Formal Analysis of the FIDO 1.x Protocol

Olivier Pereira, Florentin Rochet, Cyrille Wiedling

UCL Crypto Group, Belgium

Abstract. This paper presents a formal analysis of FIDO, a protocol developed by the FIDO Alliance project, and which aims to provide either a passwordless experience or an extra security layer for user authentication over the Internet. We model the protocol using the applied pi-calculus and run our analysis using ProVerif. Our analysis shows that ignoring some optional steps of the standard could lead to the implementation of a flawed authentication process. On the contrary, we prove that these steps are sufficient to ensure the expected security properties.

1 Introduction

Authentication is a process arguably insecure under many forms. The most common one, password, has been criticized for years and many works have tried to propose a workable replacement [BHVOS12]. It is now widely accepted that relying on passwords only is insecure for sensitive applications.

The FIDO alliance [All] aims at establishing a standard for passwordless experience and second factor authentication. FIDO has received support from many companies around the world and has been integrated by various services such as banks (e.g. Bank of America), e-mails (e.g. Gmail), social networks (e.g. Facebook), etc. As those lines are written, the FIDO alliance claims to have brought FIDO capabilities to more than 3 billions of user accounts worldwide.

The protocol is modular and allows companies to build components in an inter-operable way. Their goal is to avoid the use of a server-side shared secret, by leveraging cryptographic capabilities of an hardware token that the user must own in order to obtain a successful authentication. The protocol works in a two-phases scenario. In the first phase, the user must register his token on the desired service. The registration process corresponds to the creation of a private/public key pair for which the public key is transmitted to the services and all the private keys are held in the hardware device. This first phase is performed for each of the services that the user wants to access to, but it must be performed once. The second phase is the authentication itself: it essentially consists in signing random challenges generated by the service, after approval by the user.

In this paper, we propose a formal analysis of the authentication methodology provided by the specifications of the U_2F standard, the "second factor experience", in which a FIDO token is used as a complement to a password on web services. These specifications provide the description of the protocol and some optional features. We focused our analysis on one of these features and, using the tool ProVerif [Bla01], we prove that without it, there exists an attack

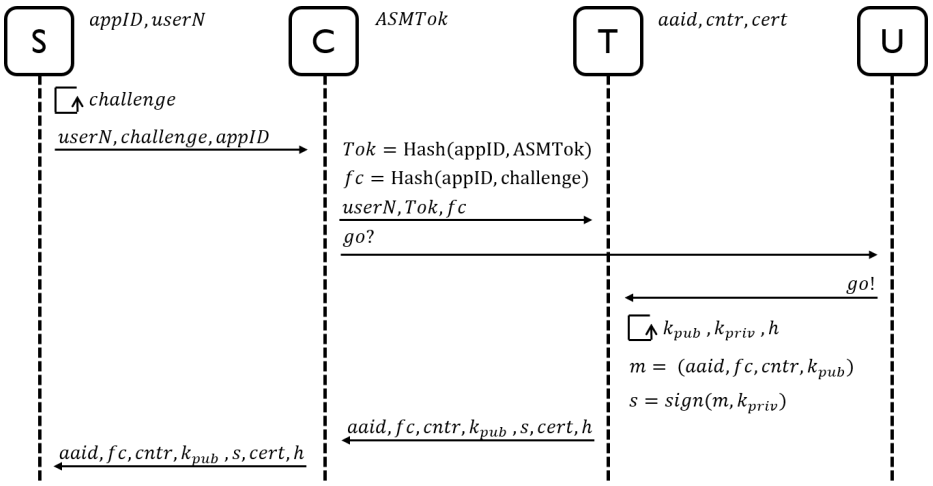

Fig. 1. Simplified Registration Step. S: Server, C: Client, T: Token, U: User

where a web-server could impersonate a user to a third party service, under the assumption of the use of weak passwords, which is precisely the issue that the FIDO U_2F protocol is expected to solve. We also provide a local test-bed attack scenario to illustrate our finding. On the positive side, we prove that, if this feature is properly implemented, then the expected security properties hold.

Roadmap. Sections 2 and 3 present the U_2F protocol and the applied pi-calculus used for our analysis. In Section 4, we define a model of the protocol and a definition of the authentication property. In Section 5, we present and analyze our results. Section 6 gives the related work.

2 The FIDO Protocol

The FIDO protocol aims to authenticate a user to a server, using a token (e.g. smartcard, USB token, etc.), in such a way that is not possible to impersonate a user without being in possession of his token, even if the username and the password of that user have been compromised. The protocol runs between a user, a Token, a FIDO Client embedded in the user's web browser, and a server, after the establishment of a secure TLS between the last two entities. As described above, it is composed of two main phases: the registration and the authentication.

