A first glance at the U2F protocol

Mickaël Bergem et Florian Maury
mickael.bergem@ponts.org
florian.maury@ssi.gouv.fr

[1] ParisTech
[2] ANSSI

Abstract Usage of Second-Factor Authentication (2FA) solutions constitutes a valid answer to the threat against weak credentials, such as passwords. Yet many 2FA schemes are vulnerable to prominent web threats such as phishing attacks.

The Universal Second Factor (U2F) protocol, specified by the FIDO Alliance, offers phishing-resistant 2FA solution, optionally based on hardware secure elements. Some well-known websites already use this authentication scheme, including Google, Dropbox and Github. Unfortunately, the U2F protocol still lacks independent reviews.

This study is a first attempt at assessing the actual security brought by the U2F authentication scheme. It confirms that, protocol-wise, most of U2F security goals are achieved. Yet, we found that some of the specified security measures benefit only from an experimental implementation and that some recommendations from the U2F specifications are not yet followed.

This paper draws a picture of the security brought by U2F and ultimately compares the protocol with some 2FA schemes: Short Message System (SMS) codes, Time-based One-Time Password (TOTP) and TLS client authentication using certificates.

1 Introduction

Using strong passwords is often difficult for end-users. Even then they can still be stolen, resulting in possibly long-term compromise of the user accounts. A solution to this issue is multi-factor authentication schemes, which combine several security checks to verify the identity of a user. Strong checks are based simultaneously on something the user knows (such as a password) and something the user possesses (such as a hardware token).

Multi-factor authentication schemes using mobile phones as a second-factor are now frequently used. Online banking is a typical use-case. Most are based on SMS codes or TOTP generation algorithms [11]. Other popular non-mobile based solutions include usage of dedicated tokens shipping secure hardware elements (e.g. smartcards).

While some of these solutions considerably improve the security of the authentication process, users may still fall for *phishing attacks*. Indeed, users of such solutions may unwittingly enter their credentials and their second-factor code on an attacker-controlled phishing website. Doing so opens way for various attacks described, for reference, in section 2 of this document. Then, section 3 sheds light on how these attacks apply to SMS and TOTP-based authentication schemes.

The FIDO Alliance [1] specified the U2F protocol to mitigate such threats. It empowers websites and applications with a phishing-resistant second-factor solution for end-user security. A presentation of the U2F protocol is provided in section 4.

At the time of writing, only Google, Dropbox and Github offer a U2F authentication support. Only few public presentations explained the protocol [12]. Also, to the best of our knowledge, no independent security analyses of the U2F protocol were published.

This paper presents the results of our first crack at assessing the actual security of the U2F protocol. During our study, we noted that most U2F security goals are fulfilled. Even so, our laboratory experiments also showed that some security measures were not implemented as specified or advised by the protocol specifications. As such, section 5 illustrates the consequences of the lack of a notification system upon completion of a cryptographic operation. We also uncovered during these experiments a possible issue with some token at the USB level. Then, section 6 discusses the efficiency and state of deployment of U2F built-in TLS Man-in-the-Middle (MITM) mitigation mechanism. In the light of the previous sections, section 7 finally compares U2F security to the one brought by the TLS client authentication scheme using certificates.

Our findings and conclusions were shared with the FIDO Alliance. When available, their feedback is included in the relevant section.

2 Threat model and web attacks

In this section, a brief overview of the web attacks that are relevant to this paper is provided for internal reference.

Cross-Site Scripting (XSS)

XSS attacks are performed by off-path attackers that may remain passive once the attack payload is planted. XSS attacks originate from the injection by a malicious third party of foreign code that is executed, client-side, within the context of a legitimate web application. Consequences are the loss of user experience control. The injected code may emulate or actually perform virtually any user action. XSS exploits generally allow the attacker to steal sensitive user data. Among these data can be listed password-based credentials, some second-factor authentication scheme credentials or some HTTP session tokens.

U2F specifications exclude this attack by assuming that the web browser acts as a trustworthy agent of the user. This hypothesis is discussed in section 7 where U2F is compared with TLS client authentication using certificates.

Cross-Site Request Forgery (CSRF)

CSRF attacks are performed by off-path attackers interacting with their victims. The principle is for attackers to automate user actions within the context of a legitimate web application. Victims might not even be aware that actions are performed on their behalf. To do so, attackers abuse the victims' user agents (e.g. the web browsers) so that authenticating information (e.g. the credentials or the session tokens) are automatically appended to arbitrary actions on a given web application. Attackers may also abuse the user agents so that they use an authenticated TLS channels to carry out the actions.

Phishing attacks

Simple phishing attacks Simple phishing attacks are performed by off-path attackers interacting with their victims. They rely on social engineering to capture credentials. For this, the attackers persuade the victims that a displayed website is genuine. Then, they entice the victims into entering their credentials on this website, hosted at a different Uniform Resource Locator (URL) than the legitimate website. The credentials can be stored by the attacker and then be replayed afterwards. For this attack to work, the authentication scheme must not involve any element ensuring the freshness of the stored credentials, such as challenges, time-constrained credentials or a ledger of already used one-time credentials.

This attack is mostly relevant to simple password-based authentication schemes. None of the 2FA schemes detailed in this paper are vulnerable to simple phishing attacks. This description purpose is to emphasize the notable differences between these simple attacks and the phishing attacks with real-time forwarding.

Phishing attack with real-time forwarding Phishing attacks with real-time forwarding are performed by off-path attackers interacting with their victims and the legitimate website. This variation of the phishing attacks is used when the authentication scheme involves a element providing freshness insurance to the authenticating party.

To perform this attack, the phishing website acts simultaneously as a server and as a client. From the legitimate website perspective, the phishing website is the workstation of the victim. From the victim perspective, the phishing website is the legitimate website. When an authentication procedure is initiated by the victim, the phishing website forwards all the messages and credentials it receives to the legitimate website. If the challenge is sent in-band (e.g. the legitimate website sends back to the phishing website a challenge), the phishing website returns it as is to the victim. If the challenge is sent out-of-band to the victim and sent back to the website by the victim, the phishing website forwards it to the legitimate website. At the end of this procedure, the victim provided to the phishing website all the credentials required to authenticate to the legitimate website.

TLS MITM

TLS MITM operations are performed by online entities interacting with both ends of the intercepted connections: the client, which is the user agent, and the server, which is the web application requested by the client. The MITM may impersonate both ends of the secure channel. To impersonate the TLS server, the MITM needs to present a TLS certificate accepted by the user agent. The user may also force acceptance of the certificate by the user agent. The interception can be performed at the network level or thanks to a specific TLS client configuration, such as the deliberate use of a corporate TLS proxy server.

Forgery attacks

Forgery attacks are a particular form of MITM attack, over secure or insecure channel. They pertain to the ability for an attacker to impersonate a user, by altering observed communications. This definition notably includes the replacement of tokens in a challenge/response authentication scheme.

