

Practicum Final Report

Solving the Problem of Password-Based User Authentication

Steve Vaillancourt
svaillancourt3@gatech.edu
GeorgiaTech

Information Security Track
CS-6727

Contents

Table of Figures.....	4
1. Solving the Password-Based Authentication Problem	5
1.1 Problem Statement	5
1.2 Server-Side Protection of Shared Secrets	6
1.3 Analysis of Existing Solutions	7
1.3.1 Kerberos and Lightweight Directory Access Protocol (LDAP)	8
1.3.2 Security Assertion Markup Language (SAML)	8
1.3.3 WebAuthn + Client to Authenticator Protocol (CTAP) = Fast ID Online (FIDO)	9
2. Designing the Solution.....	11
2.1 Solution Statement.....	11
2.2 Advantages of Passwords for Authentication	11
2.3 Reducing User Burden with App Solution	12
2.4 Technical Context of the Solution	13
3. Implementation of the Solution.....	14
3.1 Authentication Protocol	14
3.2 Authenticator Tokens	16
3.3 Digital Signatures	17
3.4 Server-Side Identity and Access Management.....	19
3.5 Client-Side Protection of Private Keys	22
3.6 Access Controls	22
3.7 Vault	24
3.8 Examples of Access Control Schemes	25
3.8.1 Password-protected Access	25
3.8.2 TPM-Protected Access.....	25
3.8.3 Yubikey Static Password	25
3.8.4 USB Security Key	25
3.8.5 Yubikey with Biometric Access	25
3.8.6 Fingerprint-Protected USB Key.....	26
4. Assessment of the solution	27
5. Security Analysis	30
5.1 Risk Analysis Summary	30
5.2 Client-Side Security Recommendations.....	30
5.2.1 For Authenticator Use	30

5.2.2 For Privacy and Confidentiality	30
5.2.3 For Endpoint Resilience	31
5.2.4 For Private Key Protection	31
5.3 Server-Side Security Recommendations	31
5.3.1 For Account Management.....	31
5.3.2 For Account Monitoring.....	31
5.3.3 For Protection of Authenticated Sessions	31
5.3.4 For Endpoint Resilience	31
6. Limitations and Future Work	32
6.1 Platform-independence and interoperability	32
6.2 Security Issues	32
6.3 Technical Skill Requirements	33
6.4 Continuous Remediation.....	33
6.4 Endpoint Compromise	33
7. Conclusion	34
8. Appendix A – Risk Analysis	35
9. Appendix B – Digital Signature Validation Server-Side	46
9.1 Python Script to Validate RSA, ECDSA, or DSA Signatures	46
10. Appendix C – Client Enrollment with CA-Signed Certificate	51
10.1 Using CA-Signed Certificates	51
10.2 Sample PHP Code.....	52
11. Appendix D – User Manual.....	54

Table of Figures

Figure 1: SAML Identity Provider SSO Flow (from https://www.pingidentity.com/en/resources/client-library/articles/saml.html).....	8
Figure 2: XML-Encoded User Registration Message from the Server.....	15
Figure 3: Class Diagram of the EasyAuthentication Protocol.....	16
Figure 4: Class Diagram of Digital Signatures in the Authenticator	18
Figure 5: Database Schema for User-Related Information	20
Figure 6: Server-side User Authentication in PHP	21
Figure 7: Class Diagram for the Protection of Private Keys.....	24
Figure 8: Yubikey C Bio (from https://www.yubico.com/ca/product/yubikey-c-bio/)	25
Figure 9: Fingerprint-Protected USB Key.....	26
Figure 10: Database Schema for X509 Certificate Use	51

1. Solving the Password-Based Authentication Problem

1.1 Problem Statement

Password-based authentication schemes in use today place an inordinate amount of responsibility on the end-user for security with disastrous results: too many passwords to remember, simplistic and predictable passwords, password reuse, and stolen passwords through social engineering. They are inadequately protected by both users and online providers, causing hundreds of millions of passwords to be exploited every year.¹ Because a password-based solution implies sharing a secret with the online providers, this also requires that we trust implementers to deploy security controls server-side to protect these secrets against cyber attacks. Stronger methods that leverage multi-factor authentication do provide additional security but unavoidably increase the complexity for the end-user and the server-side implementation. Solutions that are easy to implement and easy to use are almost non-existent if they are to provide adequate security. In his essay "Stop Trying to Fix the User"², Bruce Schneier aptly describes how security that depends on shaping the user to security rules that are counter-intuitive and complex is bound to fail. Bruce Schneier states several examples that illustrate the issue such as telling users to not click on links, not inserting USB keys, and using passwords that are difficult to remember. Bruce Schneier states:

Traditionally, we've thought about security and usability as a tradeoff: a more secure system is less functional and more annoying, and a more capable, flexible, and powerful system is less secure. This "either/or" thinking results in systems that are neither usable nor secure.

Passwords display this duality between usability and security. They can be secure if they are long, complex, seemingly random, not written down, and unique to a use. In other words, with very reduced usability.

A consensus among cybersecurity experts is that two-factor or multifactor authentication (2FA/MFA) should be adopted to increase the security of the authentication. This means that the solution should involve a combination of authentication credentials between "something you know", "something you have" and "something you are". In a recent event³, Anne Neuberger stated "*5 basic but important cybersecurity practices*" to impede the capabilities of adversaries to damage networks and exploit data. At number 2, was "*Multifactor authentication, because we know passwords are dead*"⁴. Later in that discussion, Chris Inglis, National Cyber Director, states:

We just need to make sure that technology supports the human being, as opposed to confounds the human being, [Anne] talked about an example coming over of being in a car which you didn't have to independently go out and buy and airbag, and figure out how to install the thing, and

¹ <https://financesonline.com/password-statistics/>

² <https://www.schneier.com/essays/archives/2016/09/stop-trying-to-fix-t.html>

³ Armchair discussion on October 28, 2021, at a Cybersecurity Awareness Month event organized by the Center for Strategic & International Studies with Deputy National Security Advisor for Cyber and Emerging Technology in the Biden Administration Anne Neuberger and National Cyber Director Chris Inglis.

⁴ <https://www.csis.org/events/conversation-chris-inglis-and-anne-neuberger>

which branch is going to put in, or similarly with the anti-brake locking system – or, the antilock braking system, and so on and so forth. We have in other kind of forms of technology routinized and automated the technology such that it then properly served the interest of the human being, including the delivery of resilience and robustness. We need to do the same thing in cyberspace.

The statement provided by Anne Neuberger endorsing the use of MFA because passwords are dead is a contradiction. 2FA/MFA is not replacing passwords, but instead is generally creating compound authentication schemes that use passwords as one of its authentication factors. This means that when using a 2FA/MFA solution, users will need to deal with passwords, with all the problems that it entails, and must additionally manage one or more other authentication factor. This is an added burden to the user and the implementers of the services.

Chris Inglis states that technology should support the human being, not confound it, so we can wonder if multifactor authentication is a viable solution. It confounds not just the user, but the implementer as well. Service providers point to the end-users when password-related breaches occur. They deflect the blame and state that the breach would not occur if the end-users would abide by the password policies. Now, they can also blame the user for not leveraging 2FA/MFA for their authentication when the option was made available. As 2FA/MFA solutions are at least as complex to use as passwords, it is not fair to continue to blame the user when service providers should be the ones to offer an easy and secure authentication protocol.

1.2 Server-Side Protection of Shared Secrets

In its publication “Digital Identity Guidelines” (NIST Special Publication 800-63B)⁵, NIST details how the passwords must be protected on the server side:

The verifier SHALL use approved encryption and an authenticated protected channel when requesting memorized secrets in order to provide resistance to eavesdropping and MitM attacks.

Verifiers SHALL store memorized secrets in a form that is resistant to offline attacks. Memorized secrets SHALL be salted and hashed using a suitable one-way key derivation function. Key derivation functions take a password, a salt, and a cost factor as inputs then generate a password hash. Their purpose is to make each password guessing trial by an attacker who has obtained a password hash file expensive and therefore the cost of a guessing attack high or prohibitive. Examples of suitable key derivation functions include Password-based Key Derivation Function 2 (PBKDF2) [SP 800-132] and Balloon [BALLOON]. A memory-hard function SHOULD be used because it increases the cost of an attack. The key derivation function SHALL use an approved one-way function such as Keyed Hash Message Authentication Code (HMAC) [FIPS 198-1], any approved hash function in SP 800-107, Secure Hash Algorithm 3 (SHA-3) [FIPS 202], CMAC [SP 800-38B] or Keccak Message Authentication Code (KMAC), Customizable SHAKE (cSHAKE), or ParallelHash [SP 800-185]. The chosen output length of the key derivation function SHOULD be the same as the length of the underlying one-way function output.

⁵ <https://pages.nist.gov/800-63-3/sp800-63b.html>

The salt SHALL be at least 32 bits in length and be chosen arbitrarily so as to minimize salt value collisions among stored hashes. Both the salt value and the resulting hash SHALL be stored for each subscriber using a memorized secret authenticator.

For PBKDF2, the cost factor is an iteration count: the more times the PBKDF2 function is iterated, the longer it takes to compute the password hash. Therefore, the iteration count SHOULD be as large as verification server performance will allow, typically at least 10,000 iterations.

In addition, verifiers SHOULD perform an additional iteration of a key derivation function using a salt value that is secret and known only to the verifier. This salt value, if used, SHALL be generated by an approved random bit generator [\[SP 800-90Ar1\]](#) and provide at least the minimum security strength specified in the latest revision of [SP 800-131A](#) (112 bits as of the date of this publication). The secret salt value SHALL be stored separately from the hashed memorized secrets (e.g., in a specialized device like a hardware security module). With this additional iteration, brute-force attacks on the hashed memorized secrets are impractical as long as the secret salt value remains secret.

There is good evidence that this is simply beyond the capabilities of most providers of online services. It is too complex to implement, and developers don't know how to approach building a compliant solution.

The security website "HaveBeenPwned"⁶ provides a window into the security breaches that websites suffer and the data that gets exposed. Looking at the "Recently Added Breaches", the first four entries are:

- IDC Games: exposed passwords stored as salted MD5⁷ hashes
- Ducks Unlimited: exposed passwords stored as unsalted MD5 hashes
- ActMobile: exposed password hashes (unspecified algorithm) but flagged as "unverified"
- CyberServe: exposed passwords stored in plain text

These four breaches make it clear that password scrambling practices are far away from NIST recommendations. It is arguable that a correlation exists between the lack of implementation of security controls and appearing in the list of breached websites. But that doesn't take away the fact that the security controls are complex to implement and beyond the reach of smaller providers, where those breaches are occurring and affecting users by the millions.

1.3 Analysis of Existing Solutions

This section will analyze currently available solutions for web authentication. The goal is to assess the current offering of solutions and their adoption by the industry. Solutions will be assessed based on their suitability for a decentralized, provider-independent authentication method.

⁶ <https://haveibeenpwned.com/>

⁷ The MD5 message-digest hashing algorithm has been declared broken and obsolete almost 10 years ago. Bruce Schneier declared the MD5 broken as early as 2005 (https://www.schneier.com/blog/archives/2005/08/the_md5_defense.html)

1.3.1 Kerberos⁸ and Lightweight Directory Access Protocol (LDAP)⁹

Both solutions offer interesting avenues in the realm of authentication and single sign-on solutions. The requirements for implementation and deployment make them good solutions for enterprise-managed infrastructures but they are not suited for a decentralized management of user accounts.

1.3.2 Security Assertion Markup Language (SAML)¹⁰

SAML brings several interesting concepts to the authentication solution. It is an open standard that enables single sign-on, through the concept of a federated identity. By leveraging identity providers (IdPs), the authenticated identities and their attributes can be used with just one set of login credentials. SAML can provide Identity as a service (IDaaS). An overview of the flow is presented in Figure 1.

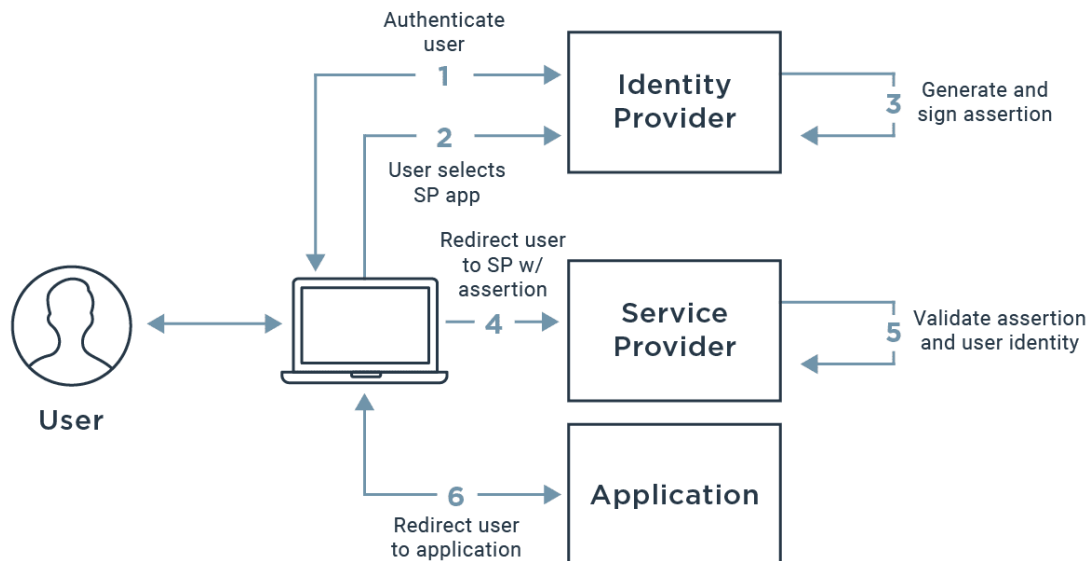


Figure 1: SAML Identity Provider SSO Flow (from <https://www.pingidentity.com/en/resources/client-library/articles/saml.html>)

Unquestionably, SAML offers a lot of great options but there are several drawbacks that prevent it from being used as a casual authentication mechanism for online service providers. It is interesting to consider Identity as a Service for formal identity verification, but for casual access to online services, it becomes difficult to manage what to reveal and what to keep secret when interacting with online providers. Users that wish to minimize the amount of identification points can be faced with a challenge if federated identities are to be used. On the side of implementers, the interaction with an Identity Provider increases the complexity and cost of operations.

⁸ <https://datatracker.ietf.org/doc/html/rfc4120>

⁹ <https://connect2id.com/products/ldapauth/auth-explained>

¹⁰ <https://www.pingidentity.com/en/resources/client-library/articles/saml.html>

Personally, I hope that the concept of federated identity becomes popular for official uses: in academia, the workforce, government services, and other similar uses. For more casual purposes, there needs to be an authentication scheme that allows the user to reveal as little as possible to the service provider. Here again, the solution is not suitable for authentication to decentralized web services.

1.3.3 WebAuthn + Client to Authenticator Protocol (CTAP) = Fast ID Online (FIDO)¹¹

The concept of WebAuthn and Fast ID Online is an up-and-coming framework that is very promising¹². It seeks to be an open standard that use asymmetric cryptography to authenticate users. The implementation is compatible with operating systems, browsers, authenticators¹³ and web servers.

By creating a framework where the client can exchange with the server to register a new account and provide a public key instead of a password, the protocol does indeed provide a flexible way to remotely authenticate without sharing secrets.

The protocol is trying to gain traction and might be on its way to become the main passwordless solution in years to come.¹⁴ There are two distinct flows to the process:

- Dialog between the online service and the browser (WebAuthn).
- Dialog between the browser and the authenticator (CTAP).

Although WebAuthn does not specifically require that the private keys be stored in a hardware authenticator, the specification was designed around that principle and allows authenticating entities to require that it be the case.

FIDO2 is likely the most innovative solution to online authentication issues. However, there are several drawbacks to the solution:

- Current solutions rely on the browser for the interaction with both the online service (WebAuthn API) and the authenticator (CTAP API). This places the browser in a centralized role and this is cause for concern:
 - The browser must be compatible with the standard and this can limit the deployment of the protocol (most browsers support the protocol)
 - The browser is an exposed component that directly interacts with many websites and has access to several sensitive components (namely session cookies, stored identities). A vulnerability in a browser could expose a very high number of clients to compromise.¹⁵
- In order to require that a hardware-based solution be used to store the private keys, the hardware device can use an attestation certificate that is signed by the vendor (Yubikey for one implements such an approach). This attestation can become a privacy-compromising tracking

¹¹ <https://webauthn.io/>

¹² <https://www.csoonline.com/article/3273009/will-webauthn-replace-passwords-or-not.html>

¹³ As of writing, the only authenticators that I could find were demonstration projects, not ready for use.

¹⁴ <https://research.kudelskisecurity.com/2020/07/08/replacing-passwords-with-fido2-updated-slides-and-resources/>

¹⁵ <https://www.forbes.com/sites/gordonkelly/2021/10/30/google-chrome-hack-new-attack-zero-day-exploits-upgrade-chrome-now/>

tool.

- If the service providers require a hardware-based solution to keep track of the private keys, then there are currently two solutions available: Yubikeys from Yubico and the Titan Key from Google.
- Hardware security keys are limited in the number of private keys that they can store (Yubikey allows for 25 different private keys to be stored).
- Some implementations require using a specific browser (Google Chrome), support in the operating system (Android, iOS, Windows) and a hardware token for private key storage.

If the solution is to gain traction, it requires tech giants to endorse it and adopt it on their platforms. When considering the adoption of the authentication protocol, users will assess the most universal implementation, i.e., the one that will work with everything they use. It would be disappointing that an authentication solution that was designed as an open standard would become a solution that is basically a Google product. There are already indications that Google is attempting to overtake the protocol: they have deployed their own hardware token (Titan Key bundle) and incorporated the protocol into Google Chrome and Android. Some uses of the protocol will require Google products. For example, if one wants to use a Titan Key to authenticate with Google or Twitter accounts, Google Chrome browser is required.¹⁶ This overtaking of protocols and open standards is not rare from tech giants of course. They are in a privileged position to impose their own implementation of a standard¹⁷ and gradually push users towards their own chain of products.

There are privacy implications to the solution as well. As already mentioned, the hardware device can become a tracking vector through the requirement of an attestation of hardware-based private key storage. Google has developed a reputation as a data-siphoning organization, and the company is trying to fix their image in that regard with limited success.¹⁸ ¹⁹ This should concern users that would see Google Chrome imposed in an authentication protocol. If the solution is using a smartphone as an authenticator solution, there are again concerns of vulnerabilities and privacy issues including sharing phone numbers with online providers as an authentication requirement. Currently, Twitter is encouraging (though not requiring) that users associate a phone number to their identity.

Undeniably, many vendors and researchers are actively developing technical solutions to address the weaknesses of current authentication solutions. As discussed in this section, some of these solutions are very promising but also present significant drawbacks.

¹⁶ <https://www.pcmag.com/reviews/google-titan-security-key-bundle>

¹⁷ <https://www.inlinepolicy.com/blog/the-geopolitics-of-standards-setting>

¹⁸ <https://www.forbes.com/sites/gordonkelly/2021/12/05/google-chrome-upgrade-warning-microsoft-warns-against-chrome-upgrade/>

¹⁹ <https://www.washingtonpost.com/technology/2021/09/23/google-privacy-settings/>

2. Designing the Solution

2.1 Solution Statement

In the last few months, I have challenged myself to come up with a solution to the problem of password-based user authentication when using online services. I would only consider the challenge successful if the solution met the following criteria:

- Strong enough to be used as a single factor of authentication, yet compatible with multifactor authentication
- Easier and faster to use than passwords
- No reliance on user memory or technical knowledge for the use of the system
- Platform-independent solution that does not require vendor-specific technology
- Versatile and transferable authentication scheme that is not attached to a device
- No privacy implication when using the authentication scheme
- Not completely automated and requiring user intervention to authorize the authentication process
- Easier to implement on the server-side than password-based authentication
- Resistant to server-side insider threat
- Resistant to server-side network intrusions and data breaches

In my effort to design the best possible solution to replace passwords, I used inspiration from existing solutions, FIDO2 in particular. I tried to retain the best aspects of commonly used authentication schemes and factors to design my own solution.

2.2 Advantages of Passwords for Authentication

Passwords present several advantages that explain why they are so entrenched in IT solutions. They offer low cost, flexibility, platform-independence, suitability for remote use, and absence of privacy implications. However, the burden on users and implementers is high and leads to security issues. The cybersecurity community endorses the use of password managers as an efficient way to reduce the burden of generating and especially remembering passwords. This tool can mitigate the “password fatigue” that users exhibit from having to manage a high number of passwords. Of course, it implies that an app is needed for the management of passwords, but instead of reducing usability, the opposite typically happens: the process is faster and easier than manually writing passwords. Security advisors get nervous at the thought of users storing several passwords in one location, the implication being that an endpoint compromise would disclose the entire set of passwords. However, experience indicates that both user experience and security increases with the adoption of a password manager. In the document “Digital Identity Guidelines” (NIST Special Publication 800-63B)²⁰, the authors state:

Verifiers SHOULD permit claimants to use “paste” functionality when entering a memorized secret. This facilitates the use of password managers, which are widely used and in many cases increase the likelihood that users will choose stronger memorized secrets.

²⁰ <https://pages.nist.gov/800-63-3/sp800-63b.html>

So, in my choice of a solution, I concluded that the use of a user-friendly application that allows for convenience and stronger security is desirable. However, in contrast to several popular solutions, I did not opt for a cloud-deployed solution but instead one that uses locally stored and encrypted vault files.

2.3 Reducing User Burden with App Solution

Password managers enhance the security and usability of passwords, exactly in line with the objectives pursued. However, this only applies to the user side: there is no impact for the implementers of the solution when password managers are thrown in the mix. The password still needs to be sent over to the server for validation.