2.1 Registration Phase

The registration step is used to link a token to the account of a user. Figure 1 offers a high-level view of its behavior. (A full version can be found at [All].) The first thing that happens when the FIDO Client is installed on a computer is the generation of a random *ASMTok*, which is used to provide a link between the key handle that will be created on the Token and the Client. The Token possesses

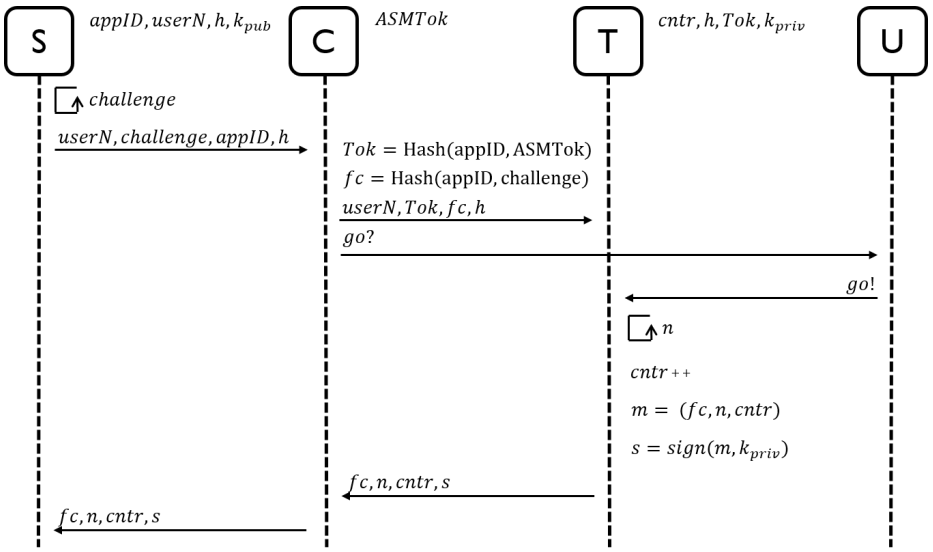

Fig. 2. Simplified Authentication Step.

different items: its unique identifier, *aaid*, a counter *cntr*, which is incremented each time it performs a signature for an authentication; and a certificate *cert*, which is delivered by the manufacturer of the Token, corresponding to the master key, which is used to sign the generated key pairs.

The registration then proceeds as follows: for a given User, identified by a username, *userN*, the Server generates a *challenge* and sends it to the Client, along with the username, and the *appID* identifying the server (e.g. its url). Then, the Client computes two values, *Tok* and *fc*, the first one links the *appID* to the secret token *ASMTok*, the second one links the *appID* to the *challenge*. Those two values are given to the Token. The latter waits for the User the permission to perform the registration step (e.g. the Client displays a message asking for the User to push a button on the Token). Once the User approves, the Token generates a key pair and stores the private key, with *Tok*, in a handle h. Using its identifier *aaid* and a counter *cntr*, it creates a message and signs it before sending it back to the Client, with the identifier of the handle h and the certificate *cert*. The Client forwards everything to the Server, which checks the certificate and the signature before adding the public key to the User's information.

2.2 Authentication Phase

The authentication step (see Fig. 2) is quite similar to the registration one. Once the User has provided his username and password, if the Server has a Token registered to that account, it will generate a challenge and send it to the Client, along with the handle h it got from the registration step. As during the registration step, the Client will create *Tok* and *fc*, and send it to the Token before asking to the User to approve the authentication (e.g. by pushing the

Token's button). Then, the Token checks if it possesses an entry h and whether its content match the *Tok* given by the Client. If it is, then the Token generates a nonce n, increments its counter *cntr* and uses the secret key stored in the entry h to sign a message. This message is passed to the Client, along with n, *fc* and *cntr*, which transmits them to the Server. Using the public key it has registered with the account, the Server will check whether the signature is correct or not, and provide access in the former case.

3 Applied Pi-Calculus

This section briefly presents the notations of the applied pi-calculus, a process algebra introduced by M. Abadi et C. Fournet [AF01], often used to model protocols and security properties.