Forwarding attacks

Forwarding attacks are a particular form of MITM attack, over secure or insecure channel. They pertain to the ability for an attacker to impersonate a user by replaying previously intercepted messages.

Parallel session attacks

Parallel sessions are initiated by off-path attackers without interaction with their victims, after an initial on-path traffic interception. Parallel sessions attacks are performed by attackers capable of inferring the user credentials for a new and possibly unrelated session with a specific website, thanks to the captured data.

3 SMS and TOTP authentication schemes security assessment

Many websites using 2FA rely either on SMS codes or on TOTP. In this section, the two schemes are presented, along with a security assessment of their usual implementations.

3.1 2FA based on TOTP

In this authentication scheme, the second-factor is either a hardware or a software module, called a token. The module contains a secret shared with the website. In typical deployments, the secret is generated by the website during the token registration procedure and the user types the secret in the token or takes a picture of a barcode representation of the secret. It is worth noting that in these deployments, the user takes no part in the secret generation procedure. As such, the randomness of the secret cannot be augmented by the user.

During the authentication procedure, the user is expected to type a TOTP read on the token in a web form. The TOTP is generated by the token, using the shared secret and a value derived from the current time. During the credentials verification procedure, the website computes its own version of the TOTP using the same input sources and compares the results with the user-submitted TOTP. Both values can only match during a limited time frame. Matching values would therefore seem to indicate that the user holds a registered token.

It is worth noting, though, that the TOTP is not bound to any technical information specific to the (user, website displaying the authentication form) tuple. As illustrated by figure 1, the user might type the TOTP on a phishing website (step 3), thus providing the attacker with a valid TOTP for the legitimate website. The attacker would then send the user credentials and the captured TOTP to the legitimate website, thus impersonating the user (step 4).

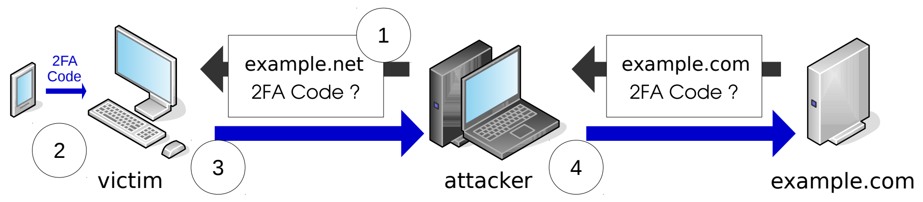

Figure 1. Phishing attack on example.com, allowing replay of the TOTP

A countermeasure could be to bind the TOTP to the website URL as perceived by the user agent. This solution is however impractical as TOTP tokens are generally devices with no access to any interface that would allow them to know the URL of the website currently displayed to the user.

3.2 SMS-based 2FA

In this authentication scheme, the second-factor is the user mobile phone.

During the registration procedure, the phone number is associated to the user account. During the authentication procedure, the phone number is retrieved from the account meta-data. Both procedures ensure the user holds the mobile phone by sending a SMS to the phone number. The SMS contains a value indistinguishable from random that the user is expected to read and type in a web form. The verification procedure is completed by comparing the user-submitted value with the value sent to the phone number.

It is worth noting that this authentication scheme is vulnerable to mobile network eavesdropping [10]. It also assumes that the mobile phone is sane and that no fraudulent applications are snooping on the received SMS.

The SMS-based 2FA scheme is also susceptible to *phishing attacks*, as illustrated by the figure 2. When the user receives the random value (step 3) and types it in on the phishing website (step 4), the attacker learns it, replays it to the legitimate website (step 5), and ultimately impersonates the user.

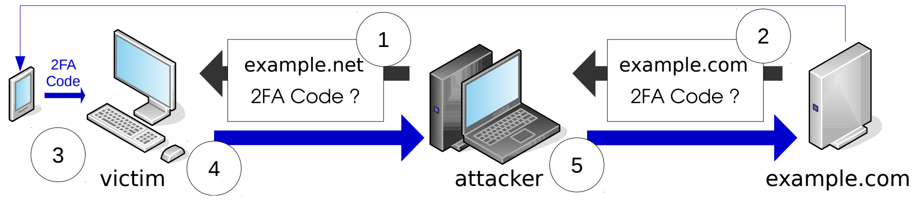

Figure 2. Phishing attack on example.com, against a SMS-based 2FA scheme

3.3 Considerations on replay attacks and parallel sessions attacks

Some implementations of the TOTP-based and SMS-based 2FA schemes allow the authentication code to be submitted multiple times to the legitimate website during a short time frame. Incentive to do so might be to limit authentication procedure failures for users connected through unreliable networks. Doing so open ways to parallel session attacks. Indeed, an attacker who captured a valid code might be replaying it in an unrelated session initiated during the short time frame.

The point for the attacker to mount a parallel session is that this session lifetime is not constrained by any user actions. As such, even if the user logs out, the parallel session remains valid.

4 A primer on the U2F protocol

4.1 The U2F registration and authentication procedures

The U2F authentication scheme relies on the usage of strong public key cryptography signatures. According to the protocol specifications, the private key may be hold within a software token or a hardware token. When using an external device, the cryptographic operations can be requested over USB, NFC or Bluetooth.

Usage of a token requires first the device to be registered to an existing account at each Relying Party (RP). A RP is basically a web application defined by its origin[3]. The definition of a RP is actually broader in

3. A web origin is composed of the protocol scheme (e.g. http or https), the domain name of the web application and the web server TCP port.

the U2F specifications. It encompasses for instance some related mobile applications. This broad definition, which relies on the U2F *AppID and Facets* features, is left out of the scope of the current paper.

During the registration procedure of a U2F token, illustrated by figure 3, the RP sends a challenge[4] (step 1) to the FIDO client (i.e. the web browser). The FIDO client hashes the challenge with various data, including the origin of the authentication web page. The origin is also hashed as a separate value. Both hashes are then sent to the token (step 2). Using these information, the token generates a RP-specific keypair[5] (step 3). It then sends back three information (step 4): the generated public key, a private key identifier, called the key handle, and a signed proof of ownership of the associated private key. The FIDO client forwards them and some metadata to the server (step 5). The registration is successful if the proof of ownership is verified by the server (step 6). The verification steps involve a consistency check of the metadata sent by the FIDO client and the use of the registered public key to verify the signature of the proof of ownership.