As the basis for my solution, I opted for what NIST refers to as a “Cryptographic Software Authenticator”:

A single-factor software cryptographic authenticator is a cryptographic key stored on disk or some other "soft" media. Authentication is accomplished by proving possession and control of the key. The authenticator output is highly dependent on the specific cryptographic protocol, but it is generally some type of signed message. The single-factor software cryptographic authenticator is something you have.

Single-factor software cryptographic authenticators encapsulate one or more secret keys unique to the authenticator. The key SHALL be stored in suitably secure storage available to the authenticator application (e.g., keychain storage, TPM, or TEE if available). The key SHALL be strongly protected against unauthorized disclosure by the use of access controls that limit access to the key to only those software components on the device requiring access. Single-factor cryptographic software authenticators SHOULD discourage and SHALL NOT facilitate the cloning of the secret key onto multiple devices.

This solves two major problems of password-based authentication: the private key stays with the client, and the server-side does not need to worry about protecting a shared secret. The public key can be stored in clear text without concern. The server generates an authentication challenge that includes a number-used-once (nonce) and sends it to the client. If the client can encrypt the challenge with the corresponding private key and return this encryption to the server, the server can validate the signature and thus be convinced that the user possesses the associated private key. As stated by NIST, this represents “something you have” and not “something you know”. The user does not “know” the private key but has access to it. The authenticator software will digitally sign the challenge on the user’s behalf using the private key. The authenticator software is integral to the authentication scheme. The server-side management is reduced to a minimum and never needs access to the sensitive authentication credentials.

2.4 Technical Context of the Solution

For the scope of this project, there are two different contexts being considered: the user side and the server side. Although special care was taken to ensure that the solution can be implemented on any platform, this demonstration uses the following settings:

Client-side:

- Custom-made authenticator software with a graphical user interface developed in C#
- Application developed for desktop use in Windows 10
- Private keys stored in AES-encrypted files or hardware tokens on the client side

Server-side:

- Custom-made web server application developed on a Linux stack using Apache2 web server, PHP7 scripting, MySQL database, and additional tools coded in Python and Java
- User management and public keys stored in a relational MySQL database
- Virtual machine implementation using VMWare to simulate a remote web server

The rest of the report will detail the implementation of the software authenticator and the website implementation.

3. Implementation of the Solution

3.1 Authentication Protocol

To maximize the efficiency of the authentication solution, a custom protocol was designed and added at the application layer to allow client-server interaction for all tasks related to identity and access management. The protocol was designed to be flexible in the requested and provided parameters. For instance, a web server could be using the protocol to set a two-factor authentication scheme in one operation. Without a custom protocol, the user needs to manually interact with HTML forms to provide authentication credentials. Some password managers attempt to facilitate this operation using auto-type and copy-paste actions, as the web form will allow. With a dedicated and flexible protocol, this is made easy for the user. The protocol is enabled by XML-encoded messages delivered through HTTPS.

There are seven interactions recognized by the protocol:

1. User registration: creating an account on the web server
2. Signing into the account: establishing an authenticated user session
3. Resetting the key to an account: recovery mechanism for lost or compromised key
4. Renewing a private key: modifying the algorithm and/or key from an authenticated session
5. Applying a digital signature to an operation: continuous mediation of risk for privileged operations
6. Modifying the user profile on the web server: editing user information (except the key)
7. Account termination: removing the client identity from the server side and the client side

The interactions all represent specific outcomes on either the client side or the server side: the account will be created or modified, the key will be modified, the account will be deactivated or deleted, or a new authenticated session will begin for the client. During an authenticated session, the server may require a validation of credentials before performing a sensitive operation. The operation is generic from the point of view of the client. The server manages which operations require a revalidation of the user session. Examples include an online purchase, administrative activities, and requests for services.

All the interactions follow the same format: the client triggers a request through the web interface, and the server issues an XML-encoded message intended for the authenticator. The authenticator analyzes the request from the server and determines which parameters need to be provided for the response. The response is crafted by the authenticator and can be brought to the web interface to complete the interaction. For example, the client can access the sign-in page of the server and ask to begin a session. The server will generate a message that contains the challenge and will ask that a response be submitted that contains the username and the digital signature of the challenge by the corresponding private key. An example of such a message is displayed in Figure 2. The example is about a user registration request triggered by the user. The server uses the protocol to define mandatory and optional parameters, data types, default values, and more. The message states which digital signature algorithms are supported by the server. The client, through the use of the authenticator, can choose which algorithm to use.

```

<?xml version="1.0" encoding="UTF-8"?>
<easy_authentication_message action="register">
  <provided_parameters>
    <parameter name="identification" type="string" value="Server Name" />
    <parameter name="avatar" type="base64_image" value="iVBORw0K..." />
    <parameter name="url" value="https://domain.tld" />
  </provided_parameters>
  <requested_parameters>
    <parameter name="avatar" type="base64_image" mandatory="false" />
    <parameter name="public_key" type="string" mandatory="true" />
    <parameter name="first_name" type="string" mandatory="true" />
    <parameter name="last_name" mandatory="true" />
    <parameter name="email_address" type="email" mandatory="true" />
    <parameter name="address" mandatory="true" />
    <parameter name="phone_number" type="string" mandatory="false" />
    <parameter name="digital_signature_algorithm" type="closed_list"
      value="RSA-2048|RSA-3072|RSA-4096|ECDSA-NIST256p|
            ECDSA-NIST384p|ECDSA-NIST521p|DSA-1024|DSA-2048|
            DSA-3072"
      default="RSA-2048" mandatory="true" />
  </requested_parameters>
</easy_authentication_message>

```

Figure 2: XML-Encoded User Registration Message from the Server

The protocol is named “Easy Authentication”, and the XML-encoded messages back and forth use the root tag “easy_authentication_message”. As seen in the example, the message is simplistic. The action attribute (from the potential seven interactions) indicates what is the interaction in progress. There are two sections to the message: “provided_parameters” which are parameters provided by the web server to qualify the interaction. In Figure 2, the server states its identification, URL, and avatar for the client to use. The authenticator can add the information to the account being created and use it for verification purposes in subsequent interactions. This is done without the user having to provide any effort. The second section is named “requested_parameters” and qualifies what the user is expected to send back to successfully complete the interaction. Both sections detail the information through “parameter” tags.

In the authenticator software, the protocol is implemented by a package of classes as defined in Figure 3. The software defines a number of keyword parameters that hold a special meaning for the interaction. This is seen in the “Attribute” enumeration. These parameters are internally defined with default attributes, although those can be overridden in the exchanged messages. The seven interactions are defined in the enumeration “MessageAction”.

When using a challenge for a digital signature, the authenticator validates the challenge that was sent. It must follow a specific format that begins with the server identification, then a UTC-timestamp that specifies when the challenge was created, then an alphanumeric pseudo-random string. The challenge must be answered within 60 seconds of its creation, and the server identification of the challenge must match the server identification in the message parameter. The different outcomes of this validation are enumerated in “ChallengeStatus”.

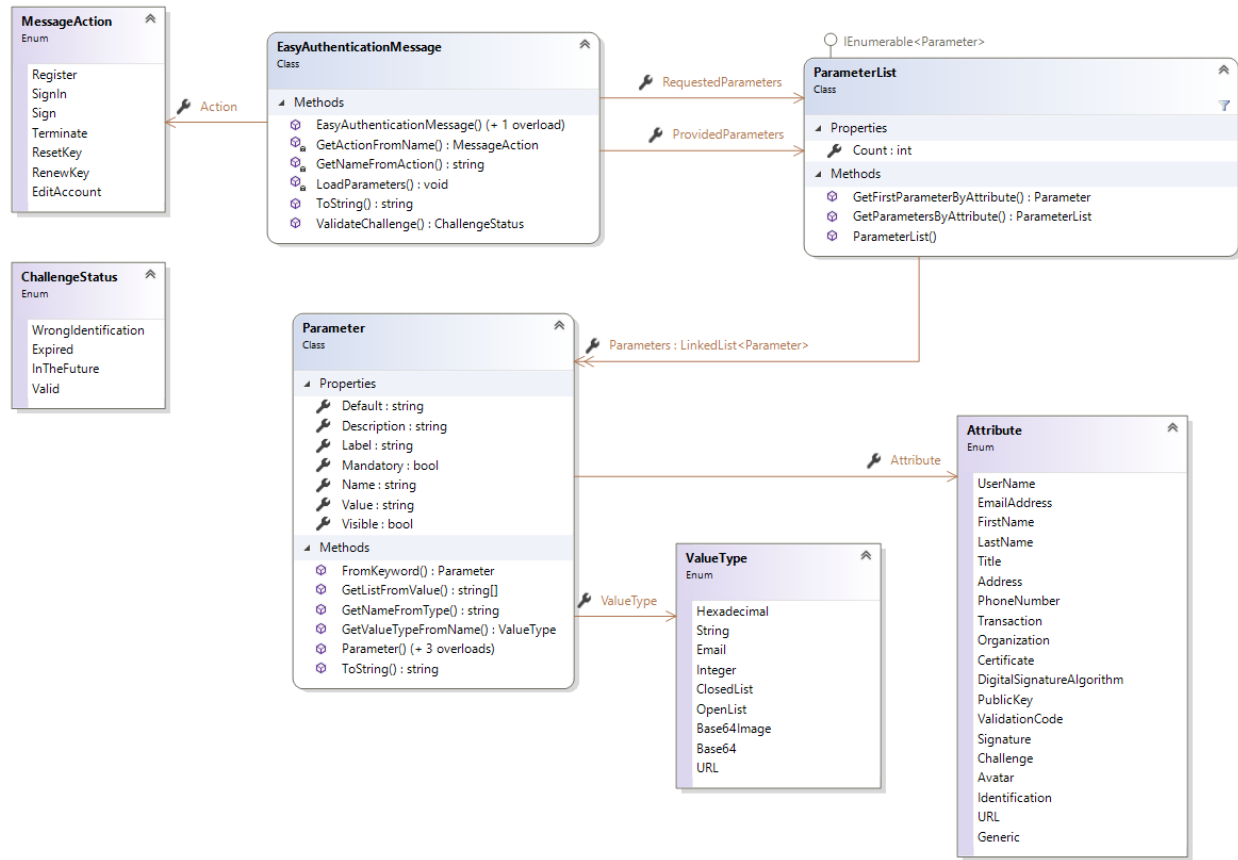


Figure 3: Class Diagram of the EasyAuthentication Protocol

3.2 Authenticator Tokens

For the desktop-based application in Windows 10, the interaction between the web interface and the authenticator is enabled by drag-and-drop operations. This was determined to be the easiest interaction. It separates the authenticator from the network and the browser since there is no automation. The browser is inconsequential and can be any modern browser. The authenticator is never exposed to the outside world directly. There is no impact on shared memory since the clipboard memory is not used.

The web server displays a graphical item in the web page that reacts to drag-and-drop operations. This graphical item is referred to as an authenticator token. The token is a vehicle for the XML-encoded EasyAuthentication messages. The client drags it from the web page and drops it in the authenticator interface. The message is extracted, analyzed, and the response crafted by the software. The authenticator embeds the response in a new authenticator token which the user drags to the web interface. Each interaction comes down to a simple back and forth drag-and-drop action from the user. An exception is the user registration, where the user registering the account must provide several identification points to the server. The authenticator simplifies the process by allowing the user to proactively define profiles in the app. Each profile defines the typical points of information that constitutes a user account on a web server. Since the protocol allows the server to request mandatory and optional points of information during the registration process, the authenticator can quickly

retrieve the information and populate the registration form. However, if the server requests non-typical information, such as an employee number, the user would have to complete the registration by providing that information manually.

For convivial identification of authenticator tokens, I used an easily recognizable look-and-feel that quickly identifies what the token is about.²¹

3.3 Digital Signatures

Of the seven interactions defined in the authentication protocol, only the account registration and the key reset do not require digital signatures. This is because in the former case the account does not yet exist for the operation, and in the latter, it is a recovery process used when the key is no longer available or trusted. Because the registration and key reset would be a very small fraction of the overall interactions in the typical lifecycle of a user account, almost all the interactions will use digital signatures.

To manage digital signatures in the authentication protocol, the authenticator software, and the server-side manipulation, a specific encoding of the information was used. The main items being exchanged and used are:

- Public keys: algorithm, implementation, and values
- Digital signatures: hashing algorithm and value

The exchange of information needs to be platform-independent and versatile. All the information required must be precisely encoded when exchanged between the entities involved. The authenticator software supports several industry-recognized digital signature implementations:

- RSA Digital Signature Algorithm using 2048-, 3072-, or 4096-bit keys
- Elliptic Curve Digital Signature Algorithm with standard curves NIST256p, NIST384p, and NIST521p
- The standard Digital Signature Algorithm based on ElGamal using 1024-, 2048-, or 3072-bit keys

In the protocol, this translates to the following digital signature algorithms:

- RSA
- ECDSA
- DSA

²¹ Details of the authenticator tokens as used in the demo applications are available in [Appendix 4](#).

The implementations are defined by specific names. The name starts with the digital signature algorithm, followed by the specific implementation. The supported implementations are:

- RSA-2048
- RSA-3072
- RSA-4096
- ECDSA-NIST256p
- ECDSA-NIST384p
- ECDSA-NIST521p
- DSA-1024
- DSA-2048
- DSA-3072

The names are standardized in the protocol; using a non-recognized digital signature implementation will cause the operation to fail. In the authenticator software, the digital signature is implemented by the package of classes detailed in Figure 4.

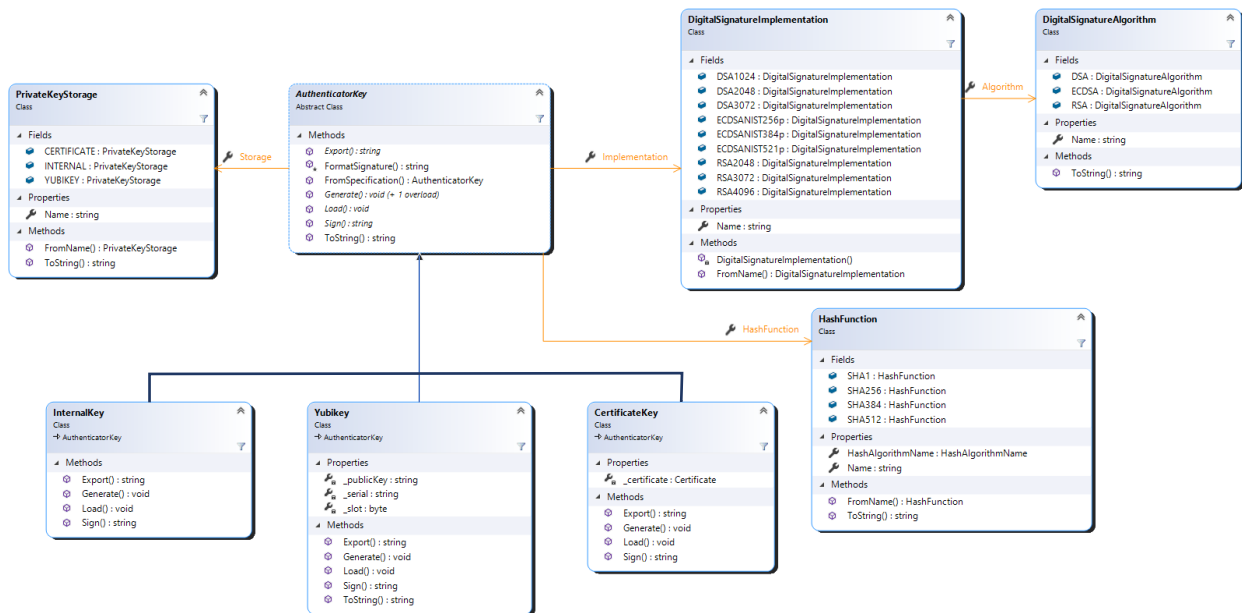


Figure 4: Class Diagram of Digital Signatures in the Authenticator

Central to the operation is the class “AuthenticatorKey”. It is an abstract class for which derived classes specify uses of private keys. For the authenticator, the main aspect of using digital signatures is how to manage the private key. It needs to be stored in a secure fashion. The class “PrivateKeyStorage” identifies the storage option of the private key. The key can be stored internally, which means that the authenticator generates and stores it in an encrypted local file. It can be stored within an X509 certificate (self-signed or CA-signed), in which case the authenticator will handle digital signature requests by having the certificate generate the signature when needed. The authenticator also supports the use of the cryptographic hardware token Yubikey. Private keys can be physically stored and protected in the Yubikey and the authenticator will interact with the hardware device to generate signatures when required. The three derived classes embody these options.

In addition to the private key storage options, the authenticator key will define which implementation is used. The class "DigitalSignatureImplementation", which itself has a "DigitalSignatureAlgorithm" attribute, defines this aspect of the digital signature. The class "HashFunction" provides an encapsulation of the hashing algorithms used by the various implementations.

When transferring a public key between the authenticator and a server, the formatting needs to be very specific. Different algorithms use different keys, but the need for standardization means that a specific encoding must be used. For the authentication protocol, a JavaScript-Object-Notation (JSON) format is used. The protocol will allow the software to define the implementation used, then to provide the key in a format that corresponds to the implementation. For RSA keys, the format is:

```
{"e": "<hexadecimal value of exponent>", "n": "<hexadecimal value of modulus>"}
```

For ECDSA keys, the public key is defined as:

```
{"publicKey": "<hexadecimal value of exponent>"}
```

And for DSA keys, the public key is defined as:

```
{"Y": "<hexadecimal value of Y>", "G": "<hexadecimal value of G>",  
"P": "<hexadecimal value of P>", "Q": "<hexadecimal value of Q>"}
```

When invoking the "Sign()" operation that is required in a derivation of an authenticator key, the instance will digitally sign the data provided using the private key and return the result in a standardized format. The format uses JSON and is defined as such:

```
{"signature": "<hexadecimal signature>", "hash_function": "<standard hash algorithm name>"}
```

This standardized signature can be embedded in the XML message and easily interpreted in any modern programming language.

3.4 Server-Side Identity and Access Management

The implementation of the authentication protocol on the server-side is straight-forward and requires few things:

- An information structure to manage the user identities, including their credentials
- The capability to verify digital signatures

In my demo server, I used a MySQL database to contain this information (Figure 5). The schema is simple yet provides a full management of the requirements for the solution. The public key is stored in a text field in the user_accounts table. Two tables, "digital_signature_implementations" and "digital_signature_algorithms" reflect their definitions in the authenticator software. The tables allow for a versatile management of supported algorithms, including their deprecation. The names and codes reflect the ones in the standard. In the user definition, a foreign key pointing to the implementation is all that is required for the web server to know which algorithm and implementation is used for the digital signature algorithm.

The “validation_codes” are used for the operations not validated through digital signatures, i.e., the user registration and the key reset. Different approaches can be used by the web server to secure these operations. A standard approach, and the one used in the demo server, is to use a trusted email account to send over the validation code. The operation is then validated by verifying that the user does have access to the email account and can produce the expected validation code. A more rigorous implementation might require the user to reach out to a technical support service for account activation/reactivation.

To provide a standardized validation of digital signatures, a centralized Python script located outside of the web server root will receive the digital signature and the public key and validate if the signature is valid or not. The entire script is available for consultation in 9. Appendix B – Digital Signature Validation Server-Side.

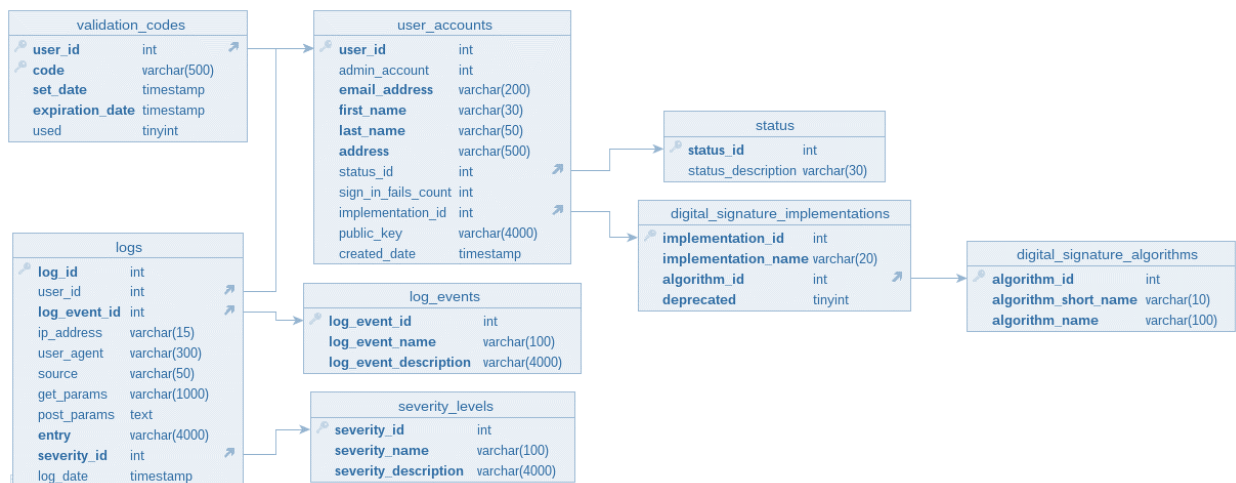


Figure 5: Database Schema for User-Related Information

The database schema contains three tables that support logging of activities at the application level. The logs contain the GET and POST parameters, along with typical log details.

