof

Several flaws were identified in the first provable cryptographic analysis of the authentication properties guaranteed by FIDO2 [1]. A corruption model with security notions ranging from Strong Unforgeability (SUF) to Unforgeability with Trusted-Binding (UF-t) was used to do a modular security analysis of the FIDO2 protocol [1]. The result was that the composed protocol of WebAuthn+CTAP2 (FIDO2) was able to achieve user authentication security guarantees under the weakest corruption model (UF-t) [1]. Specifically, CTAP2 cannot achieve Unforgeability (UF) security because all access channels between the client and the token, share the same binding state (the pinToken PT), and this means that if any of these channels are compromised by an attacker then the remaining clients are no longer secure [1]. The binding state uses a keyed-hash on the userPin to create the pinToken. The key used is the shared key derived from the Diffie-

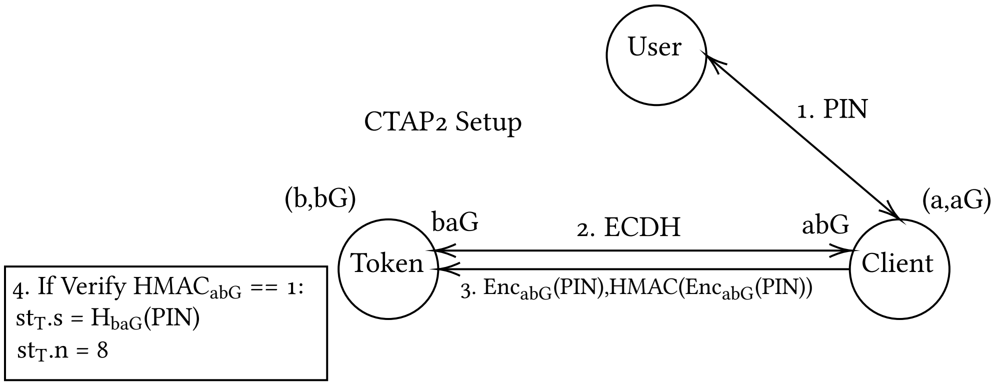

User
1. PIN
CTAP2 Setup
(b,bG)
(a,aG)
baG
2. ECDH
abG
Token
Client
3. Enc$_{abG}$(PIN),HMAC(Enc$_{abG}$(PIN))
4. If Verify HMAC$_{abG}$ == 1:
st$_T$.s = H$_{baG}$(PIN)
st$_T$.n = 8

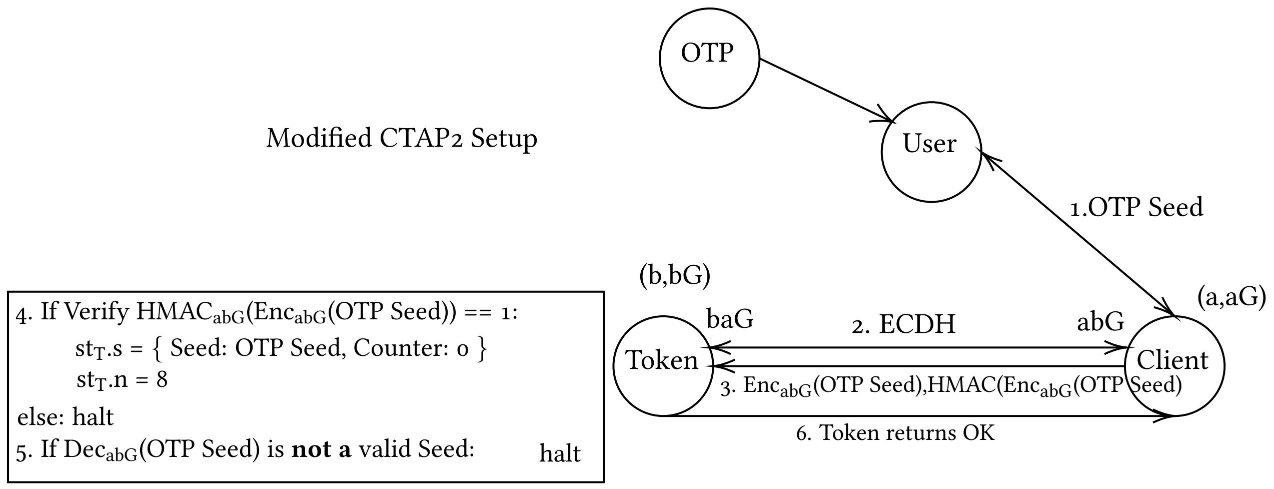
Figure 1: CTAP2 Setup Protocol

Hellman Key Exchange. To achieve stronger security all the binding states must be independent from each other. Then even if only access channel is not compromised, it is still possible to maintain its security [1]. Additionally, unauthenticated Diffie-Hellman Key Exchange (DHKE) is vulnerable to Man-in-the-Middle attacks [1].
Suggestions for improvement were also identified. In CTAP2, a protocol change is suggested in the binding phase to achieve stronger security. Specifically, replacing unauthenticated DHKE with a password-authenticated key exchange (PAKE) protocol as PAKE protocols take as input a common password and output the same random session key for both parties [1]. A proposal for a generic protocol called "secure Pinbased Access Control for Authenticators" (sPACA) and a proof of its strong security was provided [1]. sPACA was also proven to be more efficient than CTAP2, and the authors "advocate[d] the adoption of their protocol as a substitute for both stronger security and better performance" [1].

Specification & Design

Due to the security flaws identified in the CTAP2 protocol, the two sub protocols in the CTAP2 protocol that will be modified as a result of the security enhancement are the setup protocol (Figure 1) and the bind protocol (Figure 3). The more secure design proposes the substitution of a one-time password instead of a user PIN, and the addition of an external application synchronized with the token that the user would use for generating the one-time password. The addition of a one-time password application, is beneficial for solving the security flaw that was mentioned by Barbosa, Boldyreva, Chen, *et al.* [1], because it makes the each instance of the binding state independent. Previously, the binding state ($st_T.bs_i$) was the Pin Token from the Token's longer term storage (Figure 3). The modification changes ($st_T.bs_i$) to be the value of a One Time Password generated using a Seed and Counter from the long term storage (Figure 4). The modification also includes the substitution of a seed, counter, and Linear Congruential Pseudo Random Number Generator in the token (PRNG), instead of a user Pin.