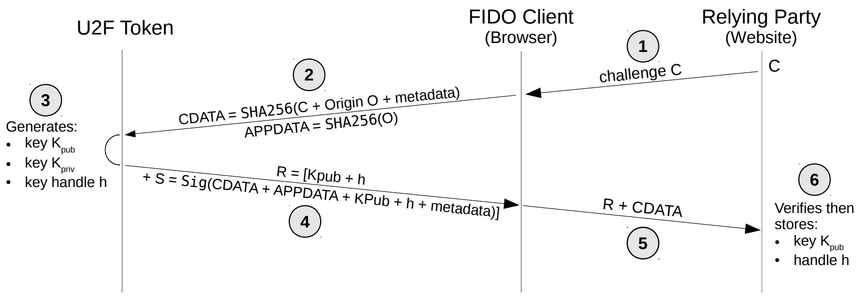

Figure 3. U2F registration process

The registration process may occur multiple times, allowing users to associate multiple tokens to their accounts. This may be of use, for instance, to avoid users from locking themselves out of their accounts if they lose their only token.

4. A challenge can theoretically either be a nonce or a random value. U2F specifications recommend the use of a random value with at least 64 bits of entropy.

5. The keypair is generated with the NIST P-256 curve. Signatures are performed using this keypair and ECDSA.

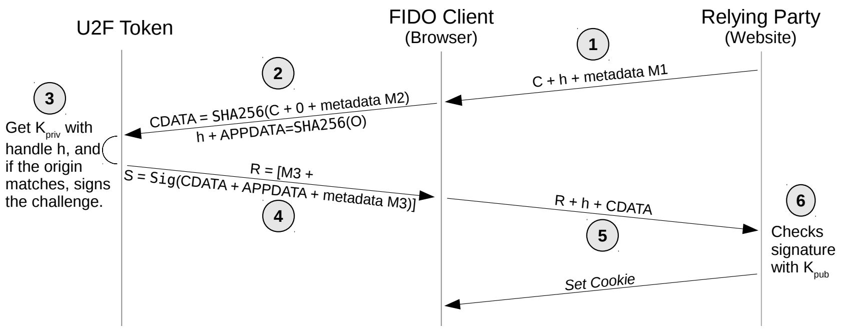
U2F Token
FIDO Client
(Browser)
Relying Party
(Website)
1
C + h + metadata M1
2
CDATA = SHA256(C + 0 + metadata M2)
h + APPDATA=SHA256(O)
3
Get K_priv with handle h, and if the origin matches, signs the challenge.
R = [M3 +
S = Sig(CDATA + APPDATA + metadata M3)]
4
R + h + CDATA
5
6
Checks signature with K_pub
Set Cookie

Figure 4. U2F authentication process

Once a token is registered to an account, this token can be used for authentication purposes, as illustrated in the figure 4. The authentication procedure consists of the following steps:

- The user is authenticated by the RP first, using another authentication scheme. The typical scenario involves the user entering a login and password in a web form;
- Once these credentials are verified at the server-side, the RP sends to the FIDO client a challenge and the key handles registered for the user account (step 1);
- The FIDO client hashes the challenge with some additional information and forwards the resulting digest to the U2F token along with the key handles[6] and the hash of the RP origin (step 2) ;
- The token recovers the private key thanks to the key handle and uses it to sign the digest and some additional data, such as a counter used for anti-replay purposes (step 3);
- The signed blob and the metadata are then sent back to the FIDO client (step 4);
- The FIDO client forwards them back to the RP (step 5) for verification of the signature using the previously registered public key (step 6).

A U2F keypair is bound to a specific RP, during the registration procedure. As such, a token must refuse to sign with a keypair issued for a different origin. This refusal is illustrated by the figure 5. The origin

6. More precisely, the FIDO client first sends one probe to the token for each key handle. This way, the token can tell the FIDO client which key handle was emitted by the probed token and which key handles were generated by some other tokens. Once the correct key handle is identified, the actual signing request is sent.

check ensures that the public keys and key handles issued for a given RP cannot be exercised by a different RP, such as a phishing website.

Another attack scenario that U2F mitigates is illustrated by the figure 6. It involves attackers trying to forward a U2F-signed authentication response sent to their phishing website. Indeed, the received signature is generated using a keypair specific to the phishing website. The genuine website would therefore be unable to verify the signature sent to the phishing website with the public key they know for the victim's account.

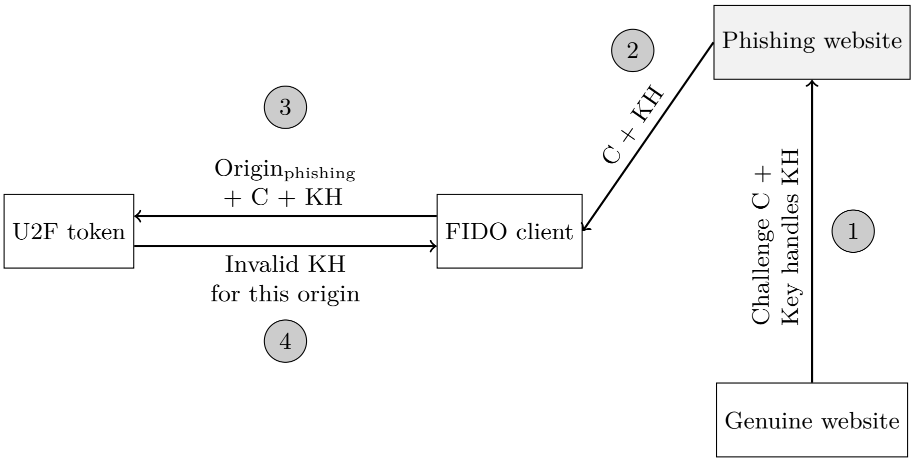
Phishing website
2
C + KH
3
Origin$_{\text{phishing}}$ + C + KH
U2F token
FIDO client
Challenge C + Key handles KH
1
Invalid KH for this origin
4
Genuine website

Figure 5. The attacker forwards the key handles and challenge from the genuine website. The token refuses to use these key handles when they come from the phishing website.

5 Parallel signing

The U2F security reference document [2] specifies that transaction non-repudiation is not a Security Goal (SG) of U2F. Another protocol, called Universal Authentication Framework (UAF), is specified by the FIDO Alliance to do so. In particular, the UAF protocol requires the use of a *secure display* and *transaction confirmation* while the U2F protocol does not. The UAF protocol is left out of the scope of this study.

This section does not present an attack against the U2F protocol. Instead, it merely illustrates an attack scenario that would affect a RP implementing, against the specification recommendations, transaction non-repudiation with U2F tokens.