To illustrate a basic interaction between the user and the web server, a code sample for a client authentication is presented in Figure 6. In the example, a user has requested an authenticated session, by providing a username (email address in this case) and the digitally signed encoded challenge from the server. The server starts with trivial validations:

- All mandatory parameters are provided and contain data
- The account exists and is active (not yet-to-be activated, locked, or any other invalid status)

After this initial validation, the code will then extract data from the database. All the information regarding that user is retrieved. The server then encodes the signature for verification by the Python script. The script needs to know what the challenge was, the digital signature algorithm used, the public key, the signature, and the hashing algorithm. If the script returns anything but the label “VALID”, the signature will be rejected. The events are logged, including counting failed attempts to authenticate if applicable.

When the authentication is successful, the server code retains the points of information as session variables. This is to avoid costly database queries and to facilitate the rest of the user session management.

The code sample begins after the initial validation and after the user data was extracted from the database. The database values are in an associative array called "row".

The logging captures the GET and POST parameters, meaning that all the information submitted will appear in the log records. If the user was submitting a secret, such as a password, it would be captured in plain text and put in a log. The use of TLS protects information in transit, but it must be decrypted when reaching the server where it becomes very vulnerable.²² In this implementation there is no sensitive secret, and the parameters can be captured and stored without concern when it comes to authentication.

```
$public_key = $row["public_key"];
$algo_name = $row["algorithm_short_name"];

$digital_signature_algorithm = '{"algorithm": "' . $row["algorithm_short_name"] .
    "', "implementation": "' . $row["implementation_name"] . "'}';

$command = "python3 /opt/util/check_signature.py '" . $digital_signature_algorithm .
    "' '" . $public_key . "' '" . $signature . "' '" . $_SESSION['message'] . "' ";

$check = shell_exec($command);
$check = trim($check);
if (trim($check) !== "VALID") {
    print("The authentication challenge failed: " . $check);
    add_log($conn, $user_id, 4,
        'Challenge authentication failed at the validation of the signature', 2);
    $valid = false;
}
else {
    // Successfully validated the user.
    $_SESSION['email_address'] = $row['email_address'];
    $_SESSION['first_name'] = $row['first_name'];
    $_SESSION['last_name'] = $row['last_name'];
    $_SESSION['address'] = $row['address'];
    $_SESSION['phone_number'] = $row['phone_number'];
    $_SESSION['public_key'] = $row['public_key'];
    $_SESSION['implementation_name'] = $row['implementation_name'];
    $_SESSION['algorithm_name'] = $row['algorithm_short_name'];
    $_SESSION['avatar'] = $row['avatar'];
    $_SESSION['admin'] = ($row['admin_account'] == 1);
    $_SESSION['authenticated'] = true;
    $_SESSION['user_id'] = $row['user_id'];
    add_log($conn, $user_id, 2, 'User has successfully signed in to the system', 1);
    $conn->query("CALL reset_login_fails(" . $_SESSION['user_id'] . ")");

    header("Location: index.php");
}
```

Figure 6: Server-side User Authentication in PHP

The implementation of user authentication is compatible with X509 certificates signed by a recognized CA. The implementation server-side requires the web server to have access to the CA certificate for

²² As a junior developer eager to use logging mechanisms for maintenance and debugging, I inadvertently captured passwords submitted by users, for example when they were inserted into the wrong field by the user. It struck me then how easy it is to compromise passwords when one has access to web server code, as I had to be careful to not capture them by accident.

verification, and the client submits a certificate for enrollment. Details of the implementation are available for consultation in 10. Appendix C – Client Enrollment with CA-Signed Certificate.

3.5 Client-Side Protection of Private Keys

The authentication framework provides great advantages when it comes to securing the sensitive aspects of the authentication process. The private key stays on the client side and is only used to generate digital signatures. Only the non-sensitive public keys and digital signatures need to transit between the client and the server for authentication.

This leaves the complex question of how to protect the private keys on the client side. Several considerations went into the design of this protection:

- The solution must be flexible and offer a wide variety of options to the user.
- Users requiring very high security must be able to implement this in a reasonable way.
- Users requiring simplicity and usability above all must be able to use such an implementation.

This part of the implementation is client-side only and has no impact on the server side. Yet, it is not entirely implemented through the authenticator software because different options might call onto other components of a trusted encryption environment (TEE). The implementation of the solution meets the following requirements:

- User account information, including the public key and optionally the private key, is kept in an encrypted file referred to as an encrypted vault
- If the private key is not in the vault, then the vault will contain the information required to point to the private key in order to request a digital signature from it
- Vaults are encrypted using AES-256
- The AES encryption key is derived from one or more access controls which can include: passwords, PIN codes, challenge-response schemes, and file hashes
- The access controls can be compounded to create a multi-factor authentication scheme to generate the decryption key
- The vaults are regular files (although encrypted) and can be transferred from one device to another²³

3.6 Access Controls

An attractive solution, and one endorsed by NIST, is to leverage the Trusted Platform Module (TPM) of the device to store the encryption key of the vault file. The key is made available after authentication to the operating system. This approach is still possible if the device where the vault is stored is using storage encryption (enabled by the TPM). I opted to explore alternatives that are independent from the platform and offer greater versatility. The implementation of vaults, accounts, and access controls in the authenticator software is illustrated by the class diagram in Figure 7.

²³ This operational requirement is contrary to NIST advice if the private key is within the vault file instead of a hardware token. NIST guidelines (<https://pages.nist.gov/800-63-3/sp800-63b.html>) states that: *“Single-factor cryptographic software authenticators SHOULD discourage and SHALL NOT facilitate the cloning of the secret key onto multiple devices.”*

The abstract class "AccessControl" is used to encrypt and decrypt vault files. It exposes a method by which an access control can be added to the current one and compound their respective keys. To achieve this, the authenticator starts by instantiating an "Unprotected" access control, meaning it uses a zeroized encryption key. Then, additional access controls will compound the key with their own (using an XOR operation) to create the final key. For example:

```
Unprotected:    [0, 0, 0, 0, 0, 0, 0, ..., 0]
                XOR
Password:       [17, 238, 45, 81, 12, ..., 211]
                XOR
KeyFile:        [54, 11, 233, 146, 15, ..., 49]
                =
Compounded key: [39, 229, 196, 195, 3, ..., 226]
```

The "Password" access control allows the user to generate a key by hashing a provided password. The "KeyFile" access control allows the client to use a file as an encryption key. Any static file can be used for the operation. The file is hashed, and the resulting digest is used as the associated key. This access control was inspired by a similar one used with KeePass password manager²⁴.

In the exploration of versatile access control mechanisms, I used the Yubikey cryptographic hardware device in several ways. One of them is for static passwords, which is of course compatible with a password access control to protect vault files. Another way is with the challenge-response protocol where a response is generated from the submission of a challenge to the Yubikey. An access control is created by combining the possession of the Yubikey to the knowledge of the challenge.

A custom access control that was designed to be used with the authenticator software is a concept that I refer to as a USB Security Key. The corresponding access control is defined in the class "UsbSecurityKey".

The definition of a USB Security Key is a removable device that contains two specific files at the root. One is named "security_key_id.txt" and the other is named "key_file.txt". The first file provides an identification string to uniquely identify the key, and the other is a static file that acts as a KeyFile security control, i.e., the encryption key is derived from the hashcode of that file. The authenticator software provides all the required tools to create, manage, and use the USB Security Keys easily for encryption purposes.

²⁴ <https://keepass.info/help/base/keys.html>

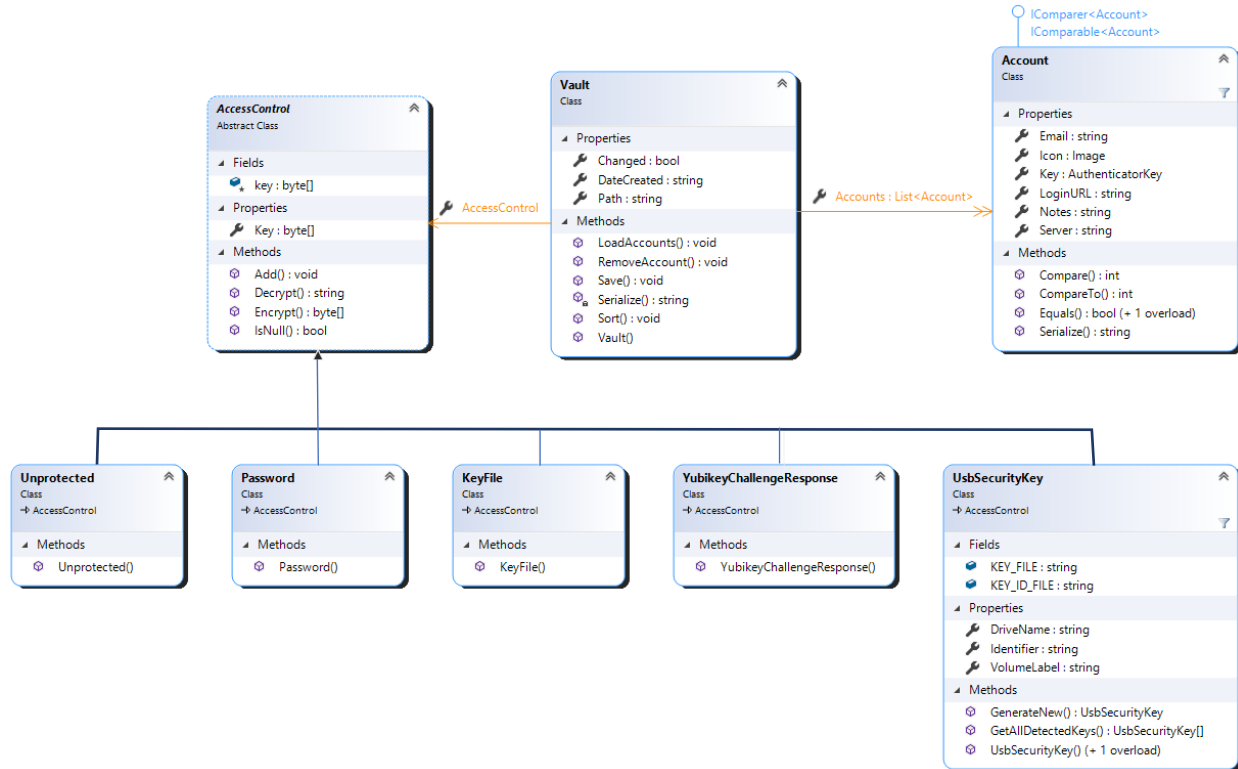


Figure 7: Class Diagram for the Protection of Private Keys

3.7 Vault

A vault is a file that contains user accounts. The authenticator software facilitates the interaction between the user and private keys. The authenticator software defines three options for private key storage:

- Internal storage: the private key is within the vault file, and protected by the access controls applied to it
- Certificate storage: the private key is embedded into a X509 certificate. When the user registers the account and provides access to the certificate, the authenticator software will store a copy of the certificate in the vault file, along with the password to decrypt it. The private key will be protected by the access controls of the vault file. The certificate file is no longer required after it has been imported into the vault file. The copy in the vault file will be sufficient.
- Yubikey storage: the private key is located within the physical Yubikey. The vault file will contain the serial number of the device and the slot number where the private key is located. The key cannot be extracted but the authenticator can ask the Yubikey to sign data. The Yubikey must be connected during the interaction.

3.8 Examples of Access Control Schemes

In order to showcase the versatility of the access controls that can be used to protect the private keys, consider the following scenarios providing different levels of security.

3.8.1 Password-protected Access

The client uses one vault protected by a main password. The password is a string of four random words, committed to memory. To unlock the vault, the user manually enters the password in the interface. The password is written on a piece of paper located in a locked cabinet for safekeeping and recovery.

3.8.2 TPM-Protected Access

The client keeps the vault in the Documents folder of his Windows 10 Operating System. There are no access controls applied to it. The hard drive is encrypted by BitLocker, using the TPM of the device. After authenticating to the system, the vault file is available for use.

3.8.3 Yubikey Static Password

The client owns a Yubikey configured to output a 50-character static password following a “long press” on the key. The password is the main password to the vault file containing the user’s private keys. The user keeps a copy of the password in a secure location.

3.8.4 USB Security Key

The user owns a USB Security Key containing a pseudo-random file used for the decryption of a vault. The vault is stored in the Documents folder of the Operating System. After authentication, the user can plug in the USB key and decrypt the vault file. A copy of the key file is kept in a secure location.

3.8.5 Yubikey with Biometric Access²⁵



Figure 8: Yubikey C Bio (from <https://www.yubico.com/ca/product/yubikey-c-bio/>)

²⁵ I was very keen to experiment with this scenario in my development but unfortunately the device was sold out and is still not available for purchase as of writing.

The user decrypts his vault file with a six-digit PIN code. The file does not contain private keys: they are located on a Yubikey. To generate digital signatures, the authenticator software asks the Yubikey to sign the data. The key starts blinking and waits for the user to apply a fingerprint to the key reader. If the fingerprint is accepted, the key will sign the data. Since the keys cannot be extracted from the hardware device, loss or failure of the device will require all accounts to be reset.

3.8.6 Fingerprint-Protected USB Key²⁶



Figure 9: Fingerprint-Protected USB Key

The user keeps the vault in a USB key. The key is equipped with a biometric reader that requires an authorized fingerprint before access to the protected section of the key is granted. A password access control is also used. To authenticate to an online service, the user must insert the USB key, decrypt it with a fingerprint, and finally open the vault inside using a password. After a short period of inactivity, the access to the key is revoked. No more authentications can be accomplished until the vault is unlocked again.

²⁶ This is the key that I purchased and experimented with to bring a biometric aspect to client-side key protection.

4. Assessment of the solution

The solution designed is an authentication framework which includes:

- The authentication protocol
- The authenticator software
- The server implementation
- The client-side protection of private keys

In this section, I will assess how well this authentication framework meets the operational and security requirements that were set out. As a reminder, these requirements were:

- Strong enough to be used as a single factor of authentication, yet compatible with multifactor authentication.
- Easier and faster to use than passwords.
- No reliance on user memory or technical knowledge for the use of the system.
- Platform-independent solution that does not require vendor-specific technology.
- Versatile and transferable authentication scheme that is not attached to a device.
- No privacy implication when using the authentication scheme.
- Not completely automated and requiring user intervention to authorize the authentication process.
- Easier to implement on the server-side than password-based authentication.
- Resistant to server-side insider threat.
- Resistant to server-side network intrusions and data breaches.

I will proceed to analyze each of the requirements.

Claim: strong enough for single factor use, yet compatible with multifactor authentication

The solution is indeed compatible with multifactor authentication, however that is a simple requirement to meet. Arguably, any single-factor authentication scheme could be supplemented by a second factor for added robustness.

The claim that it is strong enough to be used as a single factor is not as easy to determine. Based on the detailed risk analysis provided in 8. Appendix A – Risk Analysis, the solution used as a single factor of authentication is robust enough for most uses.

Claim: easier and faster to use than passwords

The demo implementation and subsequent use over several months has easily convinced me that yes, this scheme is easier and faster to use than passwords. The simple drag-and-drop process is very quick, and all the work is done by the authenticator.

There are basically two ways to make passwords easy to use: make them short and easy to remember or use a password manager. Of course, the first approach is not recommended, but the second one increases both the usability and security. The authentication framework is inspired by password managers but surpasses them in usability. I have spoken with many users of password managers, and there is a consensus: they speed up the process of using passwords. The authenticator software provides the same ease-of-use, and even more because the authentication protocol helps with the

server interaction. The dialog between the server and the client is facilitated and standardized by the authentication protocol, which regular password managers do not use.

Claim: no reliance on user memory or technical knowledge for the use of the system

There is no reliance on user memory, nor could there be since no one could be expected to remember asymmetric encryption keys. Users can resort to memory to protect keys on the client side, but it is not a requirement. There is no need to understand the intricacies of asymmetric encryption and digital signatures to use the drag-and-drop feature and log into websites. Anybody able to use passwords should be able to use this system. The system is using private keys as a "something you have" type of authentication scheme, and the keys can be remotely used with exposing them.

Claim: the solution is platform-independent and does not require vendor-specific technology

I did not prove this in my implementation. The solution as designed works for a Windows 10 desktop and the server side is implemented in Linux. However, the solution is designed for platform independence and could be ported to other systems. Nothing that was used in the framework is specific to a vendor or platform.

The implementation of the authenticator to a mobile device would require a change in the user interface design, since a drag-and-drop operation is easy in a desktop environment, but not in a mobile one. The graphical user interface could be developed in other languages.

Claim: versatile and transferable authentication scheme that is not attached to a device

Despite NIST recommendations, the authentication scheme was made to be cloneable and portable, prioritizing usability over security on this aspect. However, it is not a requirement for the solution. The authenticator software could follow NIST recommendations and not facilitate it. Using a TPM solution that holds the encryption key could render the solution non-transferable. My opinion is that it is a legitimate operational requirement to be able to transfer the private authentication credentials to other devices. Otherwise, users could find themselves in a situation where each authentication needs to be individually reset, or the solution must use a centralized solution to share the private keys to multiple accounts. I prefer a solution where I don't need to trust a third-party with my authentication credentials.

Claim: no privacy implication when using the authentication scheme

Solutions that require biometric information to be stored server-side raise concerns from a privacy perspective. The adoption of biometrics as a form of authentication is gaining in adoption on user-owned devices: smartphones, laptops, and desktops. Organizations use biometrics in a controlled manner as well. Biometrics are not easily compatible with a remote authentication to a web server. An example of disclosure of biometric information that had security repercussions is the breach of the Office of Personnel Management.²⁷ The impact of a breach of biometric information is difficult to

²⁷ <https://www.csoonline.com/article/3318238/the-opm-hack-explained-bad-security-practices-meet-chinas-captain-america.html>

assess, and the security industry is understandably reluctant to endorse widespread use of this type of data. The pushback from privacy advocates on the use of facial recognition is another such example.²⁸

Several 2FA schemes leverage the cellular network to provide an additional authentication factor, either through a PIN code or a cell network-based authentication. Even the use of phone numbers for authentication is enough to raise privacy concerns. Smartphone users avoid sharing this information when possible. It is already concerning for email addresses, but at least in this case users can resort to obfuscation, throwaway emails, and similar solutions. This is acceptable for use with trusted organizations at a reduced scale, but not suitable for wide distribution

The authenticator solution designed does not require sharing biometric information or phone numbers. Digital signatures have no privacy implications for the user to share them. Private keys are easily cancelled and replaced. They can also be unique per identification, meaning that a breached private key does not impact other authentication credentials and mitigates using public keys for user tracking.

Claim: requires user intervention to authorize the authentication process

Using a trigger on the client side instead of a fully automated process can mitigate certain vulnerabilities of authentication solutions. Fully automated authentication is convenient and fast, but users should be weary of the risks implied. Forcing a user trigger is a simple security measure but still a useful one. The authenticator requires the user to acknowledge the action and specifically trigger it through the drag-and-drop operation.

Claim: easier to implement on the server side than password-based authentication

Objectively so, especially considering the guidelines set out by NIST. This framework does not require that hashed passwords be stored in one location and salts in another. Neither does it require using at least 10,000 iterations of hashing so that the password can be adequately protected. The public keys do not require protection other than the regular management of non-sensitive production data.

Claim: resistant to server-side insider threats

Yes, especially when compared to a password-based solution. Even when the passwords are salted and hashed, they will be very vulnerable to interception on the server-side. It takes very little for an insider threat to get access to sensitive secrets being transited to the server, even with security controls in place. Simple injects in the server code can capture transmitted passwords following their obligatory decryption.

Claim: resistant to server-side network intrusions and data breaches

Yes, since the solution does not require any sensitive data to be stored on the server side. However, this only applies to the authentication mechanism. Network intrusion could expose other sensitive data being kept on the server side, such as financial credentials.

²⁸ <https://www.nbcnews.com/tech/security/2-democratic-senators-propose-ban-use-facial-recognition-federal-law-n1232128>

5. Security Analysis

A complete security analysis of the authentication framework was done using the Mitre Att&ck Knowledge Base. Detailed results are available in 8. Appendix A – Risk Analysis. The analysis highlighted several considerations to protect the deployment and implementation of the framework, and also how to address additional security considerations for security when using the protocols and tools.

5.1 Risk Analysis Summary

Security analysis led to the following observations regarding the authentication framework:

- System endpoints become targets for threat actors when the authentication is stronger:
 - The private key vault
 - The authenticator software
 - The online server
- Even with the mitigation of commonly exploited vulnerabilities with regard to authentication like phishing and social engineering, weaknesses to underlying protocols can still be exploited successfully

No authentication solution is full proof: more robust solutions are desirable because they will require additional resources and skills to exploit. Threat actors will deploy these resources based on the exploitation value of the assets. Increasing the cost of exploitation reduces the benefits from breaches.

The security analysis of the framework brings forth the following recommendations to increase the security.