The entities involves in the regular CTAP2 protocol include the client, user, and the token (Figure 1 or Figure 3). The modified version of the protocol includes the addition of a new entity that generates One Time Passwords, a One Time Password application (Figure 2 or Figure 4). The client in the sub-protocols is the browser and the device that the user is using. Since the goal of CTAP2 is bind the token to the



```
4. If Verify $HMAC_{abG}(Enc_{abG}(OTP Seed))$ == 1:
        $st_T.s$ = { Seed: OTP Seed, Counter: 0 }
        $st_T.n$ = 8
else: halt
5. If $Dec_{abG}$(OTP Seed) is **not a** valid Seed:        halt
```

Figure 2: Modified CTAP2 Setup Protocol

device, a requirement/assumption for protocol to proceed is that the user is using a trusted client. This is demonstrated by the first step in the flow of the simulation of modified CTAP2 being a prompt for the user to confirm that they are using a trusted device. The client also relays the communication between the user and the token; therefore, it needs to be able to send messages back and forth between the token and the user. The OTP Application is also a separate application by itself - it has its own interface that shows the user the number being generated for each ieration of the binding protocol. OTP Application and the external Token in the system do not need to directly talk to each other. They communicate through the User and the Client. It is assume that they are synchronized and they have to be able to share a seed. If a Time-Based One Time Password is used, that requires the Token to have access to real-time-based algorithms and time synchronization, which may require the token to have it's own source of computing power. Instead for simplicity, the design will use a Linear Congruential Number Generator with a Seed will increment based on a counter with each binding session. If we use counters, both could be on input number five for the algorithm, then both use output pseudo random number five for the One Time Password.

The Goal of CTAP2 is to get the token to bind to a trusted client so that the user knows messages relayed (received/transmitted) by the client are coming from a trusted source. The sequence of interaction for the simulation of the CTAP2 protocol is that CTAP2 begins with the setup sub-protocol (Figure 1 or Figure 2) since a brand new token does not have the seed embedded in it. CTAP2 setup starts with the sharing of the SEED, this means that the SEED is firstly embedded in the One Time Password Application and in the external token. The user would open the browser page, and then the process would begin, and then the user would enter the PSN (from the OTP application, assuming the SEED was already embedded), and then the process would continue. It begins with the user providing their PIN to the client from the external application

After the FIDO2 token is properly setup, the CTAP2 sub-protocol for binding can proceed. So the very first thing is that the user is going to initiate the client and token binding At this stage, the pin is used/or the OTP because in the current FIDO2 model the external token has the Seed embedded in it So now the user is demonstrating knowledge of this OTP based on the Seed through the client to show that