3.1 Terms

Messages are represented by *terms* built upon an infinite set of *names* $\mathcal{N}$ (for communication channels or atomic data), a set of *variables* $\mathcal{X}$ and a *signature* Σ consisting of a finite set of *function symbols* (to represent cryptographic primitives). A function symbol f is assumed to be given with its arity $\mathsf{ar}(f)$. Then, the set of terms $T(\Sigma, \mathcal{X}, \mathcal{N})$ is formally defined by the following grammar:

$$
\begin{array}{lll}
t, t_1, t_2, \ldots ::= & & \\
\quad x & & x \in \mathcal{X} \\
\quad n & & n \in \mathcal{N} \\
\quad f(t_1, \ldots, t_n) & & f \in \Sigma,\ n = \mathsf{ar}(f)
\end{array}
$$

In order to represent the properties of the primitives, the signature Σ is equipped with an *equational theory* E that is a set of equations which hold on terms built from the signature. We denote by $=_E$ the smallest equivalence relation induced by E, closed under application of function symbols, substitutions of terms for variables and bijective renaming of names. We write $M =_E N$ when the equation $M = N$ holds in the theory E.

3.2 Processes

Processes and *extended processes* are defined in Figure 3. The process 0 represents the null process that does nothing. $P \mid Q$ denotes the parallel composition of P with Q while $!P$ denotes the unbounded replication of P (*i.e.* the unbounded parallel composition of P with itself). $\nu n.P$ creates a fresh name n and then behaves like P. The process if ϕ then P else Q behaves like P if ϕ holds and like Q otherwise. $u(x).P$ inputs some message in the variable x on channel u and then behaves like P while $\overline{u}\langle M\rangle.P$ outputs M on channel u and then behaves like P. We write $\nu\tilde{u}$ for the (possibly empty) series of pairwise-distinct binders $\nu u_1 \ldots . \nu u_n$. The active substitution $\{^M/_x\}$ can replace the variable x for the term M in every process it comes into contact with and this behavior can be controlled by restriction, in particular, the process $\nu x \left(\{^M/_x\} \mid P\right)$ corresponds exactly to let $x = M$ in P.

$P, Q, R ::=$	(plain) processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\nu n.P$	name restriction
if ϕ then P else Q	conditional
$u(x).P$	message input
$\overline{u}\langle M\rangle.P$	message output
$A, B, C ::=$	extended processes
P	plain process
$A \mid B$	parallel composition
$\nu n.A$	name restriction
$\nu x.A$	variable restriction
$\{^M/_x\}$	active substitution

Fig. 3. Syntax for processes

As in [CS13], we slightly extend the applied pi-calculus by letting conditional branches now depend on formulae defined by the following grammar:

$$\phi, \psi ::= M = N \mid M \neq N \mid \phi \wedge \psi$$

If M and N are ground, we define $[\![M = N]\!]$ to be true if $M =_E N$ and false otherwise. The semantics of $[\![\ ]\!]$ is then extended to formulae as expected.

The *scope* of names and variables is delimited by binders $u(x)$ and νu. Sets of bound names, bound variables, free names and free variables are respectively written $\mathsf{bn}(A)$, $\mathsf{bv}(A)$, $\mathsf{fn}(A)$ and $\mathsf{fv}(A)$. Occasionally, we write $\mathsf{fn}(M)$ (resp. $\mathsf{fv}(M)$) for the set of names (resp. variables) which appear in term M. An extended process is *closed* if all its variables are either bound or defined by an active substitution. A *context* $C[\_]$ is an extended process with a hole instead of an extended process. We obtain $C[A]$ as the result of filling $C[\_]$'s hole with the extended process A. An *evaluation context* is a context whose hole is not in the scope of a replication, a conditional, an input or an output. A context $C[\_]$ closes A when $C[A]$ is closed. A *frame* is an extended process built up from the null process 0 and active substitutions composed by parallel composition and restriction. The *domain* of a frame φ, denoted $\mathsf{dom}(\varphi)$ is the set of variables for which φ contains an active substitution $\{^M/_x\}$ such that x is not under restriction. Every extended process A can be mapped to a frame $\varphi(A)$ by replacing every plain process in A with 0.

3.3 Operational Semantics

The operational semantics of processes in the applied pi-calculus is defined by three relations: *structural equivalence* ($\equiv$), *internal reduction* ($\rightarrow$) and *labelled reduction* ($\xrightarrow{\alpha}$).

$$
\begin{array}{lrll}
\textsc{Par} - 0 & A & \equiv & A \mid 0 \\
\textsc{Par-A} & A \mid (B \mid C) & \equiv & (A \mid B) \mid C \\
\textsc{Par-C} & A \mid B & \equiv & B \mid A \\
\textsc{Repl} & !P & \equiv & P \mid !P \\
\textsc{New} - 0 & \nu n.0 & \equiv & 0 \\
\textsc{New-C} & \nu u.\nu w.A & \equiv & \nu w.\nu u.A \\
\textsc{New-Par} & A \mid \nu u.B & \equiv & \nu u.(A \mid B) \qquad \text{if } u \notin \mathsf{fv}(A) \cup \mathsf{fn}(A) \\
\textsc{Alias} & \nu x.\{^M/_x\} & \equiv & 0 \\
\textsc{Subst} & \{^M/_x\} \mid A & \equiv & \{^M/_x\} \mid A\{^M/_x\} \\
\textsc{Rewrite} & \{^M/_x\} & \equiv & \{^N/_x\} \qquad \text{if } M =_E N
\end{array}
$$

Fig. 4. Structural equivalence.