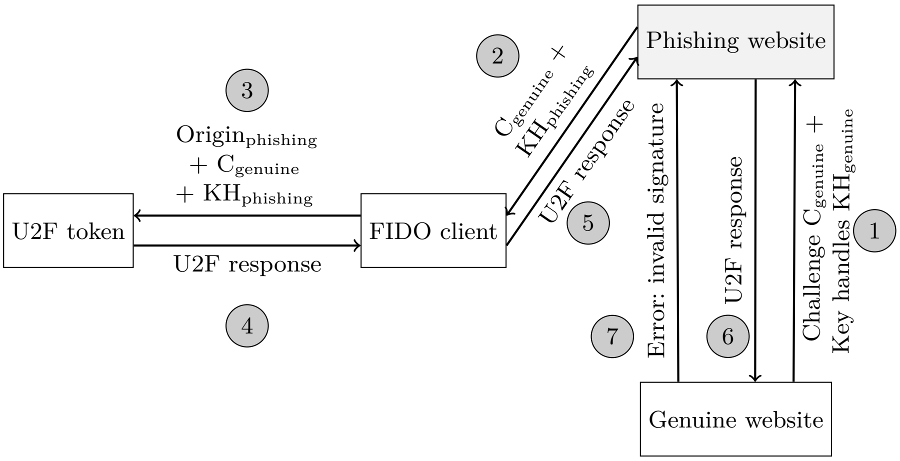
Phishing website
Genuine website
U2F token
FIDO client
1
Challenge $C_{genuine}$ + Key handles $KH_{genuine}$
2
$C_{genuine}$ × $KH_{phishing}$
3
$Origin_{phishing}$ + $C_{genuine}$ + $KH_{phishing}$
4
U2F response
5
U2F response
6
U2F response
7
Error: invalid signature

Figure 6. The attacker forwards the challenge from the genuine website and provides the key handles for the phishing website. The token signs the challenge but the genuine website rejects the signature.

We demonstrated this attack scenario in a proof-of-concept laboratory experiment. During these tests, we were also able to identify some practical issues and some deviations from the recommendations provided by the U2F specifications.

5.1 The case of two pages asking for a U2F interaction

General principle of parallel signing requests The parallel signing attack scenario involves two U2F-enabled web pages loaded from different origins. For this scenario, it is worth noting that it is irrelevant whether the two web pages are displayed in different browser instances, in different tabs or even within a single tab with the help of HTML *iframes*.

One of the web origins involved in the attack needs to be under the control of the attackers. This can be a website of the attackers or a XSS-ed waterhole [7].

This attack relies on the instrumentation or at least the detection by the attackers of which page is loaded by the victim's browser from the origin that is not directly under the attackers' control. By doing so, the attackers are capable of synchronizing two U2F operations, each one taking place from their own websites. The nature of U2F operation is

7. A waterhole is a web location commonly visited by the victim. Attackers may trap such locations, waiting for the user to access it to bootstrap the next step of their attack.

irrelevant to this attack as there is currently no difference from the user experience perspective between registration operations and authentication requests. Actually, as illustrated by the figure 7, the token has no way of telling which application asked the kernel to send an Application Protocol Data Unit (APDU) nor if this application is a FIDO client or any other unrelated sender.

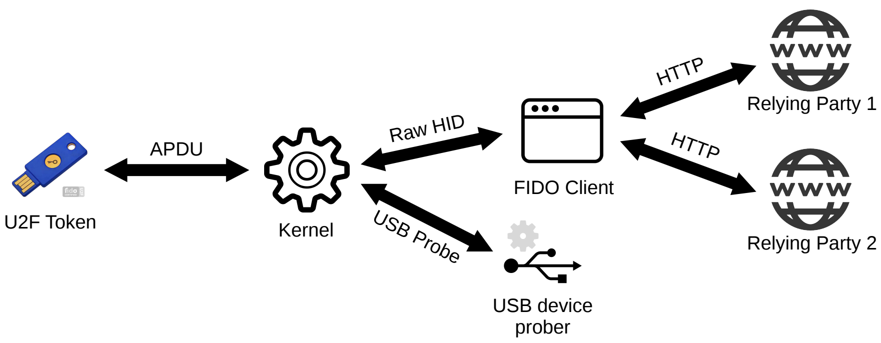

Figure 7. Message flow and APDU sender confusion

At some point in time, the two web pages will synchronously, thanks to the attackers' influence, ask for the U2F token to sign a challenge for their own websites. According to the specifications, user consent is mandatory for each cryptographic operation (SG-7). The problem is the user has no way of telling for which of the two simultaneous U2F solicitations his consent is requested first.

By manipulating the user into giving his consent to an arbitrary U2F operation of the attackers' choice, the attackers can achieve various results:

- The Denial of Service (DOS) of some U2F token implementations, which can only store a limited number of keypairs within the token. By repeatedly tricking the user, the attackers could fill these tokens with unique keypairs per origin under their control, until the token is full.
- A transaction non-repudiation violation, as illustrated by the case-study presented in section 5.2.

User awareness of parallel signing requests In the previously described scenario, the user consent is diverted to satisfy the U2F requests from the website that does not have focus or that is hidden from the user. One could think the user would be alerted by his user consent not triggering any response from the displayed website. Yet, several reasons might justify their lack of suspicion:

- The U2F token used in our experiment ended up presenting some USB connector defects causing it to be undetected by USB hubs upon insertion. This is a problem because the user consent, for our token, is given by inserting it in the USB port. The average user might not be able to tell whether a U2F operation was performed for the wrong website, or if the token simply malfunctioned.
- Some applications send probes to new USB devices [15]. In our case, `libmtp` [3], a library used, for instance, with digital cameras, probed our U2F token. This sometimes placed our token in a state where it was unable to answer signature requests. We did not investigate the situation but we guess that the probe consumed the user consent. Again, the user might not be able to tell whether the token was solicited by a random probing application, or by the browser for an unexpected website.
- U2F signing requests emitted by the browser have timeouts. The user might think that the delay was reached. They might then renew their consent for the currently displayed operation.

5.2 A case-study attack: transaction non-repudiation

Material used to implement the experiments The scenario described in this section was reproduced in laboratory. Our proof of concept used the following material:

- A USB U2F token;
- Two specially-built test websites using the Django U2F module with the underlying Yubico U2F Python library [4];
- Valid and legitimate certificates emitted by a public CA for test domains under our control;
- The Chrome browser, version 45;
- Ubuntu 15.10 as the operating system.

Case-study: CSRF protection and banking transaction non-repudiation Users might use U2F for 2FA to their bank website to improve the security of the login procedure.

Some 2FA schemes, such as SMS, are then commonly reused for transaction confirmations on these websites. For instance, it is a common practice to send a transaction confirmation code by SMS, and to have the user type it in a web form to validate a money transfer. According to the

U2F specifications, U2F should not be used to perform such transaction confirmations. This section illustrates the risk of doing it anyway.

Under the hypothesis that the user is already authenticated on the bank website, an attacker can force the user into confirming a transaction protected using U2F as the sole protection mechanism, by performing the following steps:

1. Present to the user, in a browser tab, a page allegedly asking for U2F authentication on an attacker-controlled website;
2. In the background, open an *iframe* with the order confirmation in it;
3. Entice the user into activating the U2F device, allegedly on the displayed website, while the first U2F request sent to the token is the one for the order confirmation.