5.2 Client-Side Security Recommendations

5.2.1 For Authenticator Use

1. Procure the authenticator software from verified sources
2. Validate authenticator software with developer digital signatures
3. Only apply patches and updates that are digitally signed
4. Protect authentication through a second factor when available

5.2.2 For Privacy and Confidentiality

1. Generate and use disposable email addresses and usernames to limit exposure of sensitive data
2. Do not reveal personal information to online providers beyond necessary and privilege disposable data when possible
3. Use browsers that meet your requirements for privacy and security

5.2.3 For Endpoint Resilience

1. Authenticate the server through Transport Layer Security (TLS)
2. Do not communicate with a server through links delivered in emails and text messages
3. Verify account activity on a regular basis to detect unauthorized activity
4. Use a fully patched and trusted browser for online activities
5. Explicitly terminate authenticated sessions when completed

5.2.4 For Private Key Protection

1. Protect vault access with strong multifactor authentication
2. Use hardware tokens to protect private keys
3. Do not duplicate or disseminate vaults beyond what is absolutely necessary

5.3 Server-Side Security Recommendations

5.3.1 For Account Management

1. Keep the collection of identification points to a minimum
2. Perform client enrollment through signed certificates when possible
3. Use official and dedicated email addresses for communications with clients
4. Use a robust mechanism for account validation and account reset, in accordance with the sensitivity of the assets being protected
5. Whenever possible, restrict access to the authentication portal by using supplemental authentication factors such as geolocation
6. Be careful to not reduce the security level of the authentication when engaging in account reset activities

5.3.2 For Account Monitoring

1. Engage in active monitoring of user account, using automation and analytics to highlight newly created accounts, and modifications to accounts or privileges
2. Provide visibility to authenticated users of activity with their accounts

5.3.3 For Protection of Authenticated Sessions

1. Perform continuous authentication of the user session through digital signatures for every sensitive operation
2. Do not rely on the browser for sensitive operations: use the browser only to receive and transmit non-sensitive information
3. Allow the user to explicitly and easily terminate an authenticated session
4. Automatically terminate sessions following a period of inactivity
5. Do not leak information regarding the existence of user accounts, including usernames and email addresses

5.3.4 For Endpoint Resilience

1. Protect sensitive assets on the server-side using authenticity controls: hashcodes, digital signatures, checksums, etc.
2. Use strong authentication between the components of the web application, e.g., between the web server and the database
3. Apply the concept of least privilege for the components of the web application

6. Limitations and Future Work

In this section, I will discuss limitations of the solution that I designed. This discussion will address several different vectors of the implementation.

6.1 Platform-independence and interoperability

The solution has been designed in order to be implementable on different platforms. A truly platform-independent solution would require an abstraction layer using dedicated APIs. This would facilitate implementation and deployment but also decouple the web application from the authentication procedure. In my demo web application, the activities accomplished by the user towards the account (registration, authentication, reset, removal) are integrated within the application. They have repercussions on the application data, they appear in application-specific logs, and so forth. Special work would be required to adapt the protocol as an abstracted API that could still integrate with the server-side application.

For the client-side, the biggest limitation is that the authenticator app is a Windows 10 desktop application. An obvious future endeavour would be to create a smartphone app that offer the same security and usability for a mobile device.

6.2 Security Issues

The blog KnowBe4 makes a good case about how authentication protocols are not now and will never be “hack proof”.²⁹ The dilemma of security versus usability is still present in the authentication framework that I designed, even though both security and usability are higher when compared to password-based authentication schemes.

The security analysis reveals vulnerabilities and security concerns about the protocol, namely:

- Adversary-in-the-middle attacks
- Social engineering
- Physical attacks

The designed solution offers versatility in its implementation, especially client-side, and end-users can customize a solution that meets their requirements. It is true that not all authentication is equal, and the sensitivity of the system and data being accessed should drive the decisions around the implementation. The WebAuthn protocol can impose requirements on the handling of the private key. Storing the private key in a Yubikey is a robust security solution, but one that has privacy implications and that does not offer versatility to switch devices.

Future work should evaluate how the protocol could evolve to address the concepts of private key storage requirements instead of leaving it to the whim of the end-user. An interesting avenue is to standardize the configuration of more than one public key to an account. This way, the users could register two different hardware devices for convenience and usability.

²⁹ <https://blog.knowbe4.com/yes-googles-security-key-is-hackable>

It was also noted in this paper that the implementation of the software authenticator is not compliant with NIST recommendation when it comes to facilitate the cloning of private keys. This is also a concept that needs to be addressed in future work.

6.3 Technical Skill Requirements

The concept of a password as an authentication scheme is straight-forward and easy to understand, even with very limited technical skills. An obvious advantage is that people understand it intuitively and do not need to be guided through an explanation of how it works. Users should be able to understand how authentication schemes are put together if they are going to trust their digital identities to them. Just knowing how to blindly apply the gestures to use the scheme can lead to security issues if the users cannot comprehend how the protocols work.

For an authentication framework to be adopted and widely used, people will need to understand it enough to feel comfortable using it. I don't know if users could be convinced of the efficiency of a "zero-knowledge proof" as an authentication platform.

6.4 Continuous Remediation

There are several security issues with web applications that are not dependent on the authentication protocols used. This includes Cross Site Scripting (XSS), Cross Site Request Forgery, and SQL Injection. A report from 2019 indicated that 30% of web applications were vulnerable to XSS.³⁰ My framework allows for the web server to request a digital signature when sensitive operations occur, and this helps addressing continuous remediation but presents some issues:

- It hurts the user experience (UX) by continuously asking the user to basically reauthenticate
- It relies on the implementers to determine which actions are "sensitive".

Even if the use of additional digital signatures can mitigate malicious activity, a complete security solution should restrict all unauthorized access to data and applications. I do not feel that the current framework allows for this without significantly downgrading the usability aspect.

Future work would need to explore how continuous remediation of each user request could be performed by the authenticator. Although I have given it significant thought, I do not have a proposal that would keep the intended spirit of the framework intact while offering this remediation. The solution might actually be to explore the nature of the HTTP protocol to prevent session hijacking at the source.

6.4 Endpoint Compromise

Current statistics indicate that authentication is the most exploited component of online activities by threat actors today. The assumption that fixing authentication will decrease the total number of breaches by how many are associated to authentication is erroneous. When the security increases for the authentication process, attackers will find other avenues to exploit online activities. Making their exploits more difficult to accomplish is certainly worthwhile. My point is that fixing authentication will

³⁰ <https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2019/>

move hacker efforts towards other avenues of exploitation, in a similar fashion as what has been observed for credit card fraud.³¹

One of the avenues that hackers would exploit if the authentication protocol becomes difficult to hack is endpoint compromise. It is arguable how easy it is to hack into smartphones and laptops, but malware is still definitely a concern. If exploitation efforts divert to the creation of end-point compromise malware, it is difficult to predict what the situation could become.

My framework is vulnerable to an endpoint compromise in several ways. If the private keys are internally stored in an encrypted file, a compromise could reveal them all at once. Flaws in the web application could leave session cookies vulnerable to theft.³² Future work would need to examine what could be done to protect the authentication credentials from device takeover, or at least mitigate the impact of such a breach. This may well be beyond what an authentication framework can offer and would be best addressed by browsers and operating systems.

7. Conclusion

Undertaking this challenge was by far the most enterprising project that I have tackled in my rather lengthy career. Designing the protocol and framework brought forward numerous additional challenges that I had not foreseen, more specifically the lack of standardization when trying to craft a solution that brought together technologies from various providers.

I am very satisfied with the work that was accomplished, especially in such a short timeframe. I believe that the demo implementation is very convincing and demonstrates that, while not a perfect solution, the authenticator and the accompanying protocol can be deployed efficiently to create a user-friendly and secure solution that does not sacrifice functionality.

³¹ <https://www.consumerismcommentary.com/chip-credit-cards-fraud/>

³² <https://rahuln.hashnode.dev/secure-website-against-cookies-theft-and-xss>

8. Appendix A – Risk Analysis

Consider the following assumptions before going into the security analysis:

1. The strength of the encryption algorithms will adequately resist decryption attempts.
2. Strong authentication does not compensate for lack of basic security controls.
3. Training and awareness for users and implementers is required to operate security mechanism correctly.

The first assumption implies that the algorithms have no known weaknesses to be exploited, and it can be reasonably stated that no threat actor will break the encryption. This cannot be proven of course and is not a static assertion in any case. Asymmetric encryption must evolve, and implementations get deprecated over time. The imminent arrival of quantum-enabled cryptanalysis will impact asymmetric encryption and the industry will need to adapt. The assumption is that the authentication solution will also evolve, and this assumption will remain true.

The second assumption is in consideration of the “weakest link” principle of security. Authentication, as it is used today, is an attractive target for attackers. Strengthening the authentication will force attackers to divert their attention to other vulnerabilities, but this is what stronger authentication should do. When authentication strength is enhanced and all other things remain equal, the overall security increases. Stronger authentication is not a reason to relax other security requirements.

The third assumption is related to the second. Training and awareness are a requirement for security implementation and operation. Easier and better solutions could lessen the need for training, but it should instead allow the training to focus on other more neglected aspects of security.

This section proposes a risk analysis of the authenticator solution. The solution includes the protocol, the client software, and the server-side implementation. The Mitre ATT&CK Framework³³ will be used for the security analysis. The goal is to identify the weaknesses of the authenticator by referring to the adversarial modelling tools of the framework.

Techniques for Reconnaissance

In the reconnaissance phase, threat actors will techniques to learn about their target. Not all these techniques are relevant for the authenticator solution. There are 10 techniques for reconnaissance.³⁴

Analysis

The authenticator solution does not require additional open ports for either the client or the server since it operates within HTTPS. The solution facilitates the exchange of identity information by automating it to a high degree. This creates a drive towards additional data being stored on the server-side: avatars (facial recognition potential), addresses, phone numbers, titles, etc. This accumulation of information is an attractive target for threat actors. A compromise of a web server could yield a lot of personal information.

³³ <https://attack.mitre.org/>

³⁴ <https://attack.mitre.org/tactics/TA0043/>

Recommendations

It may be unavoidable that personal data is stored on the server side, readily available to use for the management of online services. If compromised, this data could be part of a reconnaissance phase for future attacks on other services. Recommendations for the mitigation of reconnaissance are:

- Do not collect and store information that is not required: keep data collection to a minimum.
- Information that can be encrypted on the server-side can leverage the user's public key for encryption, limiting access to the owner of the corresponding private key.
- Avoid the use of private information in user definition whenever possible.
- Generate and use a random username in order to avoid the use of a non-official email for authentication purposes.

Techniques for Resource Development

In this phase, threat actors use the knowledge gained in the previous phase to stage and prepare the attack. There are 7 techniques for resource development.³⁵

Analysis

The more relevant techniques here are how the attacker will establish or compromise accounts to prepare for the eventual attack. Here, the account compromise is not the end goal of the attacker, but rather having access to accounts to use in the subsequent attack and intrusion.

Recommendations

A potential weak point in the authentication framework is the creation of a new user account.

Recommendations include:

- Whenever possible, use signed certificates for client enrollment.
- Whenever possible, use official email addresses for client enrollment.
- Validate newly created accounts in accordance with account sensitivity.

Techniques for Initial Access

With reconnaissance done, and the resources required for attack in place, the threat actor will proceed to initial access into the victim's infrastructure. There are 9 techniques for initial access.³⁶

Analysis

The authenticator software on the client-side is vulnerable to a drive-by compromise. The software is a critical point of failure in the overall authentication scheme. If the client system is compromised by custom malware, the private keys could be stolen when they are decrypted on the client side. The data stored in the vault clearly identifies which service is associated to which private key, the URL, the username, and other identification points.

³⁵ <https://attack.mitre.org/tactics/TA0042/>

³⁶ <https://attack.mitre.org/tactics/TA0001/>

The authentication framework mitigates some of the risk when compared to traditional implementations. Threat actors cannot compromise millions of users at once by breaching the web server. It could lead to disclosure of private information, but it would not automatically expose authentication credentials. Threat actors would access non-sensitive keys and could intercept digital signatures with little to no impact. A client-side breach requires time and effort and compromise one individual when the attack is successful. This is much more effort-intensive for the attacker.

As part of initial access techniques, attackers will often resort to social engineering techniques. Phishing, spearphishing and other variety of electronically conducted social engineering attacks will be used to obtain initial access to the infrastructure. The authenticator framework is not immune to social engineering but reduces the exposure when compared to other authentication techniques, especially passwords. The authenticator software does not easily let someone view their private keys and makes it virtually impossible in some implementations.

Supply chain compromise is a viable threat to the authentication framework and one that is very difficult to mitigate, as recent history indicates.³⁷ A scenario could involve a compromised update to the authenticator software, injecting malware in the most critical aspect of the authentication protocol.

The notion of using valid accounts to perform initial access is an interesting perspective. Compromises leveraged against the end-user to uncover authentication credentials, especially for privileged users, could indeed provide a high level of access to an attacker.

Recommendations

A strong authentication framework should provide robust mechanisms to prevent initial access of an adversary. The use of the authentication framework will mitigate several techniques used by attackers, especially if the following recommendations are followed:

- Provide proper authentication of the web server through TLS.
- Add mutual authentication in the framework. At client enrollment, the server could generate its own key pair specifically for that user account. This approach could greatly restrict the potential of an adversary-in-the-middle attack.
- The authenticator software and updates should be developer-signed for authenticity.
- Provide visibility to user account activities for early detection of malicious use.
- Use a hardware-based mechanism to protect the private keys, along with a user-triggered operation for digital signing which greatly limits the capacity of an attacker to take over the client-side authentication process.

³⁷ <https://www.extremenetworks.com/extreme-networks-blog/solarwinds-a-supply-chain-compromise/>

Techniques for Execution

In this stage of the attack, the intruder is attempting to execute malicious code in the victim's infrastructure. There are 12 techniques for execution.³⁸

Analysis

On the client-side, if the intruder manages an initial access and can run code, there might be little to prevent the takeover of authentication software. The usual restrictions on the client-side such as reduced privileges and code signing can contribute to securing the endpoint. As to the significance of the code execution on the client side, the impact could be a full compromise of the user's private keys. A hardware-based protection mechanism could significantly impede the capacity of an attacker to compromise the authentication. For example, if the private key is stored in a Yubikey that requires biometric authentication before signing data with the private key.

If the initial access was performed successfully on the server-side, then the authentication framework failed, and unauthorized activity can likely be conducted against the server.

Inter-process communication is a venue of compromise that is mitigated on the client-side. The communication of information is triggered by drag-and-drop operations that are not easily replicated by a remote attacker.

Recommendations

The following recommendations are issued to prevent code execution on the client side:

- Validate software through digital signatures when installing and updating.
- Perform continuous authentication of the user session through digital signatures for every sensitive operation.
- Require user-triggered activities for authentication (drag-and-drop, finger applied to Yubikey, etc.)
- Do not provide capabilities for the authenticator software to connect directly to network services: keep the operation within the client operating system.
- Do not rely on the browser for sensitive operations: use the browser only to receive and transmit non-sensitive information.

³⁸ <https://attack.mitre.org/tactics/TA0002/>

Techniques for Persistence

The adversary uses techniques for persistence to maintain a foothold in the victim's infrastructure. There are 19 techniques used by the adversary to establish persistence.³⁹

Analysis

Once the threat actor has sufficient access to the victim's system, persistence can be put in place. Account manipulation is an avenue, especially if the privileges are increased on the server side. The authentication process can be manipulated on the server side, in the web scripts or the digital signature verification script. Such manipulations could also hide the logs and make detection very difficult.

Recommendations

Here are recommendations to mitigate persistence of the adversary in regard to the use of authentication software:

- Use hardware-based security features to protect private keys on the client-side.
- Engage in active monitoring of user account activities on the server-side, including the use of automation and reporting to highlight newly created accounts, or modifications to privileges.
- Use a robust mechanism for account validation and account reset, in accordance with the sensitivity of the assets being protected.

Techniques for Privilege Escalation

In this phase of the attack, the threat actor uses already established access to discover and exploit additional vulnerabilities, weaknesses, and misconfigurations in order to increase the level of privilege. There are 13 techniques used by the adversary for privilege escalation.⁴⁰

Analysis

The authentication framework does not provide much protection against privilege escalation since it operates when authenticating users. The same techniques that grant persistence can be leveraged to circumvent security mechanisms and hijack other accounts. The use of digital signatures to confirm sensitive operations during the user session is a mitigation of privilege escalation. If an attacker were to hijack an administrator session through cross-site scripting for instance, it would not be able to use the account to create or modify another account if the application requires such a request to be specifically signed.

³⁹ <https://attack.mitre.org/tactics/TA0003/>

⁴⁰ <https://attack.mitre.org/tactics/TA0003/>

Recommendations

The following recommendations target the mitigation of techniques used for privilege escalation in regard to the use of the authentication framework:

- Protect sensitive assets on the server-side using authenticity mechanism: hashcodes, digital signatures, checksums, etc.
- Monitor for newly added accounts or modifications to existing accounts.

Techniques for Defense Evasion

In this phase, the adversary attempts to avoid detection to maintain the compromise. There are 40 techniques listed in the framework for defense evasion.⁴¹

Analysis

Several of the techniques for defense evasion have been analysed in previous sections. Avoiding detection means manipulating the accounts and/or the logs on the server side. If the attacker operates on the client-side to hijack an account, detection is going to follow quickly. But if the attacker is controlling certain aspects of the authentication framework on the server side, detection can be thwarted up to a point.

Recommendations

In the demo server, great care was taken to log user activities and make them transparent. Administrative accounts have easy access to user activities and can engage in active monitoring. Upon successful authentication, users are shown a history of their sessions, including failed attempts to authenticate. This forces the threat actor to subvert the server scripting to keep logs out of the system, or to gain access to the database to purge evidence of compromise. In the demo server, the access of the web server to the database for data modification is granted uniquely through stored procedures. Logs can only be added from the web server account: not edited nor removed.

In addition to application-specific logging, the web server will log activities, and the operating system level logging should also provide detection mechanisms. An adversary that attempts to subvert all of this needs access to server code, database accounts, and privileged operating system access.

The approach of weakening the encryption (technique 39) does not appear viable against the authentication framework.

Recommendations for this phase include:

- Use strong authentication to connect the web server scripts to the database.
- The web server access to the database should apply the principle of least privilege, namely for logging and account modification.
- Use mechanisms to detect the modification of source code in the database and web server.

⁴¹ <https://attack.mitre.org/tactics/TA0005/>

Techniques for Credential Access

In this phase, the attacker attempts to steal account names and passwords. There are 15 techniques for credential access.⁴²

Analysis

The authentication framework is particularly relevant when discussing credential access from the attacker's perspective. The following techniques are invalidated by the use of the authentication framework for obvious reasons:

- Brute Force
- Credentials from Password Stores
- Forced Authentication
- Input Capture
- Network Sniffing
- OS Credential Dumping
- Two-Factor Authentication Interception
- Unsecured Credentials

The adversary-in-the-middle might be the most credible threat to the authentication protocol when considering a resourceful adversary. The use of the authentication framework is also vulnerable to stolen Web session cookies since this happens after the initial user authentication and is a vulnerability of the web protocol more than anything else. A mitigation is offered through the digital signing of sensitive operations and the use of an authenticated and encrypted communication channel such as HTTPS.

Recommendations

Most protocols for web authentication suffer from the same caveat: even with strong authentication, a session hijack can invalidate the mechanism by letting the attacker impersonate the user through the theft/compromise of a session cookie. Recommendations to address this vulnerability are listed as such:

- Authenticated sessions should only occur over an encrypted channel using approved encryption algorithms.
- Add a second factor of authentication that can limit adversary-in-the-middle attacks.
- Reauthenticate sensitive operations using digital signatures throughout the duration of the session.
- Allow the user to explicitly terminate the session.
- Automatically terminate sessions following a period of inactivity.

⁴² <https://attack.mitre.org/tactics/TA0006/>

Techniques for Discovery

This attack phase is used by the adversary to gain knowledge about the system and internal network. There are 29 techniques for discovery.⁴³

Analysis

The techniques listed are not mitigated, but neither are they enhanced by the use of the authentication framework. Removing the use of passwords can make technique 15: Password Policy Discovery a moot point.

Recommendations

Here are recommendations to limit the efficiency of discovery techniques:

- The application must remove the visibility of existing user accounts to external users. For instance, an attacker should not be able to verify the existence of a user account easily and stealthily.
- When possible, the application should use official emails for client enrollment and other activities.
- For a publicly available server, the registration process should start with the validation of an email address. After the user has provided an email address, the rest of the process can resume from a message sent to that mailbox.

Techniques for Lateral Movement

The techniques used by an adversary to move through an environment. There are 9 techniques used for lateral movement.⁴⁴

Analysis

Although the techniques employed in this category are mostly beyond the authentication point, there are still some aspects that can be mitigated by the use of the authentication framework. Namely, replay attacks are mitigated by the framework. However, the session hijacking aspect through stealing a web session cookie after the authentication is still a viable threat. Digital signing of sensitive operations mitigates the impact of this technique.

⁴³ <https://attack.mitre.org/tactics/TA0007/>

⁴⁴ <https://attack.mitre.org/tactics/TA0008/>

Recommendations

Using a robust authentication mechanism that allows for a continuous remediation of threats would counter the lateral movement techniques of an intruder and offer greater resilience to attacks. A non-static authentication enhances the security, but still leaves some gaps, namely the vulnerability of web session cookies.

Recommendations to address lateral movement include:

- Use of a second factor of authentication.
- Use of continuous authentication throughout the user session for sensitive operations.

Techniques for Collection

The adversary is deploying techniques to gather data of interest. Data gathered brings the attacker closer to the end goal of the exploitation. There are 17 techniques used for collection.⁴⁵

Analysis

If the adversary is engaging in collection techniques, the authentication framework has failed. Of course, there are gaps and vulnerabilities that can be targeted for initial access other than the authentication. The goal of the authentication framework is to remove authentication from the list of commonly exploited vulnerabilities leveraged by attackers for their intrusion and compromise of the victim's infrastructure. Among the techniques listed, the relevant ones are adversary-in-the-middle and browser session hijacking.

Recommendations

The adversary-in-the-middle can be mitigated by the use of TLS, and browser session hijacking is addressed by keeping the browser at a distance from the authentication activities. The authentication framework relegates the browser to a non-critical role. However, it still comes into contact with some information that can be used in the discovery process by the threat actor. Keylogging is not a threat to the way that the software is used, but the browser will receive information pertaining to the enrollment and authentication activities. When dropping the authentication token, all that information become This information can likely be useful for the attacker, who could use it to engage in account reset activities or social engineering.

Recommendations to mitigate collection techniques are:

- Use a digitally signed authenticator software.
- Do not collect more information than what is absolutely required for user identification.