Structural equivalence is defined in Figure 4. It is closed by α-conversion of both bound names and bound variables, and closed under application of evaluation contexts. Structural equivalence corresponds to some structural rewriting that does not change the semantics of a process. The *internal reductions* and *labelled reductions* are defined in Figure 5. They are closed under structural equivalence and application of evaluation contexts. Internal reductions represent evaluation of condition and internal communication between processes. Labelled reductions represent communications with the environment.

4 Modeling FIDO

In this section, we present our model, in applied pi-calculus, of the FIDO protocol, and the definition of the authentication property it is aimed to achieve.

4.1 Settings

We focus our analysis on the FIDO protocol itself, and as such, our model starts after the establishment of the TLS channel between the Client and the Server. Thus, we consider a secure channel between these two. Although this channel may be out of reach from an attacker in a fully honest setting, we will consider different corruption scenarios that will grant the attacker access to it and therefore will not limit our analysis.

We consider the following entities:

- **U**ser: represents the person that is willing to connect to the Server through the Client, using the Token.
- **T**oken: represents the device that stores and uses the different keys used for authentication to the Server.
- **C**lient: represents the embedded Client in the browser of the User, used to established the connection to the Server.
- **S**erver: represents the service the User wants to connect to, using the Client.

Let us define notations for the communication channels (see Figure 6):

$$\text{(Comm)}\quad \overline{c}\langle M\rangle.P \mid c(x).Q \to P \mid Q\{^{M}/_{x}\}$$

$$\text{(Then)}\quad \text{if } \phi \text{ then } P \text{ else } Q \to P \ \text{ if } [\![\phi]\!] = \mathsf{true}$$

$$\text{(Else)}\quad \text{if } \phi \text{ then } P \text{ else } Q \to Q \ \text{ otherwise}$$

$$\text{(In)}\quad c(x).P \xrightarrow{c(M)} P\{^{M}/_{x}\}$$

$$\text{(Out-Atom)}\quad \overline{c}\langle u\rangle.P \xrightarrow{\overline{c}\langle u\rangle} P$$

$$\text{(Open-Atom)}\quad \frac{A \xrightarrow{\overline{c}\langle u\rangle} A' \qquad u \neq c}{\nu u.A \xrightarrow{\nu u.\overline{c}\langle u\rangle} A'}$$

$$\text{(Scope)}\quad \frac{A \xrightarrow{\alpha} A' \qquad u \text{ does not occur in } \alpha}{\nu u.A \xrightarrow{\alpha} \nu u.A'}$$

$$\text{(Par)}\quad \frac{A \xrightarrow{\alpha} A' \qquad \mathsf{bv}(\alpha) \cap \mathsf{fv}(B) = \mathsf{bn}(\alpha) \cap \mathsf{fn}(B) = \emptyset}{A \mid B \xrightarrow{\alpha} A' \mid B}$$

$$\text{(Struct)}\quad \frac{A \equiv B \qquad B \xrightarrow{\alpha} B' \qquad B' \equiv A'}{A \xrightarrow{\alpha} A'}$$

where α is a *label* of the form $c(M)$, $\overline{c}\langle u\rangle$, or $\nu u.\overline{c}\langle u\rangle$
such that u is either a channel name or a variable of base type.

Fig. 5. Semantics for processes

- $\mathsf{c_I}$ is a secure channel between the Client and the Server. It models the TLS-secured channel between the two entities.
- $\mathsf{c_T}$ is the secure channel between the Client and the Token. Basically, it models the USB (or NFC/Bluetooth) connection.
- $\mathsf{c_E}$ is the channel between the Client and the User, i.e. the screen of the computer where the Client may display information to the User.
- $\mathsf{c_F}$ is the channel between the User and the Token. It is just modeling the possibility, for the User, to push the Token's button to allow authentication.

4.2 Threat Model

In our model, like in numerous formal analyses, we consider an attacker who has control over the network. Therefore, he is able to intercept, forge, send messages. Restrictions are based on the nature of channels. An attacker can listen to, but not send messages over authenticated channels. Public channels are, by definition public, and everyone can listen to or send messages over them.

During this analysis, we consider two main scenarios:

- Everyone is honest. It is a classical scenario where the attacker only has access to the network (and non-secure channels).