Such a scenario might lead the user to confirm the transfer order without noticing.

This demonstration illustrates why the FIDO Alliance rightfully forbids the use of U2F for transaction non-repudiation. Among other security measures, which U2F deliberately lacks, a secure display is paramount to provide the *What You See Is What You Sign* (WYSIWYS) property, and thus a strong transaction non-repudiation property.

This kind of attack does not necessarily exist with SMS and TOTP transaction confirmation schemes. Indeed, in both of these cases, the user must interact with the browser tab that has focus (i.e. that is displayed), in order to type in the one-time code. As long as the displayed web page is genuine and is not trapped (that is that there are no hidden/transparent frames, like those used in click-jacking attacks) the user knows exactly in which website the one-time code is typed in.

5.3 Countermeasures to parallel signing requests

The U2F specifications forbid the use of the protocol for transaction confirmation. Even so, the impact of the above scenario would be more limited if some additional safe-guards were implemented. It is very important to note that implementing these additional safe-guards do not completely secure the usage of U2F for transaction non-repudiation. They merely improve the user experience and the user's ability to detect suspicious situations.

Regarding the threat generated by faulty tokens, we recommend that browsers implement the optional infobar that is already mentioned in the protocol specifications. This could be done with already available frameworks, such as the HTML5 Web Notification API. Doing so would not prevent the parallel signing attacks but would at least warn the user about unexpected signature requests.

Concerning probing applications that are unrelated to the U2F protocol, we recommend that the U2F HID protocol be updated to add some form of format checks. By doing so, undesirable probes could no longer be mistaken with U2F messages. This could be achieved by adding some magic numbers in the U2F frames sent to the tokens, by implementing a simple checksum or any other format check that would allow U2F token to sort out whether a message is a valid U2F message. The FIDO Alliance considered this protocol evolution recommendation. In the end, they rejected it on the basis that this is more of an implementation problem than a protocol one.

We agree that ultimately this is an implementation problem. Yet, we also think that the USB protocol itself is lacking some stepping stones allowing a USB device to unequivocally identify a frame sender and a frame payload purpose. As such, we think that a protocol taking advantage of the USB transport should assist the USB device into identifying if a frame is well-formed and meaningful. Should this information be lacking, the FIDO Alliance might need to drop support of tokens for which user consent is given by token insertion in a USB port. Indeed, in the long run, these tokens may contemplate only two options:

– Accept that they can be victim of accidental or deliberate DOS “attacks” from random software;
– Ignore “malformed requests” and violate the requirement to handle one single request per user consent.

The first choice bears no security risk but users might think that the product is broken. The second choice, however, would introduce a security risk (e.g. some forms of oracles) for the users of such tokens.

6 TLS man-in-the-middle resistance

The U2F specifications refer to a channel binding security measure to thwart TLS MITM attacks. For this purpose, specifications published on May 14, 2015 use an experimental alteration of the TLS protocol, known as TLS Channel ID. This variation was documented in an expired Internet-Draft [9] at the IETF. Newer versions of the U2F specifications will probably refer to its spiritual heir, the Token Binding mechanism, specified by the *tokbind* IETF working group [5].

The TLS Channel ID extension support is optional according to the U2F specifications. Lack of support of said extension does not impede most security goals of U2F. However, supporting it may grant additional protections against some attack scenarios involving a TLS MITM. It is worth mentioning that TLS MITM are very powerful attackers. In most cases, vulnerability to such attacks is an acceptable risk.

This section provides more details about TLS Channel ID. It then presents the results of our conformance testing of the experimental implementation of TLS Channel ID by the Chrome Web Browser, version 45, and the Google Accounts website.

6.1 TLS Channel ID inner working

TLS Channel ID alters the TLS handshake protocol by specifying a new TLS extension. As illustrated by the figure 8, this new extension is negotiated, as usual, in the `Client Hello` (1) and `Server Hello` (2) messages. This extension carries no data and is used for signaling purposes only. If both TLS end-points signal their support of TLS Channel ID, a new message, called `EncryptedExtensions` (7) is sent by the TLS client after the TLS `Change Cipher Spec` (6) message and before the `Finish` (8) message.

`EncryptedExtensions` create a new framework for sending extension data protected simultaneously by the negotiated cipher suite and by the TLS handshake integrity mechanism. For TLS Channel ID, the data sent inside the `EncryptedExtensions` message is the public key of a unique keypair[8] generated per (TLS client, TLS server) tuple. This keypair is generated upon first visit of an origin and is kept within the client configuration (e.g. the browser profile) for later use. The `EncryptedExtensions` message also contains a proof of ownership of the private key of this keypair. The proof covers all previously exchanged messages during the TLS handshake, similarly to the standard `CertificateVerify` message [17].

The same end-points-specific keypair is reused upon TLS session resumption or any new full TLS handshake. While the public key, and thus the TLS Channel ID, remains constant across several TLS sessions, the proof of ownership is different during each handshake. It thus provides a constant and reliable identifier of the TLS client: the public key. This identifier prevents *forwarding attacks* since the proof of ownership is unique to each TLS transaction, at least thanks to the server nonce, the server certificate and the optional `ServerKeyExchange` message. As such, a TLS MITM cannot impersonate the TLS client without knowledge of the private key associated to the TLS Channel ID.

8. The keypair is generated using NIST P-256 curve. It is exercised with ECDSA.

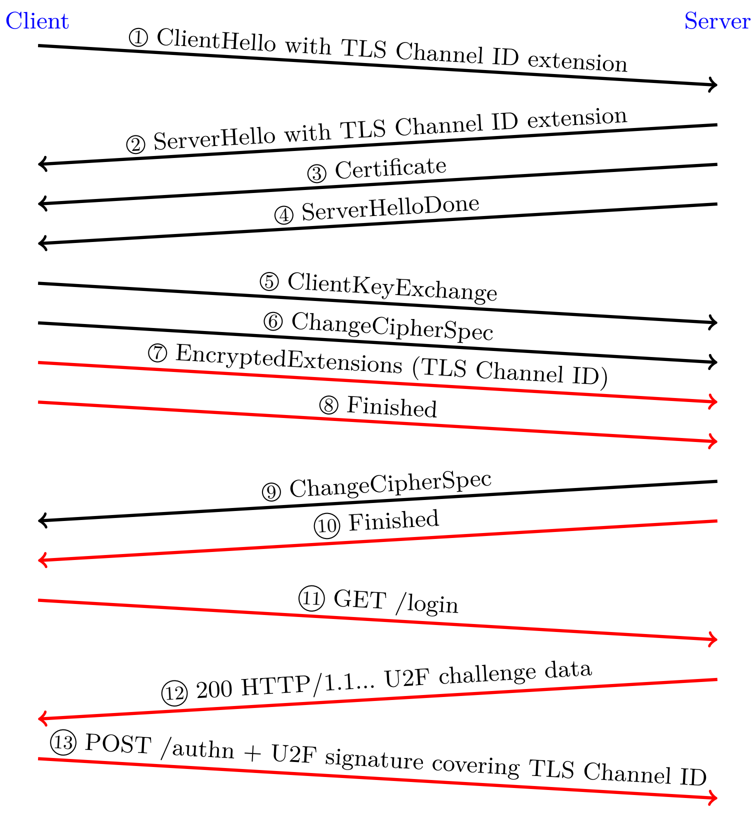
Client
Server
① ClientHello with TLS Channel ID extension
② ServerHello with TLS Channel ID extension
③ Certificate
④ ServerHelloDone
⑤ ClientKeyExchange
⑥ ChangeCipherSpec
⑦ EncryptedExtensions (TLS Channel ID)
⑧ Finished
⑨ ChangeCipherSpec
⑩ Finished
⑪ GET /login
⑫ 200 HTTP/1.1... U2F challenge data
⑬ POST /authn + U2F signature covering TLS Channel ID