⁴⁵ <https://attack.mitre.org/tactics/TA0009/>

Techniques for Command and Control

These are techniques used when the attacker is within the infrastructure and looking to establish a communication channel from outside to control and propagate the internal compromise. There are 16 techniques used by the adversary to establish command and control.⁴⁶

Analysis

At this stage of the attack, the authentication framework is irrelevant and has been subverted. Unless the protocol for authentication provides an exploitable side-channel for command and control (and it doesn't) then the framework is not involved in facilitating or preventing the establishment of a command and control structure for the attacker.

Recommendations

Authentication is only part of the answer when it comes to security. The role of a robust authentication framework is exactly to keep the adversary from reaching the latter phases of a compromise. In this case, the framework is using https to provide authentication measures, so it does not contribute to provide avenues for a command-and-control channel that would otherwise not be there. It highlights a vulnerability of using a dedicated channel to establish the authentication that would be separate from the https protocol. Establishing such a channel could allow the use of a mutually authenticated, adversary-in-the-middle resistant authentication protocol. However, caution would need to be applied so that it does not become an exploitable side channel for command-and-control purposes.

There are no specific recommendations on the use of the authentication framework to mitigate command-and-control techniques.

Techniques for Exfiltration

In this phase of the attack, the adversary is exfiltration valuable data. Exfiltration could use some of the same techniques as the ones for command and control. It could also overlay on other communication pathways. There are 9 techniques for exfiltration.⁴⁷

Analysis

Web applications that have been compromised offer a viable avenue for data exfiltration. The use of encrypted channels helps protect privacy and confidentiality but also obfuscate malicious exfiltration.

Recommendations

For this phase of the attack as well, the authentication framework cannot offer mitigations. If the adversary is attempting data exfiltration, the authentication mechanisms have long been subverted.

⁴⁶ <https://attack.mitre.org/tactics/TA0011/>

⁴⁷ <https://attack.mitre.org/tactics/TA0010/>

There are no specific recommendations on the use of the authentication framework to mitigate exfiltration techniques.

Techniques for Impact

The adversary is attempting to impact, degrade, or destroy data and services following a successful compromise. This can be instead of, or in addition to data exfiltration. There are 13 techniques for impact.⁴⁸

Analysis

The authentication framework has some vulnerabilities when it comes to denial of service. An attacker could remove the access to an account by failing its authentication several times in a row. This would force the user in an account recovery process where certain vulnerabilities become available in the attack surface. A successful attacker could destroy accounts or manipulate them through the subversion of data storage.

Recommendations

Given that it would be next to impossible to breach an account through brute forcing, it is a reasonable question to ask if account locking is a worthwhile security measure? It is quite possible that account locking following several failed attempts would actually reduce the overall security by forcing account reset procedures. Why give the opportunity to an attacker to trigger such a procedure if it is not mitigating a credible threat?

Recommendations to address impact techniques:

- Actively monitor the activities occurring at the user account level.
- Whenever possible, restrict access to the authentication portal by using supplemental authentication such as geolocation factors.
- Do not reduce the security level of the authentication when engaging in account reset activities.

⁴⁸ <https://attack.mitre.org/tactics/TA0040/>

9. Appendix B – Digital Signature Validation Server-Side

9.1 Python Script to Validate RSA, ECDSA, or DSA Signatures

The script is centralized and can validate a signature for all supported implementations. To achieve this, the approach was standardized and uses JSON-encoded information. The user public key is already stored in this way in the database, so the other parts required are the signature (encoded to specify the signature and the algorithm used), the hashing function, and the message that was hashed and signed.

```
import ecdsa
import json
import sys

from Crypto.PublicKey import DSA
from Crypto.Signature import DSS
from Crypto.Hash import SHA1, SHA256, SHA384, SHA512
from hashlib import sha1, sha256, sha384, sha512
from ecdsa.keys import BadSignatureError

def main():
    if len(sys.argv) != 5:
        print_usage()
        exit()

    algorithm_json = sys.argv[1]
    public_key_json = sys.argv[2]
    signature_json = sys.argv[3]
    message = sys.argv[4]

    algorithm = json.loads(algorithm_json)['algorithm']
    description = json.loads(algorithm_json)['implementation']

    hex_signature = json.loads(signature_json)['signature']
    hashfunction = json.loads(signature_json)['hash_function']

    if algorithm == 'RSA':
        check_rsa_signature(public_key_json, hex_signature, message, hashfunction)
    elif algorithm == 'ECDSA':
        check_ecdsa_signature(public_key_json, hex_signature, message, hashfunction,
description)
    elif algorithm == 'DSA':
        check_dsa_signature(public_key_json, hex_signature, message, hashfunction)
    else:
        print("INVALID: unrecognized algorithm: " + algorithm)
```

```
def check_rsa_signature(public_key_json, hex_signature, message, hashfunction):  
    public_key = (int(json.loads(public_key_json)['n'], 16),  
int(json.loads(public_key_json)['e'], 16))  
    result = rsa_verify(bytes(message, 'utf-8'), hex_signature, public_key, hashfunc-  
tion)  
    if result:  
        print("VALID")  
    else:  
        print("INVALID")
```

```

def rsa_verify(message, signature, key, hashfunction):
    from struct import pack
    from sys import version_info

    if signature[0:2] == '0x':
        signature = signature[2:]

    def b(x):
        if version_info[0] == 2:
            return x
        else:
            return x.encode('latin1')
    assert(isinstance(message, type(b'')))

    block_size = 0
    n = key[0]
    while n:
        block_size += 1
        n >>= 8
    signature = pow(int(signature, 16), key[1], key[0])
    raw_bytes = []
    while signature:
        raw_bytes.insert(0, pack("B", signature & 0xFF))
        signature >>= 8
    signature = (block_size - len(raw_bytes)) * b('\x00') + b('').join(raw_bytes)

    if signature[0:2] != b('\x00\x01'):
        return False

    signature = signature[2:]
    if not b('\x00') in signature:
        return False
    signature = signature[signature.index(b('\x00'))+1:]
    if not signa-
tu-
re.startswith(b('\x30\x31\x30\x0D\x06\x09\x60\x86\x48\x01\x65\x03\x04\x02\x01\x05\x00\x04\x20')):
        return False
    signature = signature[19:]

    if hashfunction == "SHA1":
        digest = sha1(message).digest()
    elif hashfunction == "SHA256":
        digest = sha256(message).digest()
    elif hashfunction == "SHA384":
        digest = sha384(message).digest()
    elif hashfunction == "SHA512":
        digest = sha512(message).digest()
    else:
        print("Unrecognized hashing function: " + hashfunction)
        exit()

    if signature != digest:
        return False
    return True

```



```

def check_ecdsa_signature(public_key_json, hex_signature,
                          message, hashfunction, named_algorithm):
    public_key = json.loads(public_key_json)['publicKey']
    curve_definition = named_algorithm

    if public_key[0:2] == "0x":
        public_key = public_key[2:]

    if hex_signature[0:2] == "0x":
        hex_signature = hex_signature[2:]

    if hashfunction == "SHA1":
        hash_object = sha1
    elif hashfunction == "SHA256":
        hash_object = sha256
    elif hashfunction == "SHA384":
        hash_object = sha384
    elif hashfunction == "SHA512":
        hash_object = sha512
    else:
        print("Unrecognized hashing function: " + hashfunction)
        exit()

    if curve_definition == 'ECDSA-NIST256p':
        selected_curve = ecdsa.NIST256p
    elif curve_definition == 'ECDSA-NIST384p':
        selected_curve = ecdsa.NIST384p
    elif curve_definition == 'ECDSA-NIST521p':
        selected_curve = ecdsa.NIST521p
    elif curve_definition == 'ECDSA-BRAINPOOLP160r1':
        selected_curve = ecdsa.BRAINPOOLP160r1
    elif curve_definition == 'ECDSA-BRAINPOOLP192r1':
        selected_curve = ecdsa.BRAINPOOLP192r1
    elif curve_definition == 'ECDSA-BRAINPOOLP224r1':
        selected_curve = ecdsa.BRAINPOOLP224r1
    elif curve_definition == 'ECDSA-BRAINPOOLP256r1':
        selected_curve = ecdsa.BRAINPOOLP256r1
    elif curve_definition == 'ECDSA-BRAINPOOLP320r1':
        selected_curve = ecdsa.BRAINPOOLP320r1
    elif curve_definition == 'ECDSA-BRAINPOOLP384r1':
        selected_curve = ecdsa.BRAINPOOLP384r1
    elif curve_definition == 'ECDSA-BRAINPOOLP512r1':
        selected_curve = ecdsa.BRAINPOOLP512r1
    else:
        # print(-3) # Unsupported ECDSA curve.
        exit(-3)

    try:
        vk = ecdsa.VerifyingKey.from_string(bytes.fromhex(public_key),
                                           curve=selected_curve, hashfunc=hash_object)

        if vk.verify(bytes.fromhex(hex_signature), bytes(message, 'utf-8')):
            print("VALID")
            exit(0) # Successfully verified the signature.
        else:
            print("INVALID")
            exit(-1) # Verification was not successful.
    except BadSignatureError as e:
        print("BAD SIGNATURE")
        print(e)
        exit(-2)

```

```

def check_dsa_signature(public_key_json, hex_signature, message, hash_algo):
    key_params = json.loads(public_key_json)
    public_key = DSA.construct((
        int(key_params['Y'], 16),
        int(key_params['G'], 16),
        int(key_params['P'], 16),
        int(key_params['Q'], 16)
    ))

    if hash_algo == 'SHA1':
        hash_obj = SHA1.new(bytes(message, 'utf-8'))
    elif hash_algo == 'SHA256':
        hash_obj = SHA256.new(bytes(message, 'utf-8'))
    elif hash_algo == 'SHA384':
        hash_obj = SHA384.new(bytes(message, 'utf-8'))
    else:
        hash_obj = SHA512.new(bytes(message, 'utf-8'))

    if hex_signature[0:2] == '0x':
        hex_signature = hex_signature[2:]

    signature = bytes.fromhex(hex_signature)

    verifier = DSS.new(public_key, 'fips-186-3')
    # Verify the authenticity of the message
    try:
        verifier.verify(hash_obj, signature)
        print('VALID')
    except ValueError:
        print("INVALID")

def print_usage():
    print("python3 check_signature.py <json-encoded algorithm definition> <json-  

    encoded public key> <signature hex> <SHA1|SHA256|SHA384|SHA512> "<message>")

if __name__ == "__main__":
    main()

```

10. Appendix C – Client Enrollment with CA-Signed Certificate

10.1 Using CA-Signed Certificates

The authentication protocol is a versatile variant on the use of certificates. Providing a public key without a signed certificate is akin to using a self-signed certificate. The methodology is between PKI and PGP, where there is a trust in the public key received, but still a client-server relationship rather than a peer-to-peer one.

The authentication framework can function with CA-signed certificates with few changes. The web server needs access to the CA certificate (or at least public key) to validate certificate that are provided. There are dates of validity and expiration in the certificate. Subject details will also be embedded in the document. Therefore, user registration entails providing a valid certificate. The web server can extract all the details from the document and register the client.

The demo server that implemented certificates for user management requirement modifications to the database. A field was added to keep a copy of the document, and two fields added for the dates of validity. In my demo, I also added two tables to manage the organisational units and the user associations to those units. The information is carried by the certificate. The modified schematic of the database is displayed in Figure 10.

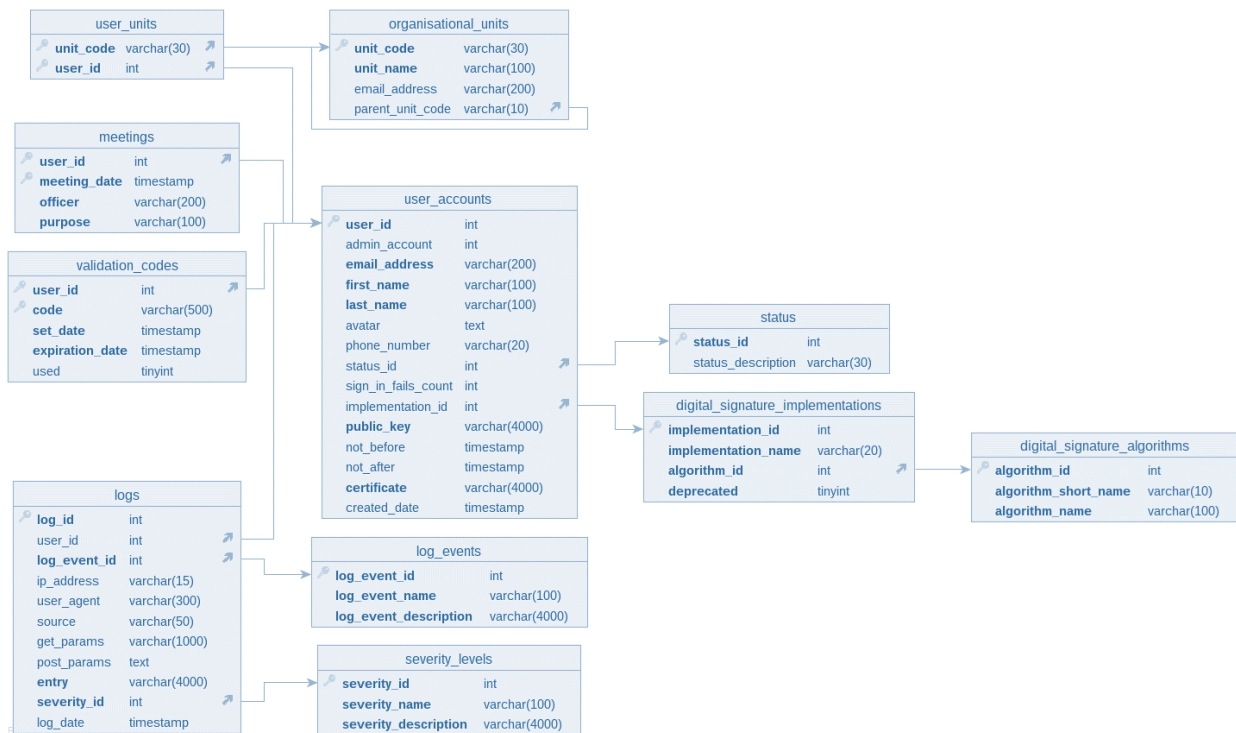


Figure 10: Database Schema for X509 Certificate Use

There are numerous advantages to using X509 certificates:

- The organization controls the data and can manage the credentials.
- The use of a signed certificate provides trust to the user enrollment, removing the need for a confirmation email.
- Adding expiration dates can help to manage user privileges easily (automatic revoking of accounts).
- The approach is easy for client: they basically only need to provide one document.
- The approach is compatible with a single-sign-on solution and automation.

There are some inconveniences as well:

- The solution is suitable for organizational management, but not for a decentralized client-server arrangement.
- User account revocation is more complex and requires additional verifications.

10.2 Sample PHP Code

In the sample PHP code, the client has submitted a certificate through the authentication protocol. The certificate is base64-encoded. The script starts after the basic verification of the submitted parameters. The certificate is analyzed using the openssl extensions of PHP. Dates are validated, subject information is extracted, the signature is verified, and if it all checks out the script adds the user to the database.

The script manages all three types of digital algorithms that can be used by the subject. The key will be converted to the standard format of the authentication protocol.

```
// Grab the contents of the certificate.
$x509data = base64_decode($cert_base64);
//print($x509data);
$cert = openssl_x509_parse($x509data);

// The certificate has been fully loaded into the variable.
// Extracting all the relevant fields.

$valid_from = date_create_from_format('ymdHisZ', $cert['validFrom'], new DateTimeZone("UTC"));
$valid_to = date_create_from_format('ymdHisZ', $cert['validTo'], new DateTimeZone("UTC"));
$now = new DateTime("now", new DateTimeZone("UTC"));

if ($valid_from > $now) {
    // The certificate is not yet active.
    print("The certificate is not yet active");
    add_log($conn, "", 3, "The provided certificate is not yet active", 3);
    $valid = false;
}
else {
    if ($valid_to < new DateTime("now", new DateTimeZone("UTC"))) {
        // The certificate is not yet active.
        print("The certificate is expired");
        add_log($conn, "", 3, "The provided certificate is expired", 3);
        $valid = false;
    }
    else {
        $email_address = $cert["subject"]["emailAddress"];
        // The email address of the subject.
        $unit_code = $cert["subject"]["OU"];
        $sql = "SELECT COUNT(*) nb FROM user_accounts WHERE email_address = ?";
        if (!$stmt = $conn->prepare($sql))
            echo $conn->error;
        $stmt->bind_param("s", $email_address);
        $result = $stmt->execute();
    }
}
```

```

$result = $stmt->get_result();
$nb_users = $result->fetch_assoc()['nb'];
$result->free_result();
$stmt->close();

if ($nb_users != 0) {
    print("There is already a registered account using that email address.");
    add_log($conn, "", 3, 'A user attempted to register an account to an email that al-
ready exists.', 2);
    $valid = false;
}
else {
    // Validating the certificate signature.
    // Get the CA certificate public key.
    $ca_pubkey = openssl_pkey_get_public(
        file_get_contents('/opt/util/serveer_rootCA.crt'));

    $keyData = openssl_pkey_get_details($ca_pubkey);

    // Verify the signature on the certificate with the CA public key.
    // 1 is good, 0 is bad. -1 is an error.

    $check = openssl_x509_verify(openssl_x509_read($x509data),
        $keyData['key']);

    if ($check != 1) {
        print "The signature verification failed: not a valid certificate";
        add_log($conn, "", 3,
            "A submitted certificate did not pass the verification", 3);
        $valid = false;
    }
    else {
        // Extracting the public key from the client certificate.
        $pub_key = openssl_pkey_get_public($x509data);
        $keyData = openssl_pkey_get_details($pub_key);

        // Format the public key to the standard.
        if (isset($keyData['rsa'])) {
            $public_key = '{"e": "0x' .
                strtoupper(bin2hex($keyData['rsa']['e'])) .
                ", "n": "0x' .
                strtoupper(bin2hex($keyData['rsa']['n'])) . "'}";
        }
        else if (isset($keyData['dsa'])) {
            $public_key = '{"y": "0x' .
                strtoupper(bin2hex($keyData['dsa']['pub_key'])) .
                ", "G": "0x' . strtoupper(bin2hex($keyData['dsa']['g'])) .
                ", "P": "0x' .
                strtoupper(bin2hex($keyData['dsa']['p'])) .
                ", "Q": "0x' .
                strtoupper(bin2hex($keyData['dsa']['q'])) . "'";
        }
        else if (isset($keyData['ec'])) {
            $public_key = '{"publicKey": "0x' .
                strtoupper(bin2hex($keyData['ec']['x'])) .
                strtoupper(bin2hex($keyData['ec']['y'])) . "'";
        }
        // Everything was verified, now adding the user to the web server.
    }
}

```

11. Appendix D – User Manual

The following pages contains a detailed user manual for the use of the authenticator software. The manual is embedded in the software and opened when invoking the Help menu.

The user manual is not for the implementation of the authentication protocol on the server side. It explains how to interact with a compatible web server to register and authenticate accounts using the authenticator.

Easy Authenticator App



User Manual

Contents

- Table of Figures 4
- 1. Easy Authenticator App User Manual 5
 - 1.1 What is an Authenticator? 5
 - 1.2 Which Web Servers are Compatible? 5
 - 1.3 Installation 5
 - 1.4 App Settings and User Preferences 6
 - 1.5 Vault Files 7
 - 1.6 Access Controls 7
 - 1.6.1 Password Protection 8
 - 1.6.2 Key File 9
 - 1.6.3 USB Security Key 9
 - 1.6.4 Yubikey Challenge-Response 9
 - 1.7 Vault File Management 10
 - 1.7.1 New, Open, Save, Close 10
 - 1.7.2 Save As 10
 - 1.7.3 Import and Export Accounts 10
 - 1.7.4 Export to Other Formats 10
 - 1.8 User Profiles 11
- 2. Account Management 12
 - 2.1 Easy Authenticator Tokens 12
 - 2.2 Server Interactions 12
 - 2.3 Register an Account 12
 - 2.3.1 Using Auto-Generated Digital Signature Keys 14
 - 2.3.2 Using a Certificate 16
 - 2.3.3 Using a Signed Certificate 16
 - 2.4 Sign into an Account 17
 - 2.5 Editing an Account on the Web Server (Except the Public Key) 19
 - 2.6 Renewing the Account's Public Key 20
 - 2.7 Digitally Sign a Privileged Operation 22
 - 2.8 Resetting a Public Key 23
 - 2.9 Terminating an Account 23
 - 2.10 Account Deletion 24
- 3. Easy Authenticator Tools 25

3.1 Yubikey Integration	25
3.1.1 Static Password with the Yubikey.....	25
3.1.2 Challenge-Response with the Yubikey	26
3.1.3 Generating a Self-Signed Certificate on the Yubikey	26
3.1.4 Importing a Certificate on the Yubikey.....	28
3.2 USB Security Key.....	28
3.2.1 What is a USB Security Key?	28
3.2.2 How to Generate a Key.....	29
3.2.3 How to Use a USB Security Key to Secure a Vault file.....	30
3.3 Certificate Viewer	30

Table of Figures

Figure 1: Authenticator App Splash Screen.....	5
Figure 2: Easy Authenticator App Settings	6
Figure 3: Vault File Icon.....	7
Figure 4: Security Parameters (Access Controls) for Vault Files	8
Figure 5: Password Management Interface	8
Figure 6: User Profiles Interface.....	11
Figure 7: Example of User Registration Interface.....	13
Figure 8: Authenticator Token for Registration	13
Figure 9: Blank Registration Form.....	14
Figure 10: Filled Registration Form	15
Figure 11: Registered Account in the Vault.....	15
Figure 12: Display of Account Details.....	16
Figure 13: Registration Requiring a Signed Certificate	17
Figure 14: Sign-In Interface.....	18
Figure 15: Sign-In Authenticator Token	18
Figure 16: Sign-In Authenticator Interface.....	18
Figure 17: User Profile Interface.....	19
Figure 18: Account Modification Interface.....	20
Figure 19: Account Modification Authenticator Token	20
Figure 20: Example of a Key Renewal Interface.....	21
Figure 21: Public Key Renew Authenticator Token.....	21
Figure 22: Public Key Renew.....	21
Figure 23: Digitally Signing a Privileged Operation	22
Figure 24: Digital Signature Authenticator Token	22
Figure 25: Digitally Signing a Generic Operation.....	23
Figure 26: Account Termination Interface.....	24
Figure 27: Authenticator Token for Account Termination.....	24
Figure 28: Easy Authenticator App Account Termination Interface	24
Figure 29: Yubikey Settings Interface.....	25
Figure 30: The Yubikey PIV Editor Interface	27
Figure 31: Generating a self-signed Certificate on a Yubikey.....	28
Figure 32: USB Security Key Editor Interface	29
Figure 33: Selected USB Security Key for Vault Access.....	30
Figure 34: Details of a Loaded Certificate.....	31

1. Easy Authenticator App User Manual

1.1 What is an Authenticator?

An authenticator app either performs the tasks associated to authentication on behalf of the user or assists with the tasks to speed up the process and facilitate the interactions. The Easy Authenticator app is based on the use of digital signatures to authenticate the user through a challenge-response protocol, which removes the requirement to generate and share a password with the web server.