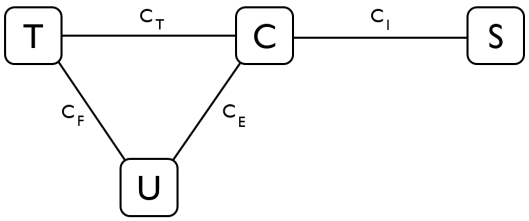

Fig. 6. Channels between the different players of the FIDO protocol.

– Either the Server or the Client is dishonest.

Note that we cannot corrupt both the Client and the Server at the same time, otherwise it would be impossible to guarantee anything. We also do not consider cases where the User may be corrupted, because the User is just an entity trying to connect to a service – which is meaningless to corrupt. The Token stores keys and an attacker may want to try to break it, but if such a Token is corrupted we can not guarantee any security on the second level authentication since the attacker would have access to the keys used for authentication.

4.3 Authentication Property

The FIDO protocol aims to provide authentication to the user. Once registered and linked to an account, the token is designed to be necessary in order to perform an authentication on the server. The user should provide login and password, as usual, but the server will also require a signature coming from the token, which is given if the user pushes a button on the token itself. In our study, we explore this authentication property, in order to see if there is any possibility for an attacker to impersonate a user on a server.

Definition 1. *Let us consider different predicates:*

- $\mathsf{LoggedIn}(appID, username, pk)$ *is issued by the server, identified by* $appID$*, when he has accepted (by finishing the protocol) a connection with the user, identified by* $username$*, and referenced under the public key* pk*.*
- $\mathsf{ExpConnection}(appID, username, s)$ *is issued by the user, secretly identified by* s*, asserting that he expects a connection to* $appID$*, with his* $username$*.*
- $\mathsf{PushButton}(s)$ *is issued by the user when he pushes the button on the token.*
- $\mathsf{TokenSign}(sk)$ *is issued by the token when it signs using the secret key* sk*.*

Using these predicates, we define the authentication property:

$$\begin{aligned}\forall\ & appID, username, pk,\\ & \mathsf{LoggedIn}(appID, username, pk) \Longrightarrow \exists\ s, sk\\ & \qquad \mathsf{TokenSign}(sk) \wedge (pk = \mathsf{pk}(sk)) \wedge\\ & \qquad \mathsf{PushButton}(s) \wedge\\ & \qquad \mathsf{ExpConnection}(appID, username, s)\end{aligned}$$

This property reflects that each time a server accepts a connection, then a user first expected to initiate a connection to this server and pushed on the button of the token, which generated the expected signature.

4.4 Modeling the Protocol

Since we are focusing on the authentication property, we model the authentication part of the protocol, assuming that all the needed registrations have been already performed, with no interference of an attacker.

Server After a registration, the Server knows, for each username registered: the public key pk, the handle h pointing to the corresponding secret key inside the Token, and the status of the counter $scntr$. During the authentication phase, when the User has provided his username and password, the Server will generate a new challenge n_s and send its $AppID$, h and n to the Client. Then, it will wait for a signature s before checking it. He also checks whether the counter sent by the token is larger than the one stored. If the verifications succeed, then the Servers accepts the login.

$$
\begin{aligned}
&S(appID, username, pk, h, cntr) = \\
&\quad \nu n_s; \\
&\quad \overline{c_I}\langle(appID, n_s, h)\rangle; \\
&\quad c_I((x)); \\
&\quad \text{let } (x_1, x_2, x_3, x_4) = x \text{ in} \\
&\quad \text{if } \mathsf{checksign}((x_1, x_2, x_3), x_4, pk) = \mathsf{ok} \wedge\ (x_3 > cntr) \text{ then} \\
&\quad \mathsf{LoggedIn}(appID, username, pk)
\end{aligned}
$$

Client The Client waits for the inputs of the Server, then verifies that the $appID$ matches the one it is expecting (a). If the condition statement succeeds, then the Client computes hashes that will be passed to the Token for the signature, using its secret value t_c. The Client then prompts a message to the User, asking for a confirmation to the Token, and waits for a response from the latter, before passing the signature to the Server. We also consider another Client model (C_c) where the IF-condition is missing, since this step is optional in the specification (see [All16b, Section 5.2.1] and our discussion in Section 5.2).