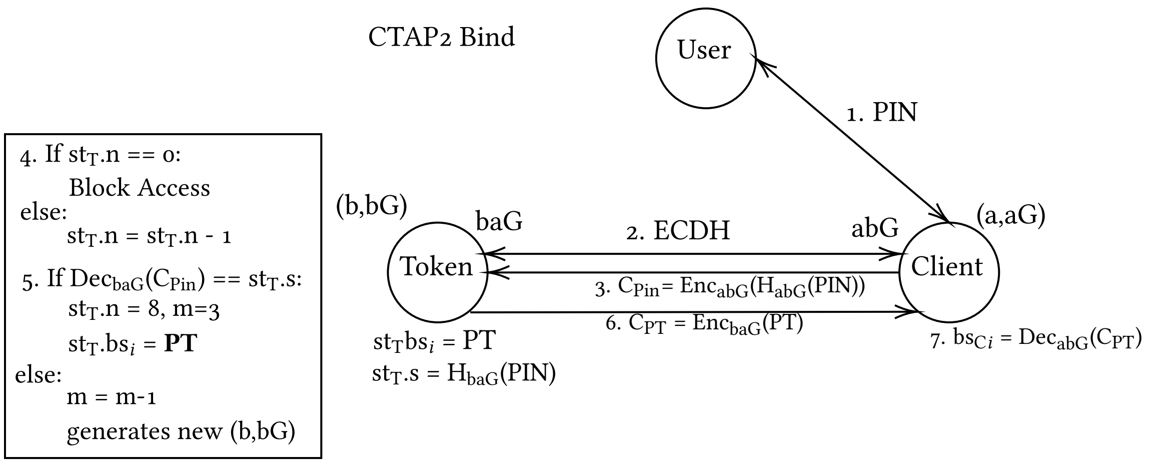


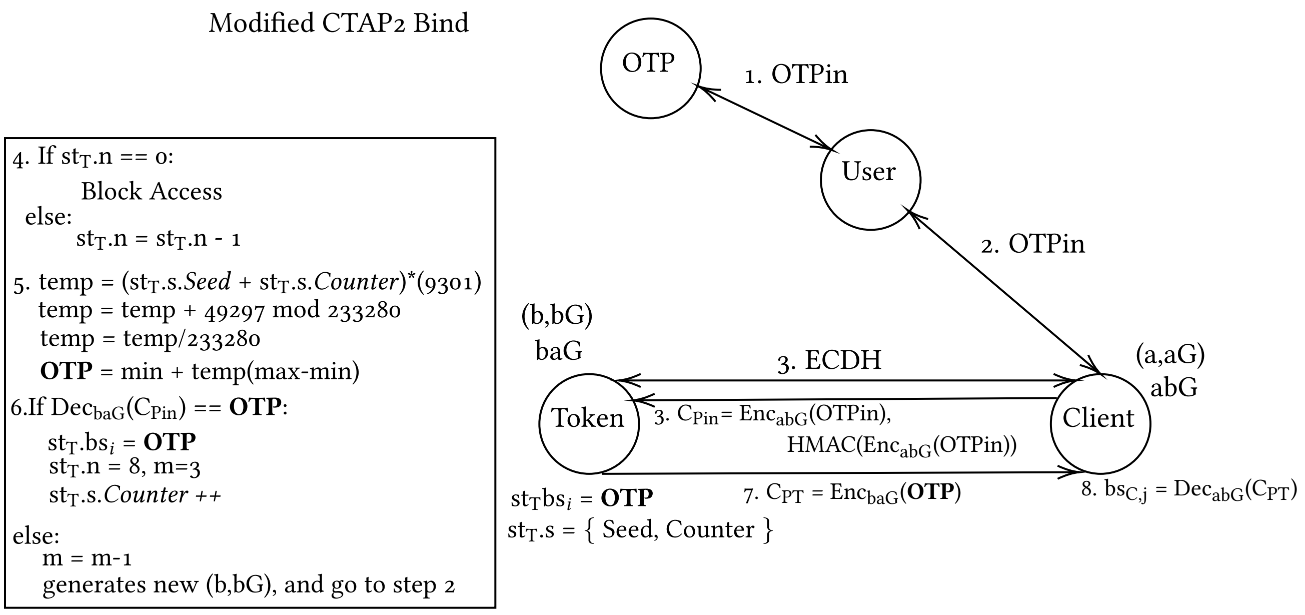
Figure 3: CTAP2 Binding Protocol

the client is trusted The binding process is a way of the user ensuring that the client is trusted, it is done by ensuring that the client and the token communicate in the expected and desired way The binding process is done by the user inputting the PSN into the client, and creating a communication channel between the client and the token that should also be able to derive the same PSN. So if the communication channel (Elliptic Curve Diffie Hellman based on the PSN) can happen, then the user can confirm that the client is trusted After this, a secure binding state between the token/client/user is created The modified CTAP2 simulation is begins with the User providing their one time password to the client, which is relayed to the token (Figure 4). Then the client begins the Elliptic Curve Diffie Hellman protocol with the external token Now the user who wants to visit the website communicates through the client with the token to start the webAuthn (authentication with the server) the token with the binding state can now authenticate the messages received by the client during the webAuthn processes WebAuthn fully completes after CTAP2 binding protocol finishes.

Implementation

The simulation of the CTAP2 system is implemented in React (JavaScript). My implementation consists of the following components; a Trust Prompt, Setup Prompt, Binding Prompt, OTP Display, Authentication Screen. These components map to the entities of the CTAP2 algorithm in the following way: The OTP Display simulates the OTP application, Setup Prompt and Binding Prompt simulates the Token, the remaining components simulate the Client, and the actions performed to flow through application/implementation simulate the behaviour of the entity of the user. Additionally, in my implementation the storage of the data is divided into 4 separate objects (user, client, token and OTP) to simulate that the actions occurring in the implementation affect different entities.

The flow of the simulation starts with User confirmation of trust in the device/client/system (Figure 5). This prompt exists because because the client actually has to be a trusted client meaning that the browser and the accompanying device has to be trusted by the user (i.e. there is no malware on the device) in order


Modified CTAP2 Bind
OTP
1. OTPin
User
2. OTPin
(b,bG)
baG
(a,aG)
abG
Token
Client
3. ECDH
3. C_{Pin}= Enc_{abG}(OTPin), HMAC(Enc_{abG}(OTPin))
7. C_{PT} = Enc_{baG}(**OTP**)
8. $bs_{C,j}$ = Dec_{abG}(C_{PT})
$st_T bs_i$ = **OTP**
st_T.s = { Seed, Counter }

```
4. If $st_T$.n == 0:
        Block Access
   else:
        $st_T$.n = $st_T$.n - 1
5. temp = ($st_T$.s.Seed + $st_T$.s.Counter)*(9301)
   temp = temp + 49297 mod 233280
   temp = temp/233280
   OTP = min + temp(max-min)
6.If $Dec_{baG}(C_{Pin})$ == OTP:
     $st_T.bs_i$ = OTP
     $st_T$.n = 8, m=3
     $st_T$.s.Counter ++
else:
    m = m-1
    generates new (b,bG), and go to step 2
```

Figure 4: Modified CTAP2 Binding Protocol

for the CTAP2 protocol to work since the client is relaying messages between the Token entity and the User entity.

Token Setup Flow

The user then moves through the Token Setup Flow, because it assumed that every execution of the simulation starts with a fresh token. The Token Setup Flow starts with the prompt for a seed for the Token through the Client (Figure 6), and in addition to entering a chosen seed the User also has to input that chosen Seed into the One Time Password Application (Figure 7). After the user has inputted the seed in the One Time Password Application, and inputted the seed in the Client, the setup cryptographic flow between the client and token begin (Figure 8). The setup cryptographic flows use the SubtleCrypto interface of the Web Crypto API.

In Figure 8, it is visible that following the CTAP2 protocol, the Client sends to the Token "cmd=2", then the Token creates it's Elliptic Curve Diffie Hellman (ECDH) private and public key pair using SubtleCrypto.generateKey(), using a 384 bit curve (P-384). The token then sends it's public key with the generator to (aG) to the Client. The Client then uses it's private key (b) in combination with (aG) that it received from the token in SubtleCrypto.deriveKey(), to derive the shared key between the Client and Token. The shared key between the Client and the Token is an Advanced Encryption Standard-Galois Counter Mode key with length 256 bits. The Token follows a similar behaviour but using (bG) which it received from the Client and (a) which was it's private key. After the shared key has been derived the Client can use SubtleCrypto.encrypt(), to encrypt the Seed that was inputted by the User into both the One Time Password Application (Figure 7) and the Prompt on the Client (Figure 6) and send it to the token. In addition to sending the encryption of the Seed to the token, the client also, signs the ciphertext, using the same key

SubtleCrypto.sign(), to create an HMAC of the encryption of the Seed, and sends that to the Token. The algorithm used for the HMAC of the encryption of the seed was SHA-512. In the simulation implemented, the SubtleCrypto library restricted what the functionalities the keys were allowed to have, so if the initial shared key was not declared to have the ability to sign/verify, a second key (from the shared parameters), had to be derived. Therefore, in the simulation of the modified version of the CTAP2 protocol, a second key exists for the HMAC portion of the protocol that is also derived from the parameters that the Token and the Client share. After receiving the Seed, the Token will decrypt it, using SubtleCrypto.decrypt() , verify the HMAC using SubtleCrypto.verify(), and then will embed the Seed and a counter initialized to 0 in it's binding state.