To validate a digital signature, the web server needs to have the user's public key. This is what will be provided to the web server at registration, instead of a sensitive password. The public key is not sensitive and does not need to be protected. The user's private key is sensitive and will not be shared with anybody. It will remain under the control of the user at all times.

In the IT world, digital signatures are commonly used to provide authenticity when interacting with entities through a public key infrastructure (PKI) such as the Internet. Organizations use them to protect the integrity of documents. Digital signatures also protect the integrity of cryptocurrency transactions, such as the ones involving Bitcoin.

1.2 Which Web Servers are Compatible?

To use the Easy Authenticator app with a web server, this server must implement the communication protocol required for the interactions. A web server that authenticates users through passwords is not compatible with the Easy Authenticator app. Easy Authenticator can be combined with a second factor of authentication to increase the robustness of the authentication process.

1.3 Installation

The Easy Authenticator is distributed as an App for Windows 10. At installation, you will see a splash screen like the one displayed in Figure 1.

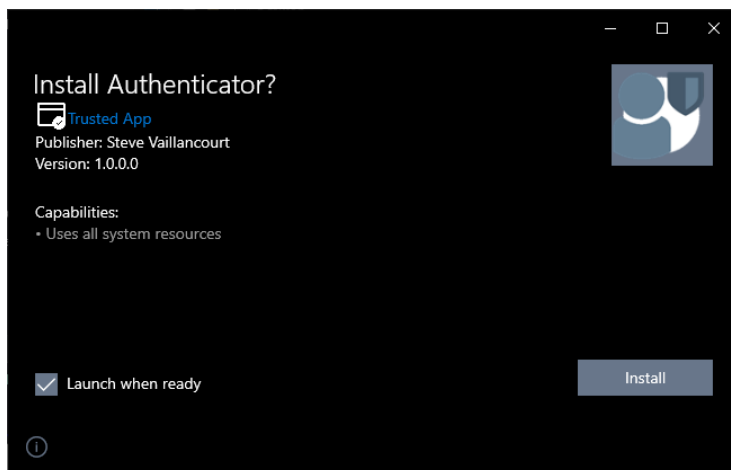


Figure 1: Authenticator App Splash Screen

After installation, the Easy Authenticator app will be integrated into the Windows Operating System and be ready to use.

1.4 App Settings and User Preferences

After installing the Easy Authenticator, it is recommended to edit the user preferences to get the best possible user experience. The settings are accessed from the menu *Edit / Settings...* See Figure 2 for an example of the interface.

The following settings can be edited:

- *Keep Application on Top*

Easy Authenticator makes use drag-and-drop operations to exchange information back and forth with web servers. To keep the Easy Authenticator app from going in the background during those operations, check the box and it will remain on top of other windows. This menu specifies the user's general preference, but the status can also be quickly edited from the *View / Always on Top* menu.

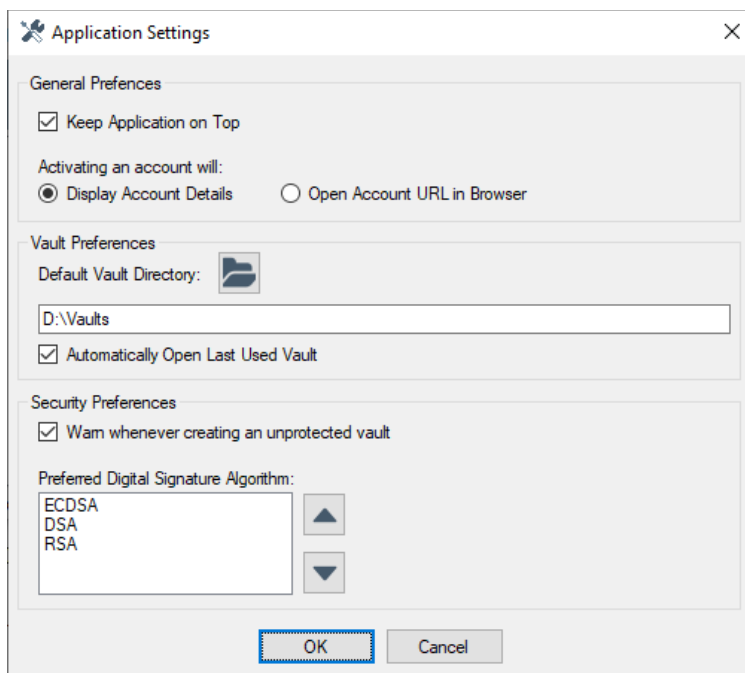


Figure 2: Easy Authenticator App Settings

- *Activating an account will: Display Account Details / Open account URL in Browser*

Activating an account means triggering it by double-clicking, selecting and pressing the space bar, or any other activation method. This option allows to specify the behaviour of the app when this happens: displaying the account details, or opening the account's URL in the user's default browser.

- *Default Vault Directory*

This value should be set to point to the folder where the Easy Authenticator app should go to by default to save and open vault files. This will define the initial directory, but it is always possible to navigate to whichever directory on your system the file is located or where it should be saved.

- *Warn whenever creating an unprotected vault*

Unprotected vaults have no [security controls](#) applied to them. Even though they will still be encrypted, the encryption is trivial and should not be seen as a real protection. Other actions that can leave the vault file unprotected are when they are [exported to other formats](#). Use this option to request a warning message whenever a vault file is about to be saved without proper security controls applied.

- *Preferred Digital Signature Algorithm*

This option allows to order the digital signature algorithms supported by Easy Authenticator in order of preference. Not every web server will support all three, and when registering a key with them, the key must conform to a supported implementation. The Easy Authenticator app will allow the user to select supported algorithms only. However, if multiple algorithms are supported, the Easy Authenticator app will sort the supported implementations using the order defined by this option. In case of doubt, leaving this setting at its default value is recommended.

1.5 Vault Files

Vault files are used by the Easy Authenticator app to store the account information, including private key information used for digital signatures. This information is sensitive and must be protected. A vault file has the extension **.vault* and is associated to the app at installation. Users can create as many vault files as needed to support their requirements. The icon associated to the file will appear as illustrated in Figure 3.



Figure 3: Vault File Icon

1.6 Access Controls

To protect the account stored in vault files, access controls will be used to generate a symmetric encryption key. If no access controls are used, the encryption key is trivial and offers no real protection. Several access controls can be compounded to create a multi-factor encryption key. See Figure 4 for the vault access interface.

There are two ways to apply access controls to a vault. The first is at its creation, using the *File / New* menu. The other is when an existing vault is saved using the *File / Save as...* menu, which allows the user to save the vault file at a different location and/or with different access controls. When opening a file, the user must provide the same access controls to recreate the symmetric encryption key.



Warning: If the user cannot recreate the access controls to decrypt a vault, it might be impossible to access the account information that it contains: accounts will need to be [reset](#).

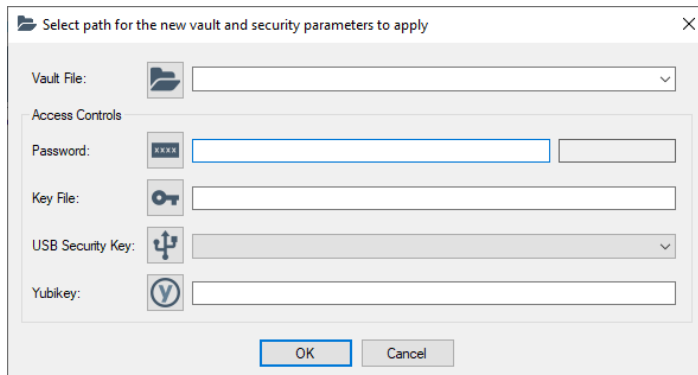


Figure 4: Security Parameters (Access Controls) for Vault Files

1.6.1 Password Protection

The encryption key can be derived from a password applied as a security control. The strength of the key will be correlated with the unpredictability and entropy of the password. To provide insight into the entropy and predictability of the key, the user can press the button next to the password entry box to display a password management interface. An example of the interface is provided in Figure 5.

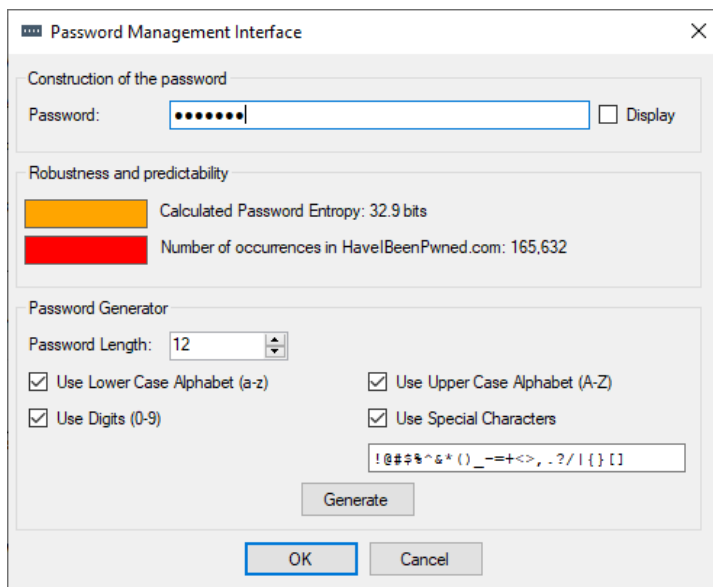


Figure 5: Password Management Interface

The interface uses the public API of the website [HaveIBeenPwned.com](https://haveibeenpwned.com) which researches and catalogues breaches to online accounts. When providing a password, the number of times it appears in known breaches will be displayed.

When protecting a vault file using a password, it is important that it is both unpredictable and presents a high value of entropy (both feedback colors will be green) so that someone that has access to the encrypted vault file will not succeed in decrypting the information.

The interface can also be used as a password generator. Make sure that you can keep track of passwords that you apply to encrypt vault files. A recommended approach is to use a password manager or a hardware device to protect your password. One such device is the Yubikey, which can be configured from the menu *Edit / Yubikey Settings...* for a static password use.

1.6.2 Key File

Another way to create the encryption key is to use a key file. This can be any file of any type provided that:

- The contents never change
- The user has easy access to it to decrypt the vault file
- The file is kept secure and out of reach from unauthorized individuals

The encryption key will be derived from the selected file using an approved hashing function: Secure Hash Algorithm 256 bits (SHA-256). It is advised to keep an offline copy of the key file. Also, it is advisable to select a file that is not “obviously” a key file for an Easy Authenticator vault, unless that file is securely stored and out of reach of unauthorized individuals.

1.6.3 USB Security Key

For your convenience, you can convert a USB key into a hardware-based [key file](#). To make it recognizable as such for the Easy Authenticator app, a USB Security Key must contain two specific files at the root. One contains the identification string of the key and the other contains the key file. Easy Authenticator can assist you in the creation of a USB Security Key (see [3.2.2 How to generate a key?](#)). This key must be handled with caution.

If a USB key is connected to your system and holds the two files to identify it as such, the user can click the icon next to the USB Security Key input to automatically select the first detected USB Security Key. If multiple keys are connected, the user can also use the drop-down list to select the required key.

1.6.4 Yubikey Challenge-Response

Easy Authenticator incorporates the use of a Yubikey. The YubiKey is a hardware authentication device manufactured by Yubico to protect access to computers, networks, and online services. One method of providing access control security to a vault file is to use the challenge-response feature of the Yubikey. The key must be configured for such use beforehand (see [3.1 Yubikey integration](#)). If your key is configured for challenge-response, clicking the icon will trigger the challenge and write the response in the corresponding entry. This can be used to create the encryption key of a vault file quickly and easily. If key requires touch activation, it will blink, and the user will need to press a finger to the key before the response to the challenge is provided.

1.7 Vault File Management

There are several operations that can be accomplished with vault files. Although encrypted, they remain files that can be copied, moved, and renamed from the operating system. Accessing the contents means that the Easy Authenticator app will have to be provided with the security parameters to decrypt the file.

1.7.1 New, Open, Save, Close

To get started with a vault file, the typical approach is to select New, either from the menu or toolbar, and create the vault file with the chosen security parameters. Saving the file updates the content and reapplies the encryption. The user will not need to submit the security parameters every time the vault is save to the file; the app will remember them from when you the file was opened. Closing the file means that the app remains opened, but the vault file will be encrypted and closed.

1.7.2 Save As

The *File / Save As...* menu is used when the user wishes to modify the security parameters and/or the location of the file. The user can overwrite the existing file with this operation, meaning that only the security controls would change. When writing the file to a new location, the original file, with its security parameters, still exists. If it is no longer required, it should be deleted after verifying that the new vault file works as intended.

1.7.3 Import and Export Accounts

It is possible to merge or reorganize vaults using the Export and Import functionalities. To export accounts, they must first be selected in the interface. The menu *File / Export / Selected Accounts...* will allow the user to save a new vault file containing the selected accounts. Specific security parameters for that vault can be applied during this operation.

To add existing accounts from another vault file to the currently opened vault file, the *File / Import / Accounts from vault...* menu is used. All the accounts from that other vault file that do not exist already in the current vault will be imported. Of course, the security controls of the other vault file will need to be provided during the operation.

1.7.4 Export to Other Formats

For convenience, it is possible to export accounts into alternative formats. Using the menu *File / Export / Vault to CSV...* will allow the creation of a **.csv (Comma Separated Values)* file containing the details of the accounts. Also, the current vault can be exported to a plain text, readable format using *File / Export / Vault to Plain Text...* These operations are one-way only: files in that format cannot be imported back into a vault.

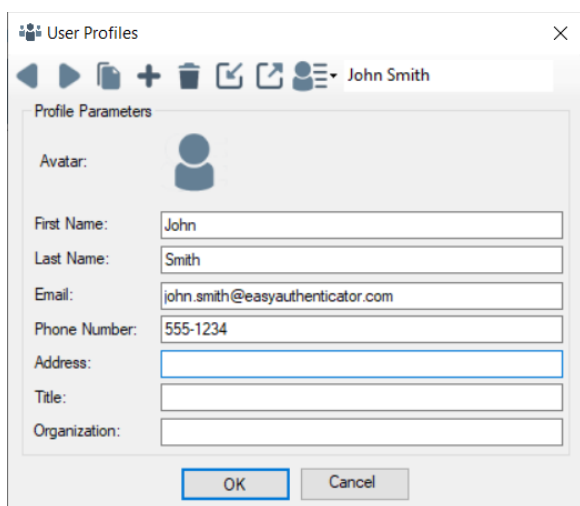


Warning: Vaults exported to CSV or Plain Text are not protected. The information contained in them is directly readable and can lead to disclosure of sensitive data.

1.8 User Profiles

User profiles are part of the settings of the Easy Authenticator app. They allow a user to predefine digital identities for quick registration with web servers. A web server will request several points of information such as email, first and last names, phone number, and date of birth. Some will be required while others will be optional. This is dependent on the web server. To avoid re-entering this data each time, the Easy Authenticator can capture this “profile” information once and provide the data to fill registration forms. The user will always have a chance to validate, edit, or remove information before sending it over to the web server. Registration is never automated. See Figure 6 for an example of a user profile define in the app.

The user profile interface allows the user to create as many different profiles as needed. The toolbar can be used to view, edit, clone, and delete profiles. Profiles can be exported to an *XML (eXtensible Markup Language)* file for safekeeping or to transfer them to another installation of the app.



The screenshot shows a dialog box titled "User Profiles" with a close button (X) in the top right corner. Below the title bar is a toolbar containing icons for back, forward, home, add, delete, clone, and edit. A dropdown menu next to the edit icon shows the name "John Smith". The main area is labeled "Profile Parameters" and contains several input fields: "Avatar" with a person icon, "First Name:" with the value "John", "Last Name:" with the value "Smith", "Email:" with the value "john.smith@easyauthenticator.com", "Phone Number:" with the value "555-1234", "Address:" (empty), "Title:" (empty), and "Organization:" (empty). At the bottom of the dialog are "OK" and "Cancel" buttons.

Figure 6: User Profiles Interface

2. Account Management

The goal of the Easy Authenticator app is to facilitate and secure the use of online accounts. The accounts will be stored in a vault file and protected with access controls. This section explains how to perform the interactions with a compatible web server for the creation, management, and ultimately destruction of user accounts.

2.1 Easy Authenticator Tokens

Easy Authenticator maintains a separation between the web servers and itself. The interaction is never fully automated. The web server will encapsulate communications and make them available from a web page. To retrieve the message, the user must grab the token containing the message, drag and drop it in the Easy Authenticator interface. The app will read and analyze the message and assist in preparing the needed response for the interaction to take place.

2.2 Server Interactions

The protocol through which Easy Authenticator talks to web servers recognizes seven distinct interactions between clients and web servers. These interactions will start with a message from the server that will have to be dragged and dropped into the Easy Authenticator app. The client response, which will be defined in the application is dragged back and dropped onto the web page of the server when ready. No interaction is automated to increase the security of the operations and give full control to the user.

The following are the interactions that are recognized by the Easy Authenticator app:

- Account Registration: defining and registering a new account on a web server.
- Signing in: starting a new user session on the web server using an existing account.
- Editing the user account/profile: modifying details of the account stored on the web server.
- Renewing the public key: providing an updated public key to the web server. This is analogous to changing the password protecting an account in password-protected accounts.
- Digitally signing an operation: validating an action by providing a digital signature to confirm it.
- Resetting the public key: this operation is accomplished, if the web server allows for it, so that a forgotten key, or a key that is no longer available can be reset and the user can keep using the account.
- Terminating the account: this is an exchange between the web server and the client by which the user expresses the intention of ending the service and invalidating the account.

2.3 Register an Account

The first basic interaction that needs to occur is the registration of the account on the web server. Whether it is accomplished by the user or by an authorized proxy, this will need to happen before any other interaction can occur.

The operation consists of a back-and-forth negotiation with the web server. By dragging the token to the Easy Authenticator app, it can analyze what the server is providing (identification) and what the server is requesting, i.e., the parameters required to build an account. This will consist of typical information such as an email address, first and last names, and non-typical information depending on the requirements that the server imposes to build the account. For instance, a school could require a student ID number when creating an account.

Of course, the interface provided for the exchange will be at the discretion of the web server. Figure 7 provides an example of what such an interface could look like. The authenticator token (see Figure 8) is the vehicle to transport the registration form to the Easy Authenticator app.

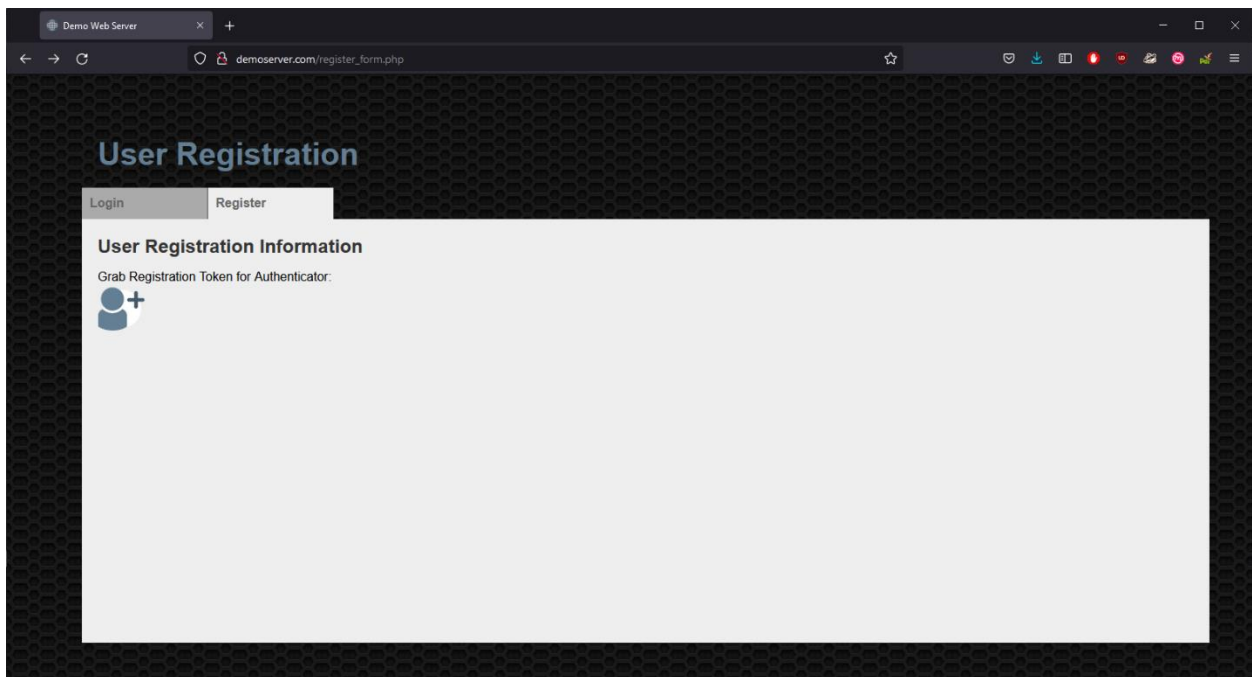


Figure 7: Example of User Registration Interface



Figure 8: Authenticator Token for Registration

Once the token has been dragged and dropped into the Easy Authenticator app, the message is analyzed, and a dynamic form is presented to the user. The form displays the required and optional fields to fill. At the top of the form is the server identification: the name, icon, and URL of the server. See Figure 9 for an example of such an identification.