Figure 8. The TLS Channel ID Messages and U2F Messages

Attackers could perform a TLS MITM, where they would impersonate the server identity, using a fake-yet-valid certificate. This would allow them to learn the TLS Channel ID public key, since they would know the cipher key to decrypt the `EncryptedExtensions` message. However, attackers would not be able to perform a *parallel session attack* using this information. Indeed, the parallel sessions TLS handshakes would have a different TLS transcript, thus preventing replay of the proof of ownership read from the decrypted `EncryptedExtensions` message. Attackers would not be able to forge such a proof either. Indeed, knowing the TLS Channel ID public key and potentially several proofs of ownership of the private key is of no use to infer the client TLS Channel ID private key when using correct ECDSA implementations and the NIST P-256 curve. Thus, attackers cannot impersonate the client by reusing the TLS Channel ID public key in a parallel session because they cannot prove ownership of the private key to the legitimate server.

TLS Channel ID used alone is vulnerable to a downgrade attack. As illustrated in figure 9, attackers posing as the TLS server can simply advertise to the legitimate client lack of support of TLS Channel ID from the server. Simultaneously, attackers posing as the client can advertise lack of support of TLS Channel ID to the legitimate server. Without this signaling, both legitimate end-points will think that TLS Channel ID is not supported by the other party. The legitimate client will not send the `EncryptedExtensions` message and the TLS server will not expect one from the client.

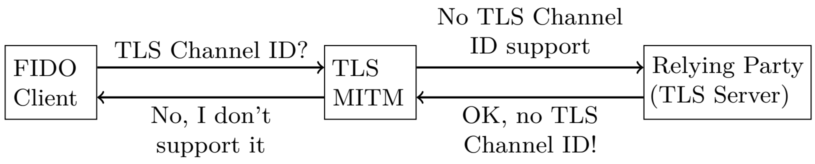

Figure 9. Illustration of trivial downgrade attack against TLS Channel ID

U2F can complement and leverage TLS Channel ID to prevent all forms of TLS MITM. According to the specifications, when a client supports TLS Channel ID and performs a handshake with a server that does not support TLS Channel ID, the U2F messages contain a U2F-signed specific entry: *cid_pubkey* with a string value *unused.* If TLS Channel ID is signaled by both parties, the string value contains instead the TLS Channel ID as seen by the TLS client. The legitimate server that supports TLS Channel

ID is expected to check the *cid_pubkey* value. If this value is incorrectly signed, if it is different from the TLS Channel ID that is expected for this client [9], or if the connection does not use TLS Channel ID and the *cid_pubkey* values *unused*, a TLS MITM is detected.

It should be noted at this point that the U2F specifications does not mention that the server should perform this verification. This requirement is somewhat implicit. We recommended to the FIDO Alliance and Google to add to future versions of the specifications a check-list of the verifications a U2F-enabled server should do. They submitted this recommendation to the other members of the consortium for revision.

6.2 TLS Channel ID support as implemented

TLS Channel ID is currently standard in the U2F protocol. All U2F messages carry U2F-signed information regarding support (or lack thereof) of TLS Channel ID by the client.

At the time of writing, in the web context, only recent Chromium and Chrome browsers support TLS Channel ID, and only Google Accounts website (and some Google test websites) advertise TLS Channel ID support during the TLS handshakes.

To test the effectiveness of TLS Channel ID as implemented by Google browsers and websites, we set up a TLS proxy performing a TLS MITM between those two components. For this, we used the OWASP `Zed Attack Proxy` [6]. This proxy lacks TLS Channel ID support, thus effectively performing a downgrade attack against TLS Channel ID. By reading and replaying all TLS messages with this proxy, we were able to read the HTTP stream, including the U2F messages sent by the browser.

This experiment resulted in two main observations. We first noted that Chrome and Chromium implementations do not implement TLS Channel ID support signaling within U2F messages as specified by the FIDO Alliance. Instead of returning a signed *cid_pubkey* string equal to *unused*, these implementations return a signed empty string. This discrepancy could lead to future interoperability problems. When we contacted Google, they told us they will fix this in the near future. It is worth noting, though, that since only Google browsers and Google websites support TLS Channel ID at the time of writing, the impact of this finding is very limited.

The second observation from our experiment was that we were able to successfully authenticate to our Google test account using U2F and

9. The TLS Channel ID must be sent to the web application running the website by the TLS server end-point at each request. This allows the website to compare it with the user-submitted U2F-signed TLS Channel ID.

Chrome or Chromium, while going through the TLS proxy. This happened in spite of the browser support of TLS Channel ID and in spite of the U2F-signed support signal. We contacted Google and the FIDO Alliance to warn them about the success of this downgrade attack. They answered that Chrome support of TLS Channel ID is currently experimental and still buggy at times. Thus, even if Google websites could detect such downgrade attacks, they decided neither to enforce the use of TLS Channel ID nor to use it to protect against such TLS MITM [10]. They also mentioned that TLS Channel ID would prevent legitimate use of corporate TLS proxies and they were not ready to keep U2F users from accessing Google services when such proxies were in use.

We think these arguments are perfectly understandable. Yet, the net result is that the only TLS Channel ID server-side implementation at the time of writing does not verify the value of *cid_pubkey* to prevent TLS MITM for most clients [11]. As such, security assessment of U2F deployment should consider that U2F does not prevent any form of TLS MITM at the time of writing. Following the acceptable level of security risks, this might or might not be a problem, since attackers capable of performing TLS MITM are considered quite powerful.

7 Comparison of U2F and TLS client certificate authentication schemes

7.1 Comparison criteria

To compare U2F with TLS authentication using client certificates, no standard framework was defined, to the best of our knowledge. Google already performed a comparison of several authentication schemes in prior art [12], without formally specifying their framework. Unfortunately, TLS authentication using client certificates was left out of Google comparative analysis.