Token Binding Flow

After the Token has been successfully set up, the user has the option to bind their token to the trusted client (Figure 9). After, the user clicks bind, they are prompted to provide their One Time Password (Figure 10, which they can access through their One Time Password Application (Figure 11). After the user inputs the One Time Password that they received from the One Time Password application, the cryptographic portion of the protocol begins. If an invalid pin (Figure 12) is entered - either incorrect syntactically, or not the One Time Password the Token was expecting) - an "Invalid Pin" message will be displayed. The cryptographic portion of the protocol will also fail - this is visible in Figure 13.

If a valid One Time Password is inputted by user instead, the cryptographic portion of the protocol begins, and the token binds to the client. Similar to the Setup, the cryptographic flows begin with the Client sending the message "cmd=2" and the Token responding with a public key (aG). However in the binding, the next message that the client sends is "cmd=5" (Figure 14) whereas in the setup the next message was "cmd=3" (Figure 8). A shared key (abG) or (baG) is again derived next, using SubtleCrypto.deriveKey(), and the One Time Password that was inputted to the Client is encrypted, sent to the Token, and then decrypted. In comparison to the setup protocol, an HMAC is not used here, because if the data is corrupted in transit, then the One Time Password that was inputted would not match the One Time Password that was expected, and therefore, the user would be expected to reinput the One Time Password anyways, similar to what would happen in Figure 13.

```
function OTPGenerator(seed,counter){
  Math.updated = seed+counter
  const max = 999999;
  const min = 100000;

  var rnd = (Math.updated * 9301 + 49297) % 233280;
  rnd = rnd / 233280;

  return Math.floor(min + rnd * (max - min));
}
```

Listing 1: Linear Congruential Pseudo-Random Number Generator (PRNG)

After the Token receives and decrypts the One Time Password from the Client, it has to also generate the expected One Time Password using the seed and counter stored in it's binding state. The OTPGenerator() function in Listing 1 is used with the seed and counter to generate the expected One Time Password. If the decrypted and the expected One Time Password match, the Token encrypted and returns the expected

One Time Password to the Client that it derived using the OTPGenerator() function, to create a shared binding state. Listing 1 displays the Linear Congruential Pseudo-Random Number Generator (PRNG) that is used for generating the One Time Passwords in both the One Time Password Application (Figure 11) and in the Token (Figure 4, Step 5). For well chosen parameters, the output of the function OTPGenerator() will have statistical uniformity (all possible values from 0 to the modulus m are possible) [6]. The cycle for this linear congruential PRNG is of size 233280. The modulus, m is 233280, and it has a prime factorization = ($2^6 x 3^6 x 5^6$). The additive constant c is 49297 (a large prime). Therefore, since c is a prime, the modulus and the additive constant are relatively prime because the only common prime factor between m and c is 1. Additionally, the multiplicative constant is 9301. The multiplicative constant minus one is 9300, and the prime factorization of 9300 = ($2^2 x 3 x 5^2 x 31$), therefore it is divisible by all the prime factors the modulus m. Lastly, since the modulus was divisible by four, the multiplicative constant minus one, 9300, also had to be divisible by four [6] and it is.

Evaluation and Results

There are two meaningful components to the CTAP2 protocol that can be measured and evaluated - the time and the added security. Extra security was introduced in the modified design with the addition of the unique one time password, and the linear congruential pseudo-random number generator. The benefits of these features is that the problem with each instance of the CTAP2 binding state not being independent is resolved. With independent binding states, an adversary can no longer take the information that they obtained from a previous binding state and use it to exploit a fresh binding state - therefore, this feature strengthens the security of the protocol. However, feasibility issues discovered with the synchronization between the One-Time pad and the token. The modified design allows for two different seeds to be embedded in the token and the One Time Password application. A redesign of the setup subprotocol would be necessary to mitigate this problem, and has been left as Future Work.

Since there are additional steps in the modified CTAP2 Setup/Binding protocol, it is expected that it will take longer than an implementation of the original protocol. No time measurements have been taken for the original CTAP2 protocol, however Figure 16 shows the completion time in milliseconds for ten iterations, and an average, for both the binding protocol and the setup protocol. The average time in milliseconds of the setup protocol is 0.87ms and the average time in milliseconds of the binding protocol is 0.72ms The difference in time between these two protocols is unexpected because there are additional steps in the binding protocol to generate the One Time Password in the Token using the Linear Congruential Pseudo Random Number Generator. Whereas, in the setup protocol, the seed is just checked to be valid and then embedded so the binding protocol should take longer. After inspecting the time values returned by the protocols the minimum time returned by the setup is 0.2ms and the maximum is 3.6ms. The minimum of the setup is less than the the minimum for binding which is 0.5ms, which is expected, but the maximum time for the setup protocol (3.6ms) is greater than the maximum time for the binding protocol (1.6ms). Additional tests and measurements of the completion time for these protocols would be beneficial.

Future Work and Reflection

Due to resource constraints, exploration in some areas for this project was limited. Future work would include a more in depth implementation by modifying the FIDO Alliance's implementations of the CTAP2 protocol. This would involve first combining three github repositories (authenticator/token, client, additional framework) together, to create a working CTAP2 simulation. This would allow testing to determine baseline measurements for speed, since the modified protocol has additional steps to derive the One Time Password, and therefore should take longer. This cannot be determined without a baseline measurement of the implementation of the real CTAP2 protocol. An additional area for future work would be to define formally the security goals and prove the security formally using a computationally secure model or using the SUF, UF-T model proved by Barbosa, Boldyreva, Chen, *et al.* [1] for the proposed design. Integration of a Time-Based One Time Password, would be another interesting area of exploration because it could possibly solve the synchronization issue between the Token and the One Time Password Application. However, it requires the token to have some sort of persisting computing power. More research on a better random-number-generation algorithm that would meet the needs of the token and the One-Time-Password Application would be another topic that needs further exploration, as there are a lot of assumptions in the current simulation to allow for the design to function properly.

References

[1] M. Barbosa, A. Boldyreva, S. Chen, and B. Warinschi, "Provable security analysis of fido2," in *Annual International Cryptology Conference*, Springer, 2021, pp. 125–156.