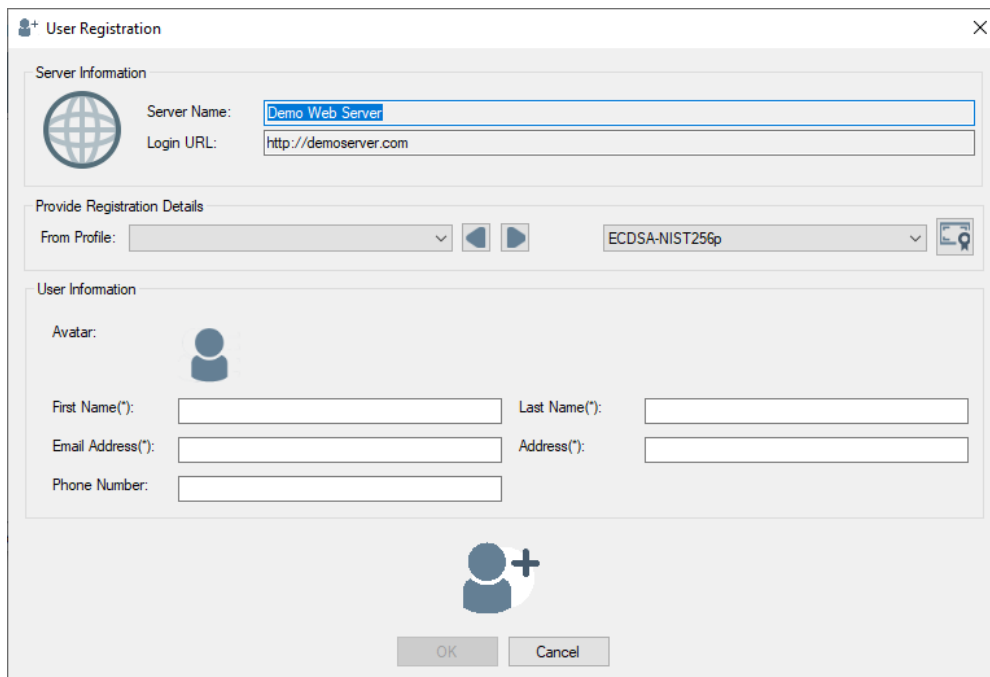
The image shows a 'User Registration' dialog box with a title bar containing a plus icon and a close button. The dialog is divided into three main sections. The top section, 'Server Information', features a globe icon and two text input fields: 'Server Name' with the value 'Demo Web Server' and 'Login URL' with the value 'http://demoserver.com'. The middle section, 'Provide Registration Details', includes a 'From Profile' dropdown menu, navigation arrows, and a dropdown menu for the signature algorithm, currently set to 'ECDSA-NIST256p', with a key icon to its right. The bottom section, 'User Information', contains an 'Avatar' field with a person icon, and five text input fields: 'First Name(*)', 'Last Name(*)', 'Email Address(*)', 'Address(*)', and 'Phone Number'. At the bottom center is a plus icon with a person silhouette, and at the bottom are 'OK' and 'Cancel' buttons.

Figure 9: Blank Registration Form

If profiles have been defined in the settings of the Easy Authenticator app, they can be invoked to instantly fill out the form. The exchange protocol defines several keywords that are associated to the profile, and by using these keywords the profile can link up the stored information. If required information is missing or some information needs to be edited, the user can modify it in the form. The form is not automatically sent to the web server in any case: it will only be submitted through a drag-and-drop operation.

2.3.1 Using Auto-Generated Digital Signature Keys

The goal of the Easy Authenticator app is to remove the use of passwords by replacing them with digital signatures. The registration process needs to provide the server with a public key: in the example seen in Figure 9, the key is generated by the authenticator, using the algorithm specified (Elliptic Curve Digital Signature Algorithm using NIST curve 256p). The authenticator will manage the private and public keys and store them in the vault, while a copy of the public key goes to the web server to complete the registration.

Figure 10: Filled Registration Form

To complete the registration process, the form must be submitted to the web server, and the form data must be accepted as valid. The submission is done by dragging the authenticator token that holds the registration information back to the web server.

The web server will validate the account using which ever means it deems acceptable. For example, sending a validation email to the account of the newly registered user to prove ownership of the account is likely to be sufficient in many cases, similarly to password-protected accounts. The user can save the newly created account in the authenticator.



Figure 11: Registered Account in the Vault

Account Information

Account Information

Email: john.smith@easyauthenticator.com

Server ID: Demo Web Server

Private Key: Internal, using ECDSA-NIST256p with SHA-256

Login URL: http://demoserver.com

Notes:

OK Cancel

Figure 12: Display of Account Details

2.3.2 Using a Certificate

As seen in [section 2.3.1](#), the registration process can leverage the authenticator to create a pair of encryption keys especially for the account. However, it is also possible to refer to an existing X509 digital certificate that contains the public key to use for the account. The same certificate can be used for several accounts since the public key is not sensitive.

When using a certificate to register the account, the user clicks the *Certificate* button on the registration form (see Figure 10) to select the certificate, then provides the password that protects the file, and finally loads the information. This will provide the public key for the registration process. The rest of the information must be provided separately, either from a stored profile or by manually entering the details.

2.3.3 Using a Signed Certificate

In some cases, the web server will ask for a certificate that holds the user's public key, and that is validated by the signature of a Certification Authority, or CA. This provides authenticity to the registration process and validates that the information was verified by a registration authority, or RA. See Figure 13 for an example of such a registration form.

In the message from the web server, the form displays a certificate field, flagged as mandatory. The description of the field states that a certificate, signed by a specific CA, must be provided. The user will be authenticated to that digital identity if they are in possession of the corresponding private key.

This process is like the previously detailed operation to register the account. The difference is that instead of trusting the information provided by the user, the server instead will place its trust in the CA that signed the certificate.

Figure 13: Registration Requiring a Signed Certificate

2.4 Sign into an Account

When the account is registered and validated, the user can now sign into the account and begin an authenticated session. The sign-in interface provides an authenticator token to start the operation (see Figure 15). The operation is like the registration process and all the other interactions with the web server: the token is dragged onto the authenticator app to be analyzed and to prepare a response.

When dragging the token to the authenticator app, the server identification will be read from the message, and the corresponding account will be identified, if it exists. Then, using the private key of the account, Easy Authenticator will generate a digital signature on the challenge provided by the server (see Figure 16).

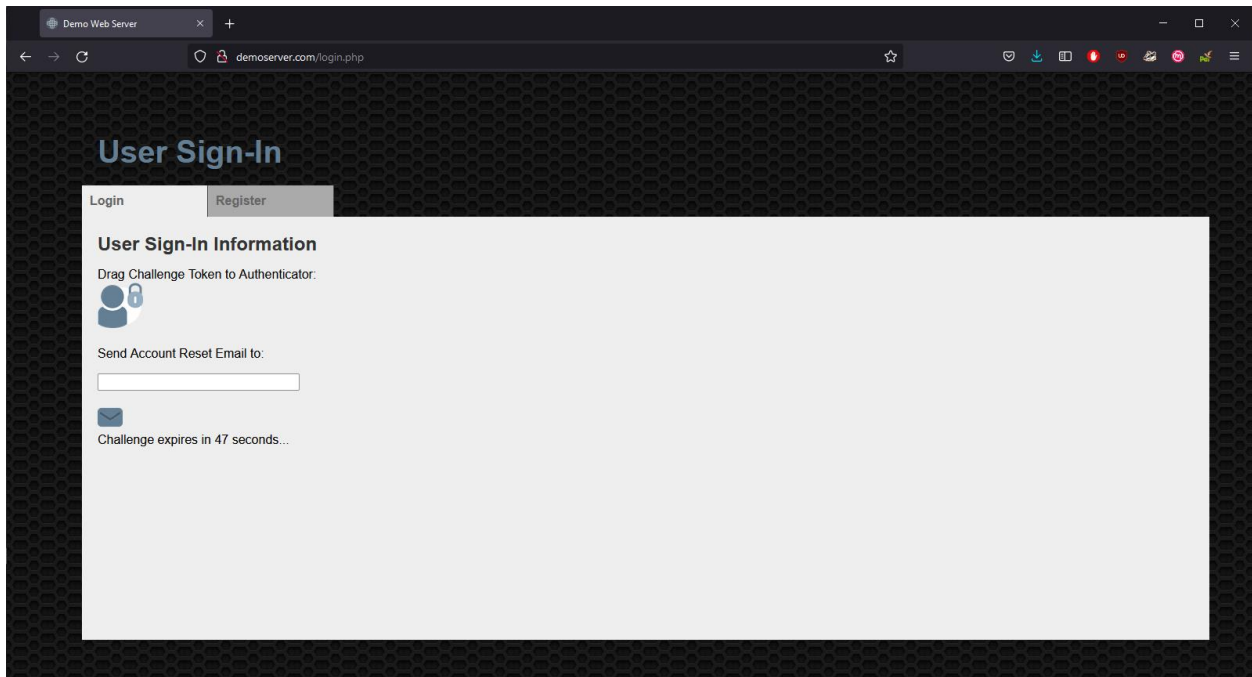


Figure 14: Sign-In Interface



Figure 15: Sign-In Authenticator Token

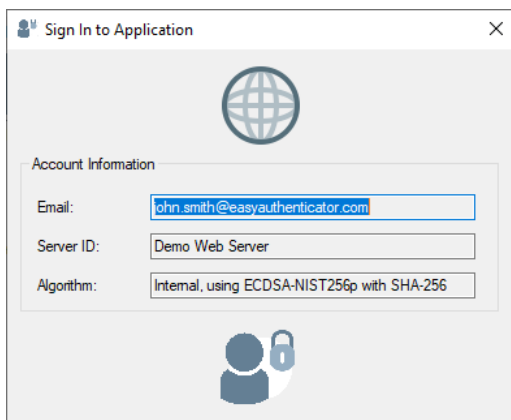


Figure 16: Sign-In Authenticator Interface

The sign-in procedure will display the server identification and a description of the private key used for the signature. If the details correspond to the operation being performed, the user should then drag back the authenticator token from this interface onto the web server.

2.5 Editing an Account on the Web Server (Except the Public Key)

During the lifetime of the account, certain details may need to be edited, such as the phone number, the address, the job title, or even the email address. This operation only concerns the details of the user account that are not used for the digital signatures. To request an account modification, the user will typically access the profile interface of the web server and drag the authenticator token for profile modification to the Easy Authenticator app. See Figure 17 for an example of such an interface.

The authenticator token for account modification (see Figure 19) will hold the current details of the account. Once in the authenticator, the user can modify them, possibly by using a profile. See Figure 18 for an example of the account modification interface.

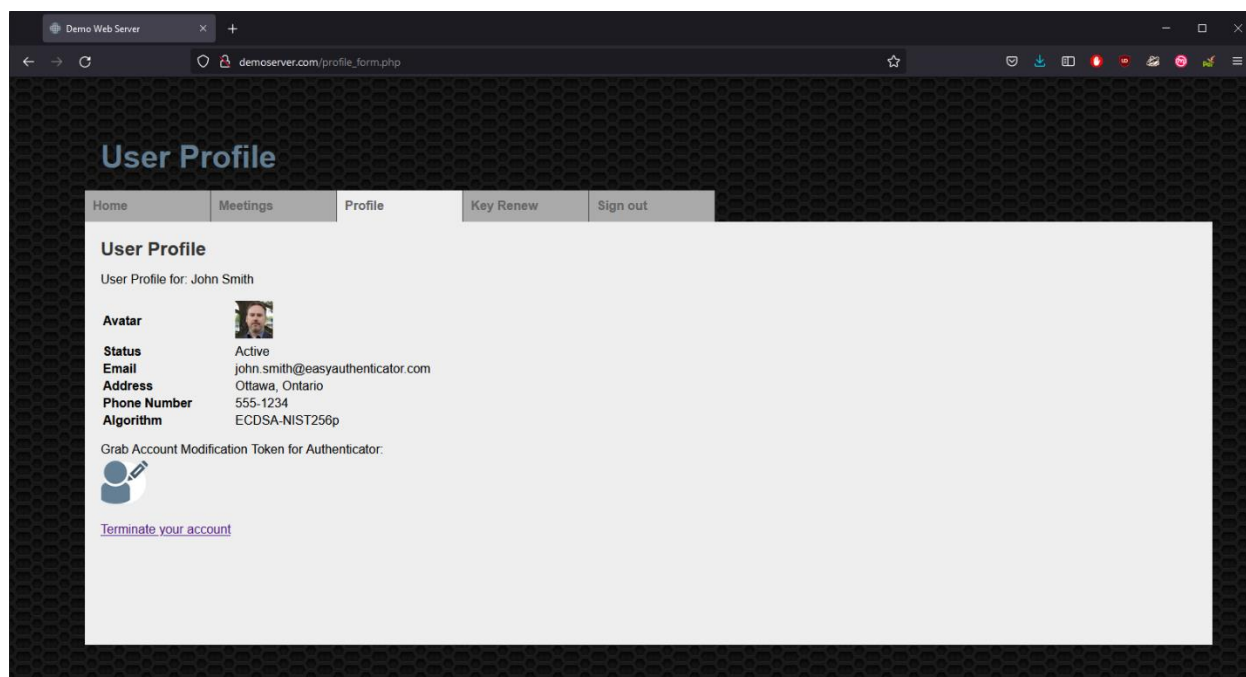


Figure 17: User Profile Interface

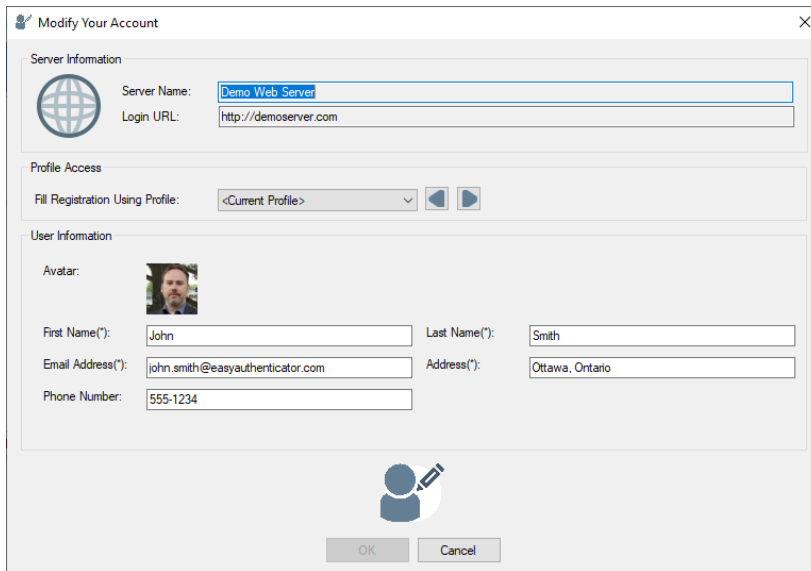


Figure 18: Account Modification Interface



Figure 19: Account Modification Authenticator Token

2.6 Renewing the Account's Public Key

The user's public key is stored on the web server to validate the user's digital signature. The public key must correspond to the user's private key, securely detained by the user. If for some reason, the public key needs to be modified, then the user must request a key renewing operation to the web server. Reasons for this might include the digital signature algorithm becoming deprecated. It could also happen if there is some concern that the key might be exposed, through theft or loss of hardware for instance.

The interface that allows the user to renew a key renew is available after the user is authenticated. This means that a valid key must be in possession of the user to do such an operation. If the user does not possess such a key, then the operation required is a [public key reset](#).

When the authenticator token for a key renewal (see Figure 21) is dragged into the Easy Authenticator interface, the application will identify the server through the information provided in the message, and digitally sign the response that will validate the user and provide a new public key. The operation can leverage a certificate, self-signed or CA-signed, as a vehicle for the new public key provided to the server.

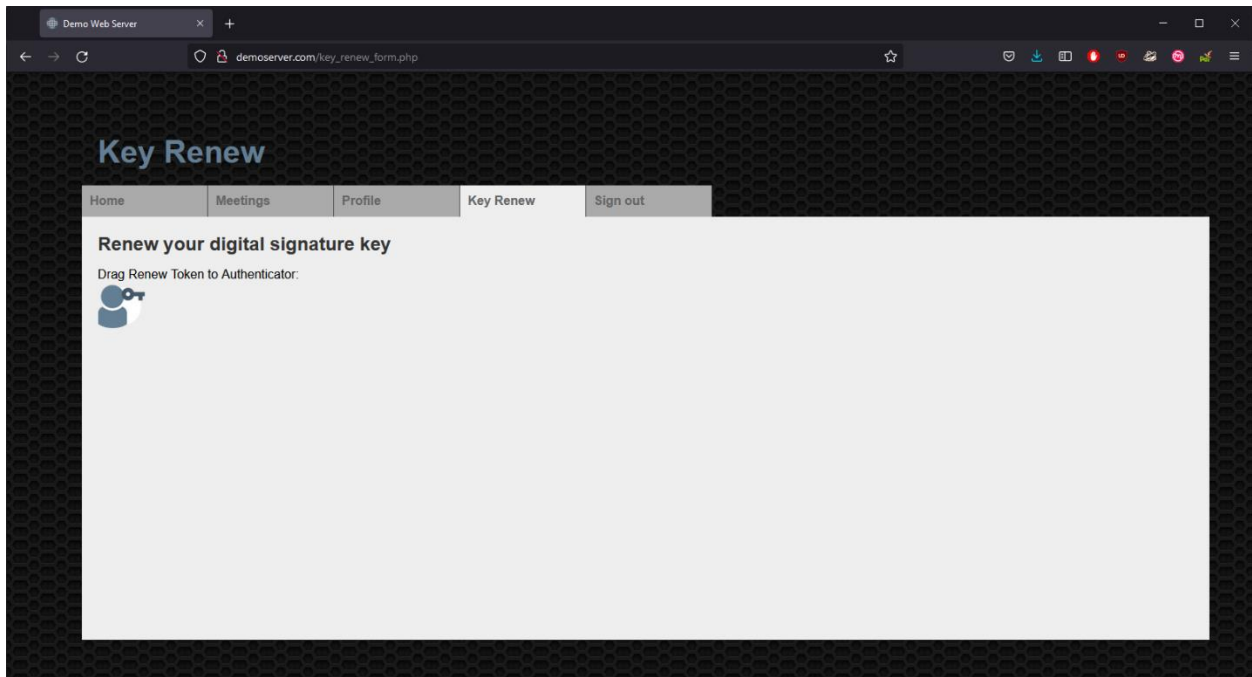


Figure 20: Example of a Key Renewal Interface



Figure 21: Public Key Renew Authenticator Token

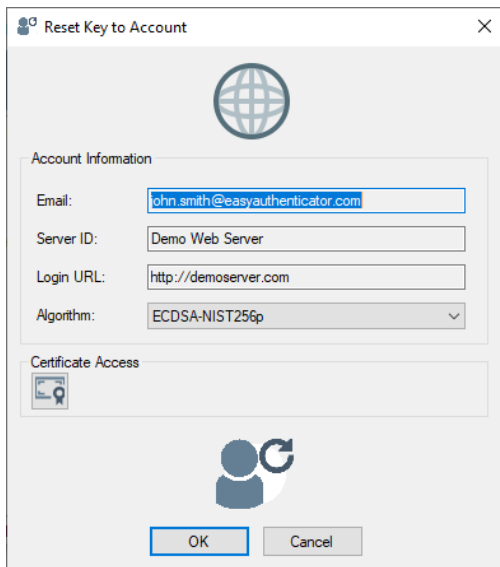


Figure 22: Public Key Renew

2.7 Digitally Sign a Privileged Operation

Digital signatures are a cryptographic tool that provides authenticity to transactions. It can be used in many different contexts, including authentication, but the basic principle is to validate information by encrypting the data with a private key. A privileged operation, one that holds a certain sensitivity, can be specifically signed to protect its integrity and authenticity. This could be described as a generic sensitive transaction from the client to the web server. As with other transaction, the authenticity is validated by a digital signature, with a back-and-forth drag-and-drop operation.

The authenticator token for generic digital signature (see Figure 24) will contain the data to sign, and the authenticator app will take care of the rest. An example of an interface providing a sensitive operation is provided in Figure 23. In the transaction, the web server will encode what is being validated through a digital signature (see Figure 25).