$$
\begin{aligned}
&C(a_c, t_c) = \\
&\quad c_I((x)); \\
&\quad \text{let } (x_1, x_2, x_3) = x \text{ in} \\
&\quad \text{if } (x_1 = a) \text{ then} \\
&\quad \overline{c_E}\langle go?\rangle; \\
&\quad \text{let } KToken = \mathsf{hash}((x_1, t_c)) \text{ in} \\
&\quad \text{let } fc = \mathsf{hash}((x_1, x_2)) \text{ in} \\
&\quad \overline{c_T}\langle(x_3, KToken, fc)\rangle; \\
&\quad c_T(y); \\
&\quad \overline{c_I}\langle y\rangle;
\end{aligned}
\qquad
\begin{aligned}
&C_c(t_c) = \\
&\quad c_I((x)); \\
&\quad \text{let } (x_1, x_2, x_3) = x \text{ in} \\
&\quad \text{let } KToken = \mathsf{hash}((x_1, t_c)) \text{ in} \\
&\quad \text{let } fc = \mathsf{hash}((x_1, x_2)) \text{ in} \\
&\quad \overline{c_E}\langle go?\rangle; \\
&\quad \overline{c_T}\langle(x_3, KToken, fc)\rangle; \\
&\quad c_T(y); \\
&\quad \overline{c_I}\langle y\rangle;
\end{aligned}
$$

User The User does not do much except for sending a go-message to the Token. We introduce a secret s_h, which captures the assumption than the button can only be pressed by the User: no one can push the button on the Token, except the User. During authentication, the User waits for the Client's prompt, and then, pushes the button of the Token to allow the authentication.

$$
\begin{aligned}
&U(\mathit{ExpAppID}, \mathit{username}, s_h) = \\
&\quad \mathsf{ExpConnection}(\mathit{appID}, \mathit{username}, s_h); \\
&\quad c_E((x)); \\
&\quad \overline{c_F}\langle(\mathit{go}!)\rangle; \\
&\quad \mathsf{PushButton}(s_h)
\end{aligned}
$$

Token During the registration step, the Token stored, under a handle h, the secret key sk, and a token KToken generated by the Client. The Token itself is identified by an aaid and updates a counter, cntr, which is used to avoid replay attacks. During the authentication step, the Token waits for inputs from the Client and the confirmation of the User (e.g. by pushing a button). It also checks whether the handle and the token are valid and provides a signature using the corresponding secret key if everything is in order.

$$
\begin{aligned}
&T(\mathit{aaid}, \mathit{cntr}, h, \mathit{KToken}, sk) = \\
&\quad c_T((x)); \\
&\quad \text{let } (x_1, x_2, x_3) = x \text{ in} \\
&\quad c_F((x)); \\
&\quad \text{if } (x_1, x_2) = (h, \mathit{KToken}) \text{ then} \\
&\quad \nu n_t; \\
&\quad \text{let } s_t = \mathsf{sign}((x_3, n_t, \mathit{cntr}), sk) \text{ in} \\
&\quad \mathsf{TokenSign}(sk); \\
&\quad \overline{c_T}\langle(x_3, n_t, \mathit{cntr}, s_t)\rangle
\end{aligned}
$$

FIDO Protocol The authentication part can now be modeled by placing all processes in parallel, which is writen as follows:

$$P_a = \nu\tilde{n}, a.!\left[S(a, p_u, pk_u, h, c) \mid C(a, s_c) \mid T(a, c, h, tok, k_u) \mid U(a, p_u, s_u) \mid \Gamma\right].$$

where $\tilde{n} = a, k_u, s_u, p_u, h, c, s_c, pk_u, tok$ are bound names which represent the different items created during the registration phase that are used as arguments for our processes, and $\Gamma = \{^{\mathsf{pk}(k_u)}/_{pk_u}, ^{\mathsf{hash}((a,s_c))}/_{tok}\}$ is a frame describing the content of tok and pk_u.

We also model the case where the Client model does not verify the appID:

$$P_c = \nu\tilde{n}.!\left[S(a, p_u, pk_u, h, c) \mid C_c(s_c) \mid T(a, c, h, tok, k_u) \mid U(a, p_u, s_u) \mid \Gamma\right].$$

5 ProVerif Results

In this section, we present our results, obtained using the ProVerif tool, studying the authentication property of the FIDO protocol in various scenarios.

5.1 Client Model with AppID Verification

To analyze the FIDO protocol with respect to the authentication property, we choose to use the ProVerif tool, developed by B. Blanchet [Bla01], which is an automated verifier that can achieve to prove injection properties (and more), for protocols. It has been used for various examples (like [KR05],[KT08]) and also has some limitations (e.g. [CW12]), especially when protocols get too complicated (e.g., involving several cryptographic primitives), but the FIDO protocol only involves signatures and hashes, and remains within ProVerif's scope. Our ProVerif files can be found at [WRP].