[2] "Web authentication: An api for accessing public key credentials - level 2." (2021), [Online]. Available: `https://www.w3.org/TR/webauthn-2/`.

[3] "Client to authenticator protocol (ctap) proposed standard." (2021), [Online]. Available: `https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html`.

[4] M. Bellare and P. Rogaway, "Entity authentication and key distribution," in *Annual international cryptology conference*, Springer, 1993, pp. 232–249.

[5] "Fido security reference." (2021), [Online]. Available: `https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-rd-20210525.html`.

[6] T. E. Hull and A. R. Dobell, "Random number generators," *SIAM review*, vol. 4, no. 3, pp. 230–254, 1962.

1 Appendices

GitHub Code

CTAP2 Simulation: https://github.com/crawford134/CTAP2Project

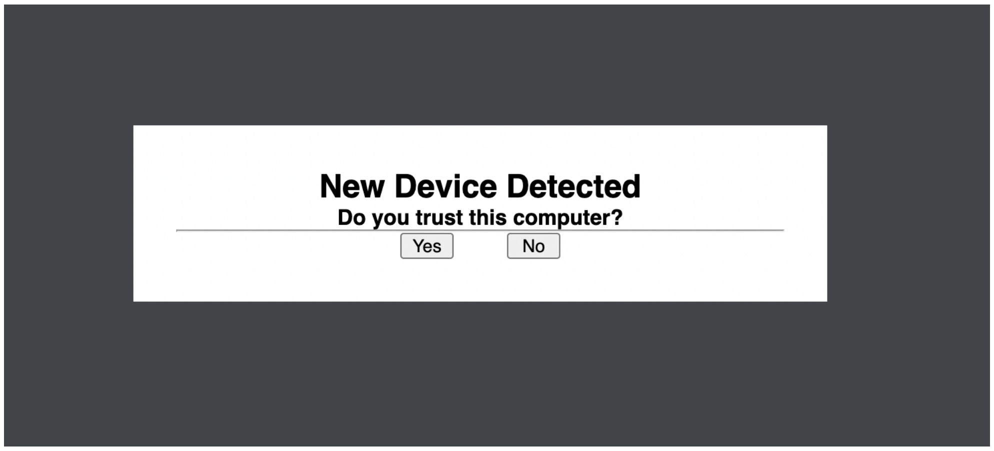


Figure 5: The CTAP2 Protocol cannot proceed unless the User is using a Trusted Device

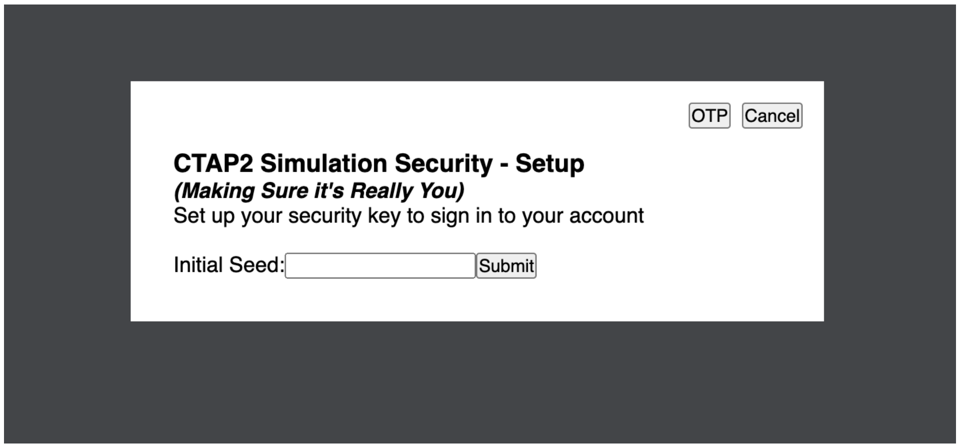

OTP
Cancel
CTAP2 Simulation Security - Setup
(Making Sure it's Really You)
Set up your security key to sign in to your account
Initial Seed:
Submit

Figure 6: The User is prompted to setup their Token through the Client by providing a Seed

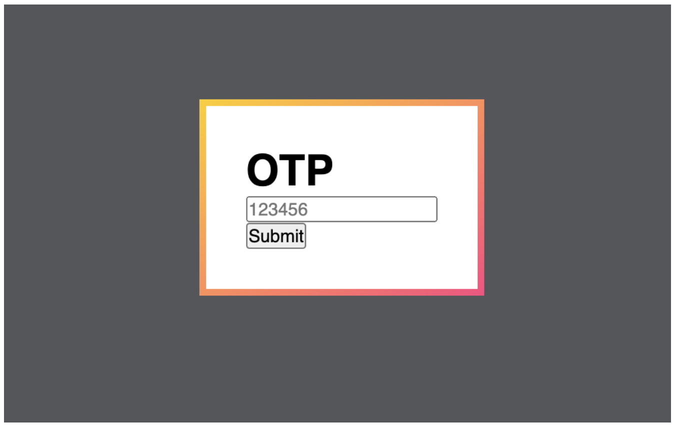
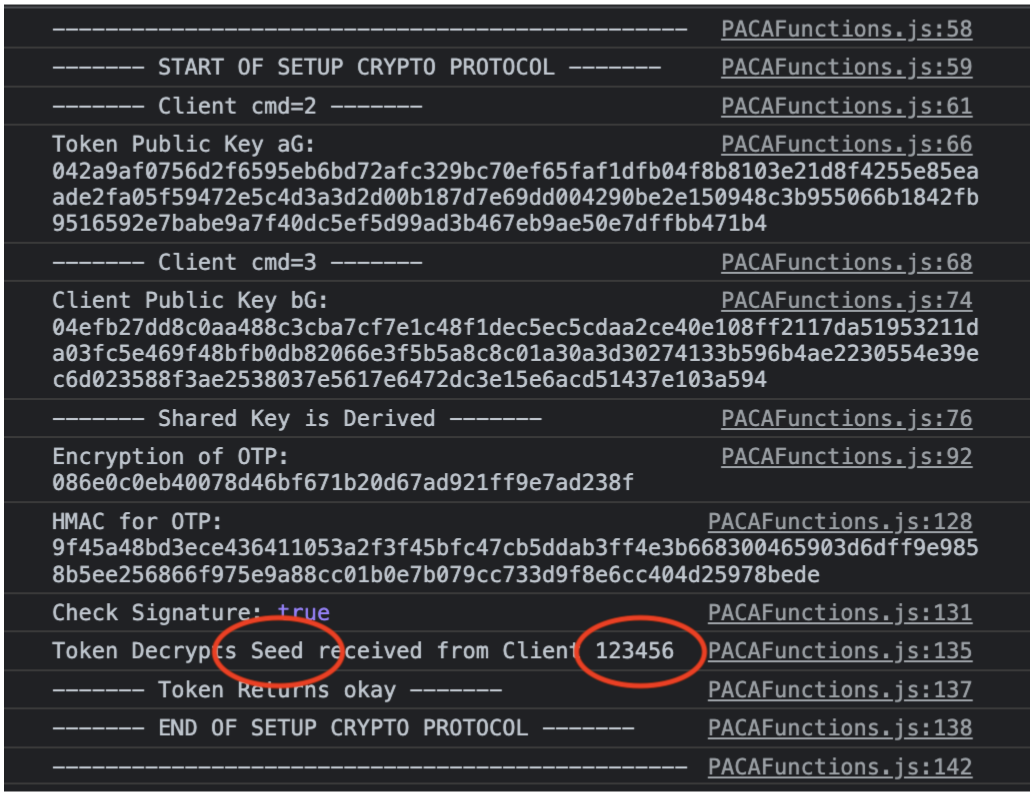