For these reasons, we arbitrarily choose to compare these two authentication schemes with the following criteria list: phishing attack resistance, TLS MITM attacks resistance, web application protection before authentication, centralized revocation capability, ease of use, protocol maturity, implementation maturity, and unlinkability of accounts.

10. Google websites are protected against most TLS MITM attacks thanks to HTTP Strict Transport Security (HSTS) and public key pining. These protections are disabled when certificates from private CA are used. This is the case with corporate TLS proxies.

11. Google activated on our test account an experimental server-side check that effectively prevented the attack. This check is activated for Google employees' account.

Most criteria are self-explanatory. Unlinkability of accounts relates to a privacy threat for users owning several accounts identified by unrelated pseudonyms. When evaluating an authentication scheme, a privacy risk exists if an authenticating party or several colluding parties can identify or suspect that a list of accounts at that authenticating party are owned by a single user because of the authentication scheme.

7.2 Assumptions used for the comparison

We assume that Software Security Module (SSM) [12] are not used for either authentication scheme. As such, private keys for the TLS authentication scheme are assumed to be stored in a hardware secure element, such as a smartcard. Similarly, a U2F token with a hardware secure element is used.

7.3 Survey of the criteria and justifications

TLS MITM resistance The TLS `CertificateVerify` message is bound to one TLS handshake between two specific endpoints. The server nonce, the server certificate and the optional `ServerKeyExchange` message are supposed to prevent replay of captured messages by a TLS MITM. These elements are not always present in a TLS handshake, though. This is the case during a TLS session resumption, for instance. This led to the triple handshake attack [13], which demonstrated a *forgery attack* enabled by a mix of several TLS features. Several TLS vendors implemented a workaround to temporarily thwart this attack. Conscientious system administrators also disabled some of the rarely-used features that are required to perform the attack.

U2F effectively prevents *forgery attacks*, *forwarding attacks* and *parallel session attacks* using channel binding, as described in section 6. However, at the time of writing, support of TLS Channel ID, used in U2F for channel binding purposes, is still experimental. Only Google Accounts website and some Google test sites deploy this countermeasure and its effectiveness is limited to a constrained list of users. This leaves all other U2F users vulnerable to TLS MITM attacks.

It is worth mentioning that the SLOTH attack [14] affected TLS authentication using client certificates as much as TLS Channel ID and its spiritual heir, Token Binding. The SLOTH attack demonstrated that the

12. A SSM is functionally equivalent to a Hardware Security Module (HSM). However, SSM rely only on software to protect the cryptographic material, while HSM store it in a tamper-proof or at least tamper-evident secure element.

hash algorithms negotiated during early steps of the TLS handshake or required by the TLS RFCs could lead to the use of weak hash algorithms or shorten Message Authentication Code (MAC). In both cases, this may lead to collision attacks. Such hash collisions may allow attackers to perform a *forwarding attack* by replaying to the legitimate TLS server captured `CertificateVerify` or Token Binding messages. Several implementations fixed the issue by disabling MD5 for signature purposes. In response, Token Binding draft also changed the signed value from the `tls_unique` value from RFC5929 [7] to the `exported keying material` from RFC5705 [16].

The SLOTH attack is especially relevant to this authentication protocols comparison as it illustrates that some TLS attacks affecting the TLS client authentication scheme using certificates may also affect U2F because of its reliance on some TLS infrastructures.

Phishing attack resistance: By construction, a *phishing attack* uses a different origin than the legitimate website.

In the case of TLS, the `CertificateVerify` message is bound to one TLS handshake between two specific endpoints. The certificate subject, related to the origin, is distinct in the `Certificate` server message sent by the phishing website and in the one sent by the legitimate server. Thus, the handshake transcripts are different, and so are the `CertificateVerify` messages. As such, an attacker cannot, in theory, replay a `CertificateVerify` message to impersonate a user. In practice, the triple handshake and SLOTH attacks, already discussed above, showed that some combination of TLS features can be used to either avoid sending the TLS messages causing a transcript variation or to make the transcript digests collide.

In the case of U2F, keypairs are bound to specific origins. A phishing website would therefore only receive U2F-signed messages unrelated to those that the legitimate website would receive.

Web application protection before authentication: TLS client authentication using certificates happens before any application data can be exchanged. As such, unauthenticated users or attackers are unable to directly interact with the web application, whatever the number of vulnerabilities that it contains. On the other hand, lack of support of a logout feature in all major browsers prevents a TLS-authenticated user from avoiding a *CSRF attack*, even after the user stopped interacting with the web application.

U2F authentication scheme occurs at the web application level, using web forms and JavaScript. As such, an attacker could completely circumvent the authentication procedure in the case of a web application vulnerability exploitable by unauthenticated attacker. On the other hand, session logout can be implemented by websites taking advantage of U2F.

Centralized revocation capability: Revocation of client certificates is standardized. Revocation is controlled by the emitting certification authority. The status of a client certificate is known either by checking for its presence and revocation date in a Certificate Revocation List (CRL) or by querying an Online Certificate Status Protocol (OCSP) responder accredited by the emitting certification authority. These mechanisms are widely available, at least with Apache and Nginx support.

U2F lacks centralized revocation capability. A U2F keypair is manually registered at each website. Revocation of a token follows similar procedure, with the user manually unregistering the token from each website. No universal insurance can be brought that the user unregistered a token from all websites it was registered to.

Ease of use: Keypair generation on a smartcard is not part of the TLS authentication scheme *per se*. If a user expects to perform such an authentication, though, a keypair and a certificate are a prerequisite. No major browser capable of interacting with a smartcard using a PKCS#11 interface is providing a graphical user interface to generate the keypairs or the certificate signing request [13].

We could assume that the TLS certificate keypair was already generated and the certificate is loaded on the smartcard by the IT department of the company offering the website to authenticate to. We think this assumption is realistic for intranet websites, since this is already a common practice in corporate workstation smartcard authentication deployments. Aside from TLS authentication, credit (smart)cards are also delivered to consumers ready to use. As such, one could easily think of a internet banking website distributing smartcards within USB key form-factored reader for authentication of their client to their online banking website.

Even if the smartcard initialization is set aside, TLS authentication using client certificates is well-known for its usability issues [8]. Most web browsers provide labyrinthine user interfaces for certificate management. The matter gets even worse when using a smartcard holding the keypair

13. Such tools exist to store keypairs in the browser SSM.

and certificate, depending on the TLS client, PKCS#11 library and the operating system of the user.

On the other hand, U2F is almost plug and play [14]. Enrollment and authentication user experience can be customized by each RP with rich web interfaces.