Results Our results, obtained using ProVerif, are compiled in Table 1. We show that even if we corrupt the Server, or the Client (but never both of them), there is no possible attack. The server-in-the-middle case is more interesting: we suppose that a user registered to two different servers and does not know that one of them is corrupted. We also consider that one of them is corrupted and knows the login and password of the user for the honest one. (e.g., it could be possible that the user uses the same couple login/password for the two servers.)

5.2 Client Model without AppID Verification

In the model above, we check if the *appID* provided by the server matches the origin of the request. The *appID* verification is correctly implemented in the Chrome FIDO Client but nothing prevents an other implementation to miss this important step. Indeed, this is an ambiguous step in the FIDO specifications, where nothing is requested in the UAF protocol description. In particular, we found the following recommendation:"The FIDO client should perform the following steps: - Verify the application identity of the caller."[All16b]

It is clearly stated as a recommendation but, for reasons that we will detail later, we believe that this should be an assertion (MUST). The optional character of this recommendation is indirectly confirmed in other official documents. For instance, the overview [All] reads: "Say a user has correctly registered a U2F device with an origin and later, a MITM on a different origin tries to intermediate the authentication. In this case, the user's U2F device won't even respond, since the MITM's (different) origin name will not match the Key Handle that the MITM is relaying from the actual origin."

The above statement ignores a possible *appID* verification since if the device does not respond, then, it must have seen the request despite the different origin. Therefore, the verification is missing, otherwise the data would not have been sent to the device. Moreover, since the token is never given the actual origin of a request, it cannot perform such a verification.

Results As we see in Table 1, ProVerif outputs that the authentication property does not hold anymore. In practice, it corresponds to the following attack: when the User authenticates on Server A (Server ITM), the Server initiates an authentication towards Server B and forwards the challenge to the victim. Without the

Table 1. ProVerif results on our authentication property.

Corrupted Players	Without AppID Verification	With AppID Verification
None	✓	✓
Server	✓	✓
Client	✓	✓
Server + Client	×	×
None + Server ITM	×	✓

appID verification, the User provides a valid Server B authentication credential that Server A can use to impersonate the User on Server B.

In practice, even if the cryptographic primitives are correctly implemented, forgetting to verify the *appID* is enough to impersonate a User in a plausible attack scenario. Moreover, if FIDO is used as a second factor authentication, the Server ITM might already have access to the User password and login through its own database, since the victim might use the same password across multiple services, which is a scenario against which the Token should offer a protection.

As a result, we believe that our attack scenario violates at least the following security goal stated in the specification [All16a]:

– **SG-3** (credential disclosure resilience), in the sense that a loss of credentials is sufficient to run the man-in-the-middle attack described above and successfully bypass the two levels of authentication.

Corresponding Attack on the Real Token We confirm the ProVerif's results in practice by providing a working java web server implementation [Roca] that can run either as a honest server or as a corrupted server. We tested our attack with Chrome 40 [Goo] packaged for Debian and with our modified FIDO Client [Rocb]. We had to modify the FIDO Client provided by Google to remove the *appID* verification that was hopefully correctly integrated. A tutorial is provided in Appendix A to reproduce the attack.

Lesson learned FIDO is secure if we assume both that the implementer understood the importance of the *appID* verification despite its optional character in the specification. Moreover, we have also to assume that there is no possibility to fool the origin verification in the browser. We recommend the specification to enforce the *appID* verification: among other changes, the *appID* verification must not be written as a recommendation (using a should) but instead as an assertion (using a must). Also, these results suggest that current FIDO existing Clients should be audited to check if the *appID* verification exists and does not suffer from any weakness. Very few such clients are freely available, though.

6 Related Work

In 1996, Lowe [Low96] raised the interest in analysing authentication protocol when he presented a MITM attack against Needham and Schroeder public key

protocol [NS78]. Later, the progress in web technologies drove a consortium of internet companies to unite and design a user-friendly standard identity management and an authentication protocol. This initiative was called the Liberty Alliance Project. A research group, led by Pfitzmann, laid the basis for formal analysis of such web-based identity management protocol [GPS05,PW03]. They did not designed automated formal analysis tools but they discussed formal descriptions of those protocols. Eventually, the Liberty Alliance produced SAML 2.0, an open-standard data format to exchange information between parties. These specifications lead to SSO protocols (Single-Sign-On) used for cross-authentication. The SSO protocol implemented by Google got broken with automated formal analysis [ACC+08], where researcher found how to impersonate a user when acting as a dishonest service, at another service provider. For years now, browsers have gained cryptographic capabilities and many authentication protocols appeared and gained in complexity. The need for mechanical analysis got more evident, since modelling the protocol became the time-consuming task and proofs were provided by the tool. In this line of work, Bortolozzo et al.[BCFS10] designed a tool to audit PKCS#11 tokens and found weaknesses on many of them. Some other tools have been designed to automate the process of proving security properties, such as the one we use in this paper: ProVerif [Bla01] but also APTE [Che14] and aKiss [CCK12]. Other hardware tokens, such as Yubikeys doing one-time passwords, have been analysed with other automated tools [KS12,KK16]. They are mostly chosen depending on the property we want to prove. Some are easier to use in some circumstances.