Figure 7: The User provides the same Seed to the OTP Application as they did to the Token



```
------------------------------------------------   PACAFunctions.js:58
------- START OF SETUP CRYPTO PROTOCOL -------      PACAFunctions.js:59
------- Client cmd=2 -------                        PACAFunctions.js:61
Token Public Key aG:                                PACAFunctions.js:66
042a9af0756d2f6595eb6bd72afc329bc70ef65faf1dfb04f8b8103e21d8f4255e85ea
ade2fa05f59472e5c4d3a3d2d00b187d7e69dd004290be2e150948c3b955066b1842fb
9516592e7babe9a7f40dc5ef5d99ad3b467eb9ae50e7dffbb471b4
------- Client cmd=3 -------                        PACAFunctions.js:68
Client Public Key bG:                               PACAFunctions.js:74
04efb27dd8c0aa488c3cba7cf7e1c48f1dec5ec5cdaa2ce40e108ff2117da51953211d
a03fc5e469f48bfb0db82066e3f5b5a8c8c01a30a3d30274133b596b4ae2230554e39e
c6d023588f3ae2538037e5617e6472dc3e15e6acd51437e103a594
------- Shared Key is Derived -------               PACAFunctions.js:76
Encryption of OTP:                                  PACAFunctions.js:92
086e0c0eb40078d46bf671b20d67ad921ff9e7ad238f
HMAC for OTP:                                      PACAFunctions.js:128
9f45a48bd3ece436411053a2f3f45bfc47cb5ddab3ff4e3b668300465903d6dff9e985
8b5ee256866f975e9a88cc01b0e7b079cc733d9f8e6cc404d25978bede
Check Signature: true                              PACAFunctions.js:131
Token Decrypts Seed received from Client 123456    PACAFunctions.js:135
------- Token Returns okay -------                 PACAFunctions.js:137
------- END OF SETUP CRYPTO PROTOCOL -------       PACAFunctions.js:138
------------------------------------------------   PACAFunctions.js:142
```

Figure 8: After receiving the Seed from the User, the Client and the Token engage in the Cryptographic Protocols to embed the Seed in the Token

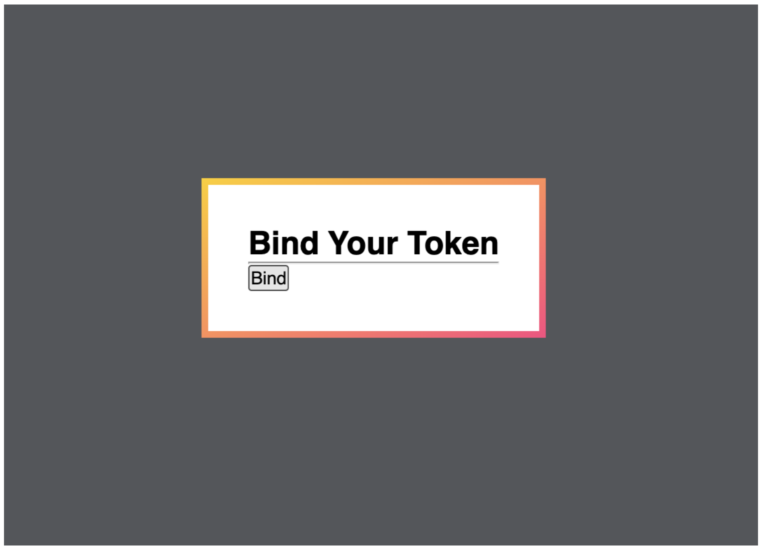
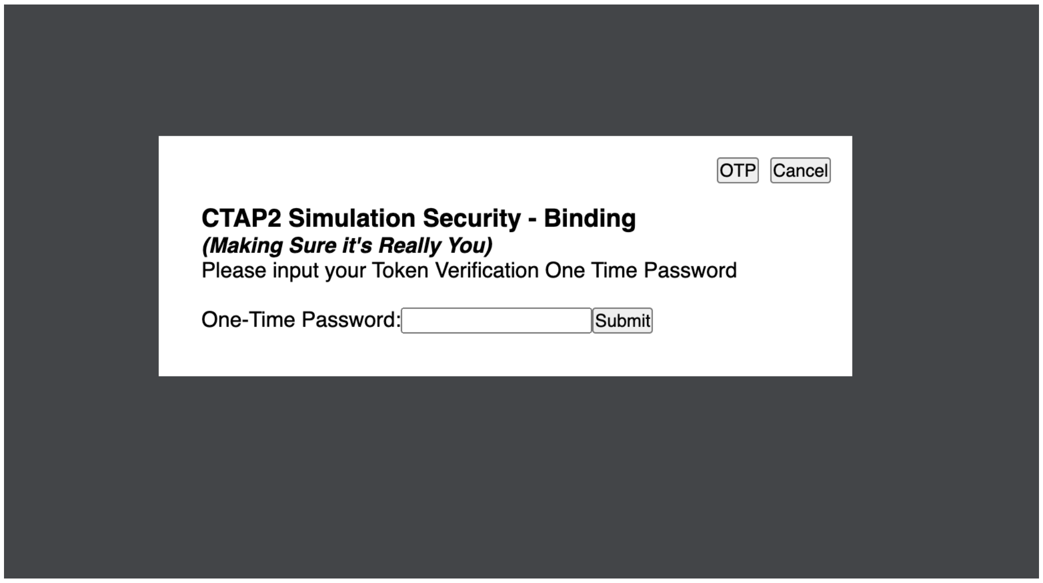