Protocol maturity: TLS authentication of clients based on certificates is an authentication scheme specified since SSLv2, back in 1995. Studies of the TLS protocol are regularly published, yet few of these vulnerabilities pertain to the authentication of clients using certificates. The notable exceptions are the triple handshake attack [13] and the SLOTH attack [14] that were mitigated in most popular implementations.

On the other hand, U2F specifications were published in 2015. Our contribution is, to the best of our knowledge, the first report of an independent security study whatsoever.

Implementation maturity: Support of TLS client authentication using certificates is universal, with a support from all major web browsers and support from most programming language TLS libraries. The code is deemed mature enough by several popular website administrators to use it as their main authentication scheme. Examples of websites that use or used for several years TLS with client certificates authentication (with the private key stored in a SSM) encompass the StartSSL certification authority or the French tax administration website for individual tax payment. It is worth noting though that these examples are the rare exceptions. Indeed, the usability issues often result in this authentication scheme being dismissed. As a consequence, the implementation robustness might be less studied than if it was ubiquitous.

Concerning U2F, client-side support is currently limited to Chrome and Chromium web browsers, version 41 or higher. On the server-side, no reference open-source implementation of TLS Channel ID is known to the authors of this paper. Yubico provides several server-side libraries in several languages. Their code maturity, their compliance to the specifications and the size of their underlying community was not assessed during our study. An experimental patch for Wordpress support of the U2F authentication

14. GNU/Linux users need to insert a new set of rules in their udev configuration to enable unprivileged users to access the token, exposed as a raw Human Interface Device (HID) device. Some more customization of the operating system might be required, as illustrated by the libmtp problem that was presented in section 5.

scheme exists. Among the major actors, only Google, Dropbox and Github advertise some form of server-side support.

Unlinkability of accounts: Most smartcards save the cryptographic objects (e.g. keys and certificates) within the smartcard memory, which has very limited storage capacity. Users trying to ensure unlinkability of their accounts would rapidly need to own several smartcards. Also, users must be mindful not to send a certificate for a specific website to another one, as neither the protocol nor the user interface help preventing this.

U2F specifications, on the other hand, allow wrapping of the private keys within the key handle. The key handle is stored by the server and is provided to the token in each subsequent authentication request for unwrapping and signing operations. This allow for a virtually limitless number of keypairs per U2F tokens. As such, user accounts cannot be linked across websites using a shared public key or certificate. Users should however be aware that some tokens do not generate their keypairs from a fresh random seed. Instead, these tokens use values provided by the RP and the FIDO client along with a fixed secret value stored within the token memory. If a user or a group of users were to register several accounts at the same origin with such a token, all these accounts would share the same public key, and the server could tell that all accounts use the same U2F token. These accounts do not even need to exist simultaneously for this privacy issue to exist.

8 Conclusion

At first glance, our study tends to confirm that the security features offered by the U2F protocol are superior to the security brought by many SMS-based and TOTP-based second-factor solutions. This feat comes from the use of hardware secure elements, a phishing-resistant approach, the use of strong public key cryptography and an authentication scheme with distinct public keys per website.

Our conclusion regarding the U2F protocol cannot be definitive as we did not use any formal verification. It is also worth mentioning that we left out some elements of the specifications, such as the USB protocol and the so called *AppID and Facets* features. In particular, AppID need careful thinking as their purpose is to indicate to a FIDO client that some websites and non-web applications are associated to a particular web origin. In this case, there might be an unstudied security and privacy risk coming from colluding parties. Finally, it should be noted that our main focus was not on implementation robustness. As such we did not try to survey the various tokens nor Yubico's reference server-side implementations.

At a theoretical level, the web part of the U2F protocol seems sound. Although we did not mean to test the implementations, our hands-on verifications of the security properties revealed that some of them still need some maturation. For instance, this study allowed us to detect some discrepancies between the reference client implementation and the specifications. Also, several implementation recommendations are not followed at the time of writing. Among them are a user interface for transaction notification and the use of TLS Channel ID.

We contacted the FIDO Alliance and Google about these issues. We thank them for the quality of their feedbacks, their insight into the specifications and their willingness to help us further test our findings.

Overall, the U2F protocol seems like a viable second-factor authentication scheme with phishing-resistant properties. Its ease of use and privacy features further promotes it for a wide web deployment. Even so, at the time of writing, our comparison of U2F with the TLS authentication scheme relying on client certificates indicates that the latter should be used in environment with the most stringent security requirements.

References

1. FIDO Alliance website. `https://fidoalliance.org/`. Accessed: 2016-01-07.
2. FIDO Security Reference. `https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-security-ref.html`. Accessed: 2016-01-07, Version: 2015-05-14.
3. LibMTP. `http://libmtp.sourceforge.net/`.
4. Repository of 'python-u2flib-server' by Yubico. `https://github.com/Yubico/python-u2flib-server/`.
5. Token Binding IETF Working Group. `https://tools.ietf.org/wg/tokbind/`.
6. ZAP Proxy. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`.
7. J. Altman, N. Williams, and L. Zhu. Channel Bindings for TLS. RFC 5929, RFC Editor, 2010.
8. D. Balfanz. TLS Client Authentication. `http://www.browserauth.net/tls-client-authentication`. Accessed: 2016-01-07.
9. R. Hamilton D. Balfanz. Transport Layer Security (TLS) Channel IDs (Internet Draft). `https://tools.ietf.org/html/draft-balfanz-tls-channelid-01`. Accessed: 2016-01-07, Draft expired: 2013-12-31.
10. Dmitry Kurbatov. Hacking mobile network via SS7: interception, shadowing and more. `http://secuinside.com/archive/2015/2015-2-7.pdf`, 2015.

11. M'Raihi et al. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, RFC Editor, 2011.
12. Juan Lang and Alexei Czeskis and Dirk Balfanz and Marius Schilder and Sampath Srinivas. Security Keys: Practical Cryptographic Second Factors for the Modern Web. `http://fc16.ifca.ai/preproceedings/25_Lang.pdf`, February 2016.
13. Karthikeyan Bhargavan and Antoine Delignat-Lavaud and Cedric Fournet and Alfredo Pironti and Pierre-Yves Strub. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. `https://mitls.org/downloads/tlsauth.pdf`, May 2014.
14. Karthikeyan Bhargavan and Gaëtan Leurent. Security Losses from Obsolete and Truncated Transcript Hashes. `https://www.mitls.org/downloads/transcript-collisions.pdf`, February 2016.
15. Linus Walleij. Fear and Loathing in the Media Transfer Protocol. `http://events.linuxfoundation.org/sites/events/files/slides/Media%20Transfer%20Protocol.pdf`, 2014.
16. E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). RFC 5705, RFC Editor, 2010.
17. T. Dierks and E. Rescorla. On the Use of Channel Bindings to Secure Channels. RFC 5246, RFC Editor, 2008.