7 Conclusion

We modeled the authentication phase of the FIDO protocol in Applied Pi Calculus. We considered two different client models, based on the execution or not of a verification step by the client, step that is left optional in the specification.

Our ProVerif analysis shows that, when the verification is performed, the expected authentication properties are satisfied. However, when it is ignored, a man-in-the-middle attack becomes possible, assuming compromised (or reused) credentials, which definitely falls within the scope of the attacks that the use of a FIDO token is expected to prevent.

As a result, we recommend making this verification step mandatory, and that the authors of the 49 certified client implementations listed on the FIDO Alliance website check whether this step is actually performed.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable feedback. This work was partially supported by the Innoviris C-Cure project and the Region Wallonne TrueDev project.

References

[ACC+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10. ACM, 2008.

[AF01] Martin Abadi and Cédric Fournet. Mobile Values, New names, and Secure Communication. In *28th ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[All] FIDO Alliance. FIDO Documentation. https://fidoalliance.org/specifications/download/.

[All16a] FIDO Alliance. Fido security reference. `https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-security-ref-v1.1-id-20160915.html`, 9 2016.

[All16b] FIDO Alliance. FIDO U2F JavaScript API. `https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-javascript-api-v1.1-id-20160915.html`, 9 2016.

[BCFS10] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and Fixing PKCS#11 Security Tokens. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[BHVOS12] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 553–567. IEEE, 2012.

[Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, 2001.

[CCK12] Rohit Chadha, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 108–127. Springer-Verlag, 2012.

[Che14] Vincent Cheval. APTE: An Algorithm for Proving Trace Equivalence. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014.

[CS13] Véronique Cortier and Ben Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 2013.

[CW12] V. Cortier and C. Wiedling. A formal analysis of the Norwegian E-voting protocol. In *First International Conference on Principles of Security and Trust (POST)*, 2012.

[Goo] Google. Chrome browser download. `http://google-chrome.en.uptodown.com/ubuntu/old`. Accessed: 2016-01-13.

[GPS05] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Browser model for security analysis of browser-based protocols. In *ESORICS*, volume 3679, pages 489–508. Springer, 2005.

[KK16] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

[KR05] Steve Kremer and Mark Ryan. Analysis of an Electronic Voting Protocol in the Applied Pi Calculus. In *14th European Symposium on Programming (ESOP)*, 2005.

[KS12] Robert Künnemann and Graham Steel. Yubisecure? formal security analysis results for the yubikey and yubihsm. In *International Workshop on Security and Trust Management*, pages 257–272. Springer, 2012.

[KT08] Ralf Küsters and Tomasz Truderung. Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach. *CoRR*, 2008.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[NS78] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[PW03] Birgit Pfitzmann and Michael Waidner. Federated identity-management protocols. In *Security Protocols Workshop*, volume 3364, pages 153–174. Springer, 2003.

[Roca] Florentin Rochet. Fido compliant library and java web application example. `https://github.com/frochet/java-u2flib-server`. Accessed: 2016-01-13.

[Rocb] Florentin Rochet. Modified fido client as a chrome extension. `https://github.com/frochet/u2f-ref-code`. Accessed: 2016-01-13.

[WRP] Cyrille Wiedling, Florentin Rochet, and Olivier Pereira. Proverif implementation of the fido protocol. `https://git-crypto.elen.ucl.ac.be/frochet/fido_proverif`.

A Attack on a real FIDO authentication

We detail the steps to test the attack from Section 5.2 in a local test-bed.

- Clone repositories [Roca], [Rocb] and download chrome [Goo]
- From chrome tab extensions, activate the developer mode and load the unpacked u2f-chrome-extension.
- Inside java-u2flib-server, do *mvn clean install* then cd inside u2f-server-demo and configure two .yml files, one for the compromised server and one for the honest server. Use the available model, you just need to provide a different port number.
- Run:
  - java -jar target/u2flib-server-demo.jar server [your_consigDishonest_file.yml] localhost [port_honest_server] https://localhost:[port_dishonest_server] &
  - java -jar target/u2flib-server-demo.jar server [your_consigHonest_file.yml] https://localhost:[port_honest_server] &
- Use chrome and your FIDO-compliant authenticator to register in both services then try to authenticate.