Figure 9: After successfully setting up their Token, the User is able to Bind their Token to the Client


OTP Cancel

CTAP2 Simulation Security - Binding
(Making Sure it's Really You)
Please input your Token Verification One Time Password

One-Time Password: Submit

Figure 10: During the Bind process, the Client prompts the User to input a One Time Password

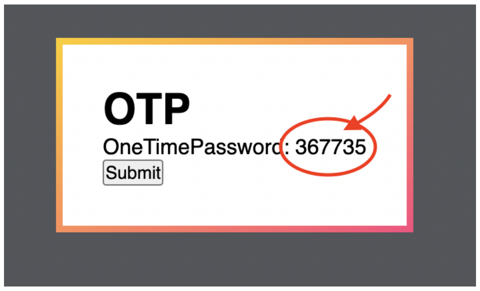
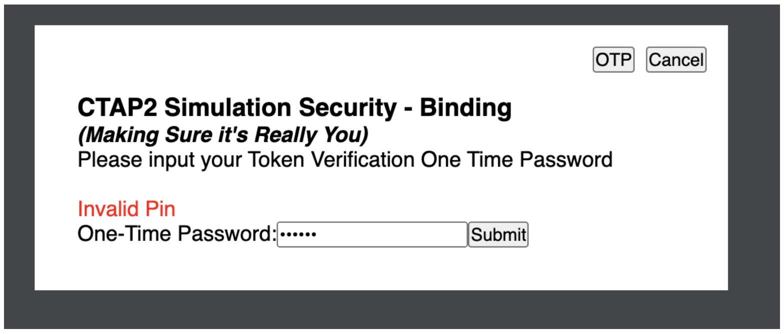

OTP
OneTimePassword: 367735
Submit

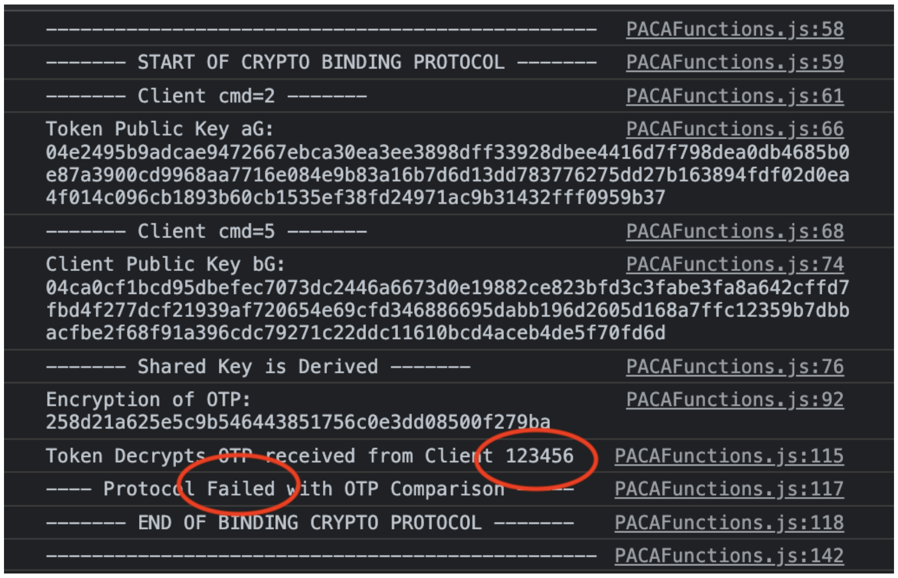
Figure 11: The User receives the One Time Password that is used in Binding from the One Time Password Application


OTP Cancel

CTAP2 Simulation Security - Binding
(Making Sure it's Really You)
Please input your Token Verification One Time Password

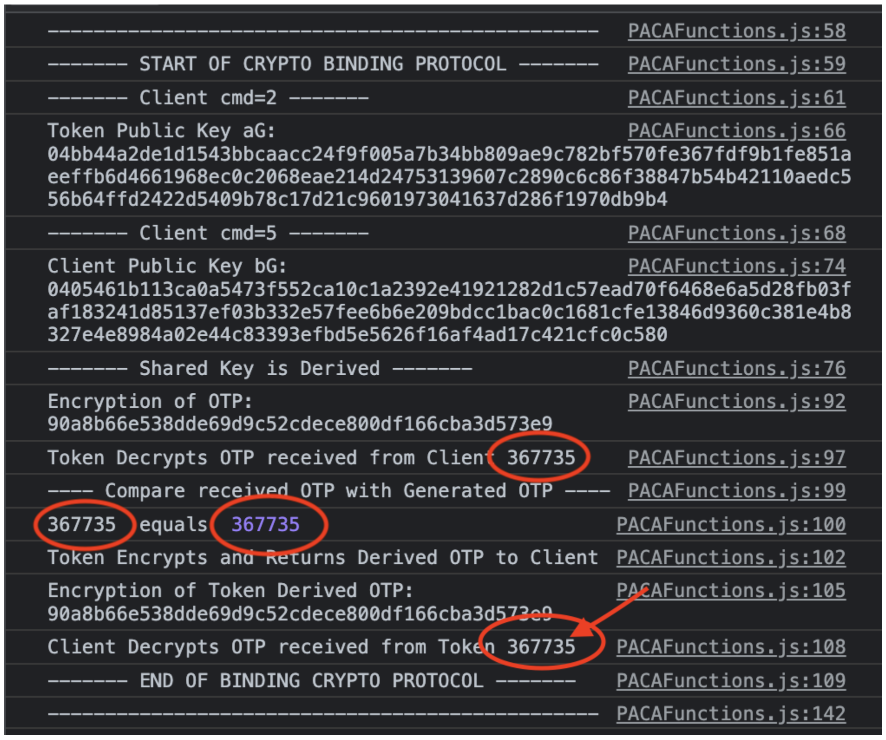
Invalid Pin
One-Time Password:•••••• Submit