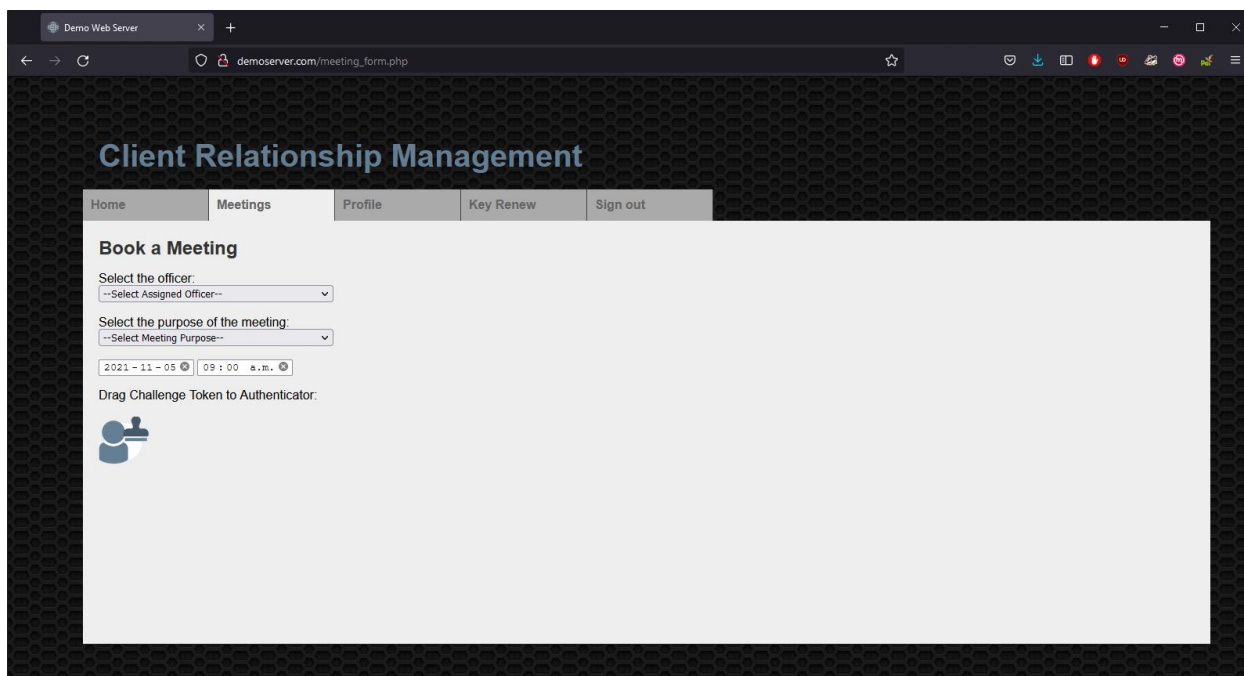


Figure 23: Digitally Signing a Privileged Operation



Figure 24: Digital Signature Authenticator Token

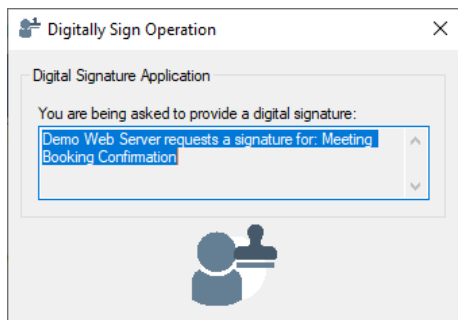


Figure 25: Digitally Signing a Generic Operation

2.8 Resetting a Public Key

An account reset operation is accomplished when the private key is no longer available. This could happen if the vault is lost, if an access control that protects the vault is no longer, or any other reason that prevents the user from opening an authenticated web session.

The process of resetting an account comes with risk. Threat actors could attempt to hijack an account by abusing the account reset procedure. Web service providers need to assess the risk and provide an account reset methodology that mitigates undue risk. This could involve using an email or a phone number associated with the account to initiate a reset or enquiring to customer tech support to resolve the issue. Some service providers might use an associated peer account to validate the operation.

2.9 Terminating an Account

A specific type of interaction between the user and the web server is the request the termination of the user's account. This signals an intention to end the client engagement and invalidate the account on the server side. It is a privileged operation and needs to be digitally signed to protect the integrity of the account.

The user must be within an authenticated session to request an account termination. The procedure starts by dragging an account termination token (see Figure 27) to the authenticator app. The app will identify the account with the information embedded into the token. The user will confirm the intent by dragging the token back to the web server. The exact nature of an account termination is dependent on the web service provider. On the client side, the authenticator app will delete the account from the vault, as it should no longer be usable.

An example of an account termination interface is provided in Figure 26, and an example of the Easy Authenticator interface for account termination can be seen in Figure 28.

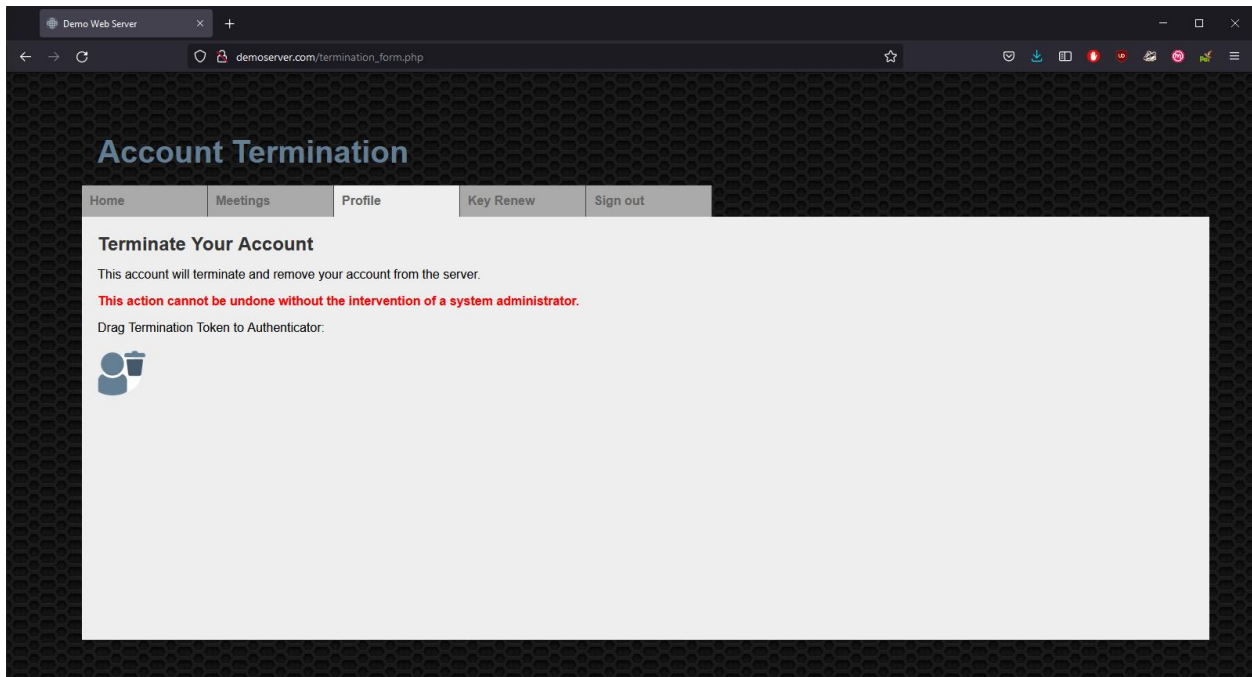


Figure 26: Account Termination Interface



Figure 27: Authenticator Token for Account Termination

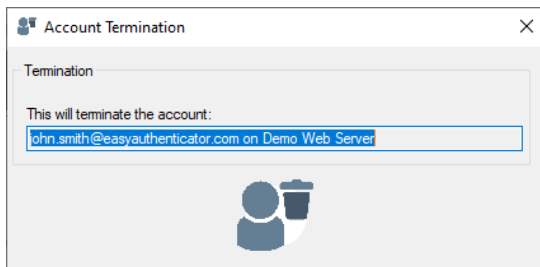


Figure 28: Easy Authenticator App Account Termination Interface

2.10 Account Deletion

Note that it is possible to delete an account from the vault directly, using the Easy Authenticator interface. This does not terminate the account on the related web server. No information will be relayed to the web server from this operation. Furthermore, deleting the account directly will make it impossible to sign in using the account and performing an account termination procedure on the web server. An account should only be directly deleted from the vault once it is already invalidated on the server side.



Warning: Directly deleting the account removes it from the vault but not the web server. To access the account again, it will likely need to be reset.

3. Easy Authenticator Tools

3.1 Yubikey Integration

Easy Authenticator integrates several uses of a Yubikey. It can be used to secure access to vault files, generate and store private and public keys, and certificates.

To access the basic settings of a connected Yubikey, the user must use the interface at *Edit / Yubikey Settings...*

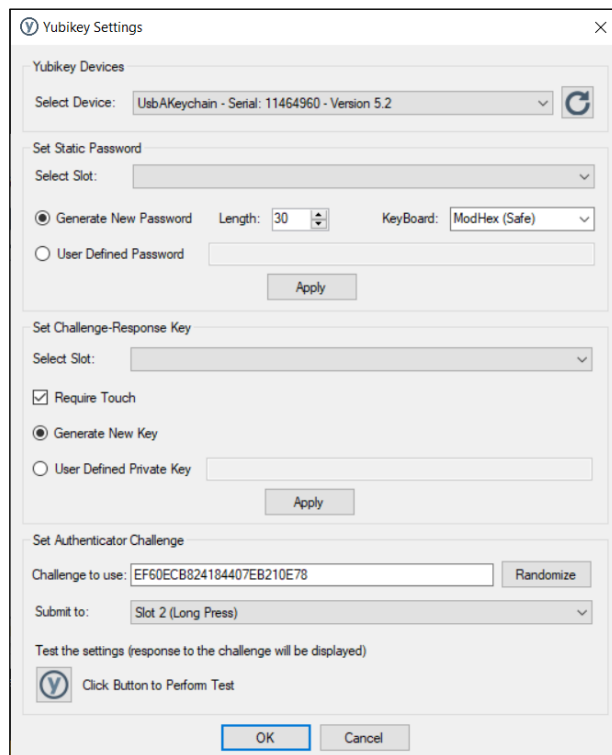


Figure 29: Yubikey Settings Interface


3.1.1 Static Password with the Yubikey

Passwords can be used to protect vault files, but they are not convenient to generate and remember. User can be tempted to use small and easy to remember passwords which diminishes the protection afforded. A solution for this is to use the static password feature of a Yubikey to store, remember, and type the static password. Writing the password becomes as simple as placing a finger on the connected Yubikey.

There are two slots on the Yubikey reserved for either a static password or a challenge-response: slot 1 (short press) and slot 2 (long press). The interface (see Figure 29) allows the user to set a static password. It can be set by the user, or randomly generated by the Yubikey.

When choosing to have the Yubikey generate the password, select the keyboard configuration to use. Because the Yubikey emulates keystrokes on the keyboard, the password generated may be affected by the keyboard configuration. Yubico has developed the modified hexadecimal keyboard (ModHex) to generate passwords that are consistent on any keyboard configuration. This limits the characters that can be used and diminishes the overall entropy, but this can be compensated with longer passwords. If the keyboard configuration used is always going to be English US, this can be selected for the password generation.


To test the password, simply place the focus on an editable entry form (like in a basic text editor) and press a finger to the key. The password (in clear) will be typed in the entry form.

 **Warning:** Loss or failure of the Yubikey could mean that the password is no longer retrievable. It is recommended to keep a copy of the password in a safe location.

3.1.2 Challenge-Response with the Yubikey

The challenge-response feature can be used to protect access to a vault file. There are two components involved in this operation: the private key and the challenge submitted. Because the encryption and decryption processes need to generate a specific key, the same private key and challenge must be used consistently to access the file. The private key will be stored on the Yubikey (in slot 1 or 2) and the challenge will be stored in the settings of the Easy Authenticator app. A threat actor trying to decrypt a vault file protected with the scheme will need both pieces of information to be successful.

To configure the parameters in the Easy Authenticator settings, In the Yubikey settings interface (see Figure 29), the user should first select the device, then the slot that will be configured for the challenge-response operation. In the section “Set Authenticator Challenge”, the user can define which challenge to submit to the Yubikey and to which slot. Those settings will be used when creating a Yubikey access control.

 **Warning:** To have the capacity to recreate those settings (on another Yubikey for instance), you must keep the private key and the challenge in a safe location, preferably offline.

3.1.3 Generating a Self-Signed Certificate on the Yubikey

A Yubikey has a total of 25 slots that can contain Personal Identification Verification (PIV). While some are reserved for specific uses, several can be leveraged by the user for authentication and digital signatures. Easy Authenticator can interact with the Yubikey to generate pairs of private/public keys to be stored in PIV slots. The private key is then securely stored on the hardware device and is not in the associated vault. The vault will only contain tracking information: the Yubikey serial number and the slot where the key is located. This allows a quick and easy interaction through Easy Authenticator when managing accounts. Several accounts can be protected through the same digital identity.

A Yubikey can store a private key and its corresponding public key in a slot. The public key is embedded in a certificate for standardized manipulation of the data. Easy authenticator can communicate with a Yubikey to generate such information and store it on the Yubikey.

To access this functionality, use *Tools / Yubikey PIV Editor...* An example is provided in Figure 30.

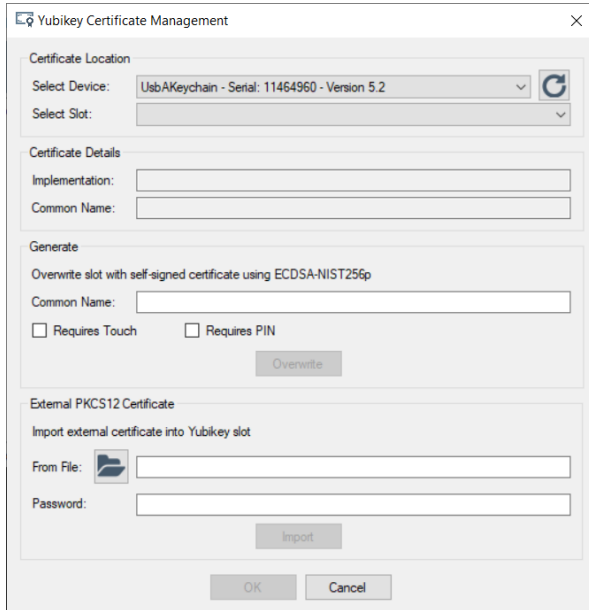


Figure 30: The Yubikey PIV Editor Interface

To get started, the Yubikey must first be selected at the top of the interface. If the key does not appear, or is connected after the interface has been generated, the *Refresh* button to the right can be used to actualize the list of devices. Once the key is selected, the slot that will be edited must be selected. It is possible to overwrite a slot that already contains information or to put information into an empty slot. When overwriting a slot, the information it previously contained will not longer be available and accounts requiring the private key that was stored at that location will become unreachable.

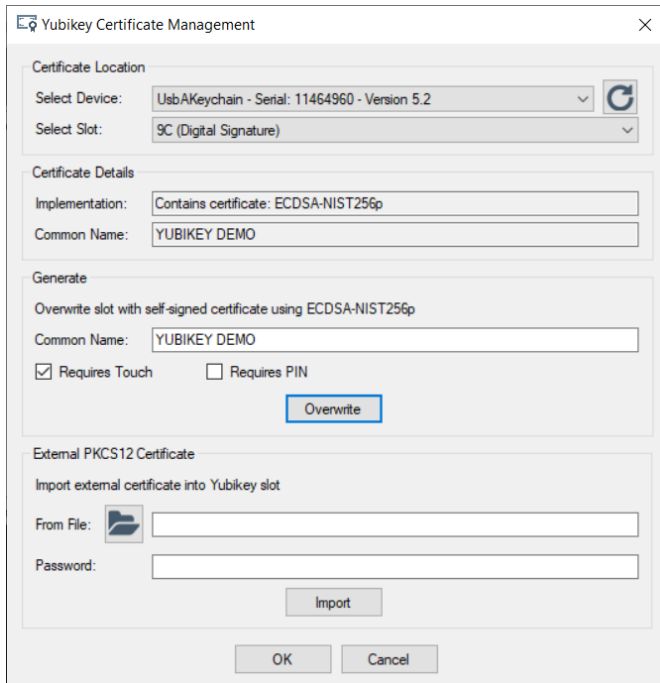


Figure 31: Generating a self-signed Certificate on a Yubikey

3.1.4 Importing a Certificate on the Yubikey

The bottom portion of the interface is to load an existing certificate into a Yubikey Slot. If you are registering an account that is validated through a signed certificate, this certificate can be loaded into the Yubikey for management and protection, instead of being in a password-protected file.

The format expected is PKCS#12, usually having a file extension of *.pfx*, *.pem*, or *.p12*. This format encapsulates the public certificate and the private key in one file. There is usually a password protection on the file, and the password must be provided when importing the certificate. The private key will be stored alongside the certificate in the Yubikey. Afterward, the certificate can be exported but not the private key, so it is recommended to keep a copy of the certificate file in a safe location.

Once imported, associate the account that is being registered with the certificate in the Yubikey, as you would for a [file-based certificate](#).

3.2 USB Security Key

3.2.1 What is a USB Security Key?

A USB Security Key is a hardware storage for a key file, that can be use as a token to access encrypted vault files. The key can be any type of USB key. At the root of the key, there will be two files: one is titled *"key_file.txt"* and is the file from which the encryption key is derived by hashing the contents of the file. The contents do not really matter as long as they are unpredictable and protected from unauthorized access. The second file is named *"security_key_id.txt"* and contains the identification

string of the hardware key. This should be a recognizable identification to distinguish this security key from others.

3.2.2 How to Generate a Key

Although the security keys can be manually created by creating the required files at the root of a USB key, Easy Authenticator provides an interface to easily create a key. The interface is invoked by the menu at *Tools / USB Security Key...*

In Figure 32, you can see an example where Easy Authenticator detected a USB Security Key (by validating the presences of the two files previously described) and has displayed the volume label and identification string of the key. The example uses a fingerprint protected USB key, which can help further secure the contents through biometric authentication.

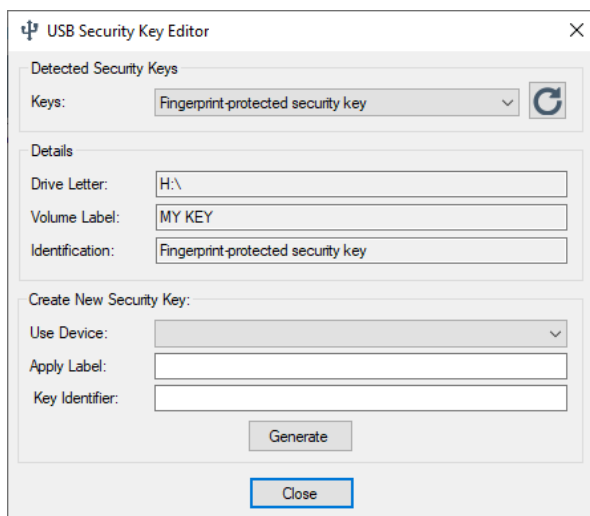




Figure 32: USB Security Key Editor Interface

The bottom part of the interface is used to easily create a new USB Security Key. The drop-down list will display all the removable drives connected to the system. To create a key, the drive must be selected, then optionally the volume label is defined, and the identification string is provided. When clicking *Generate*, the files will be created or replaced at the root of the key. The key file used for encryption will be randomly generated and written to the USB automatically, also overwriting any previously existing key file. No file other than these two specific ones is affected by the operation, and the USB key can contain other content, including vault files.

 **Warning:** If the key is already a USB Security Key, the files will be overwritten, and any vault secured with the key file that was previously on the USB key may become inaccessible.

 **Warning:** USB Keys can be lost, stolen, or break down. It is advisable to keep a copy of the key file in a secure location to be able to recreate the USB security key if needed.

3.2.3 How to Use a USB Security Key to Secure a Vault file

The USB Security Key, or more precisely the key file it contains can be used to derive the encryption key for a vault. The key can be compounded with other security controls for a more robust layer of protection. In the vault access interface, the available USB Security Keys will be detected and can be selected from the drop-down list. Easy Authenticator will use the key file on the selected USB key to generate the security parameter.

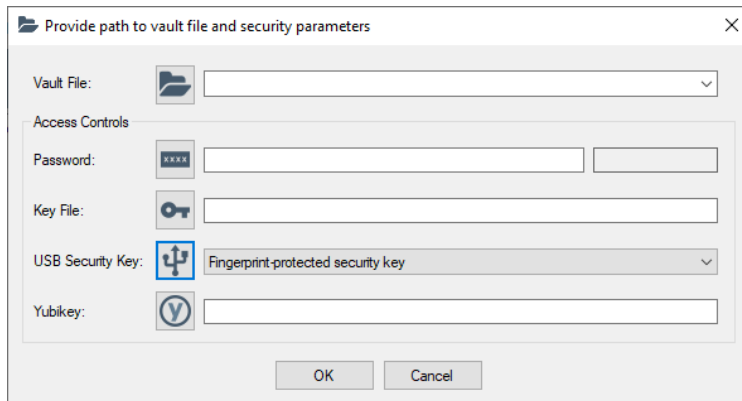


Figure 33: Selected USB Security Key for Vault Access

3.3 Certificate Viewer

As a utility function, Easy Authenticator allows the user to view the contents of a X509 certificate. This feature is invoked from the menu *Tools / Certificate Viewer...* To view the contents of a certificate, the user must provide the path, the password if one is required, and load the details. See the example in Figure 34.

The interface also provides access to certificates that are within a Yubikey. In this case, the user will need to click on the Yubikey icon, select the Yubikey connected to the system that contains the certificate, and select the slot where the certificate is located. The details of the certificate will then be displayed in the interface when the *Load* button is clicked.

Digital Certificate

Import Certificate

From File:

From Yubikey:

Password:

Certificate Details

Format: Version:

Serial Number:

Thumbprint:

Valid From: To:

Holds Private Key: Expires:

Issuer

Name:

Canonical:

Signature Algorithm:

Subject

Name:

Email:

Canonical:

Signature Algorithm:

Figure 34: Details of a Loaded Certificate