Figure 12: The Binding One Time Password Prompt Display after a User enters an Invalid/Incorrect Pin



```
-------------------------------------------------   PACAFunctions.js:58
------- START OF CRYPTO BINDING PROTOCOL -------   PACAFunctions.js:59
------- Client cmd=2 -------                       PACAFunctions.js:61
Token Public Key aG:                               PACAFunctions.js:66
04e2495b9adcae9472667ebca30ea3ee3898dff33928dbee4416d7f798dea0db4685b0
e87a3900cd9968aa7716e084e9b83a16b7d6d13dd783776275dd27b163894fdf02d0ea
4f014c096cb1893b60cb1535ef38fd24971ac9b31432fff0959b37
------- Client cmd=5 -------                       PACAFunctions.js:68
Client Public Key bG:                              PACAFunctions.js:74
04ca0cf1bcd95dbefec7073dc2446a6673d0e19882ce823bfd3c3fabe3fa8a642cffd7
fbd4f277dcf21939af720654e69cfd346886695dabb196d2605d168a7ffc12359b7dbb
acfbe2f68f91a396cdc79271c22ddc11610bcd4aceb4de5f70fd6d
------- Shared Key is Derived -------              PACAFunctions.js:76
Encryption of OTP:                                 PACAFunctions.js:92
258d21a625e5c9b546443851756c0e3dd08500f279ba
Token Decrypts OTP received from Client 123456    PACAFunctions.js:115
---- Protocol Failed with OTP Comparison ----     PACAFunctions.js:117
------- END OF BINDING CRYPTO PROTOCOL -------    PACAFunctions.js:118
-------------------------------------------------  PACAFunctions.js:142
```

Figure 13: A failed Binding Crypto-Protocol is displayed here due to an Invalid One Time Password being inputted by User



```
-------------------------------------------------   PACAFunctions.js:58
------- START OF CRYPTO BINDING PROTOCOL -------   PACAFunctions.js:59
------- Client cmd=2 -------                       PACAFunctions.js:61
Token Public Key aG:                               PACAFunctions.js:66
04bb44a2de1d1543bbcaacc24f9f005a7b34bb809ae9c782bf570fe367fdf9b1fe851a
eeffb6d4661968ec0c2068eae214d24753139607c2890c6c86f38847b54b42110aedc5
56b64ffd2422d5409b78c17d21c9601973041637d286f1970db9b4
------- Client cmd=5 -------                       PACAFunctions.js:68
Client Public Key bG:                              PACAFunctions.js:74
0405461b113ca0a5473f552ca10c1a2392e41921282d1c57ead70f6468e6a5d28fb03f
af183241d85137ef03b332e57fee6b6e209bdcc1bac0c1681cfe13846d9360c381e4b8
327e4e8984a02e44c83393efbd5e5626f16af4ad17c421cfc0c580
------- Shared Key is Derived -------              PACAFunctions.js:76
Encryption of OTP:                                 PACAFunctions.js:92
90a8b66e538dde69d9c52cdece800df166cba3d573e9
Token Decrypts OTP received from Client 367735    PACAFunctions.js:97
---- Compare received OTP with Generated OTP ---- PACAFunctions.js:99
367735 equals 367735                              PACAFunctions.js:100
Token Encrypts and Returns Derived OTP to Client PACAFunctions.js:102
Encryption of Token Derived OTP:                  PACAFunctions.js:105
90a8b66e538dde69d9c52cdece800df166cba3d573e9
Client Decrypts OTP received from Token 367735    PACAFunctions.js:108
------- END OF BINDING CRYPTO PROTOCOL -------    PACAFunctions.js:109
-------------------------------------------------  PACAFunctions.js:142
```

Figure 14: After the User Inputs the One Time Password, the Client and Token engage in Cryptographic Protocols to confirm that the expected One Time Password was entered

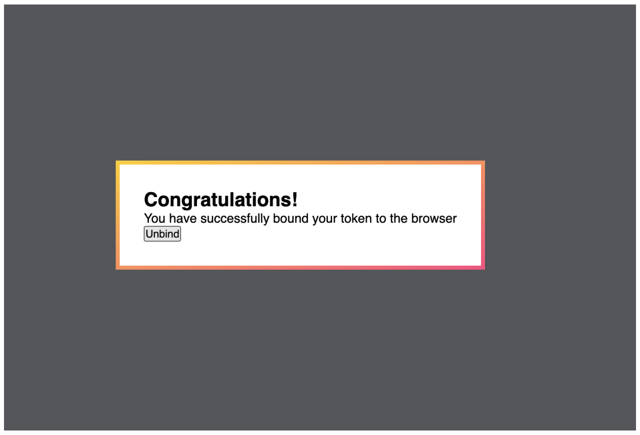

Congratulations!
You have successfully bound your token to the browser

Figure 15: A message after the protocol finishes indicating that Token has successfully bound itself to the Client

Iteration	Protocol	Time (ms)
1	Setup	1.6
2	Setup	0.2
3	Setup	0.3
4	Setup	0.2
5	Setup	0.9
6	Setup	0.2
7	Setup	3.6
8	Setup	0.9
9	Setup	0.9
10	Setup	0.8
	Average	0.87
1	Bind	1.1
2	Bind	0.6
3	Bind	0.5
4	Bind	0.6
5	Bind	0.8
6	Bind	0.8
7	Bind	0.6
8	Bind	0.8
9	Bind	0.6
10	Bind	0.8
	Average	0.72

Figure 16: Number of Milliseconds/Modified CTAP2 Protocol, and their respective averages