Bachelor thesis
Computing Science

Radboud University

ECDSA Based 2FA in the Browser

Author:
Mathieu Bangma
s1017020

First supervisor/assessor:
dr. ir. Bart Mennink
`b.mennink@cs.ru.nl`

Second assessor:
prof. dr. Eric Verheul
`e.verheul@cs.ru.nl`

January 8, 2021

Abstract

This thesis proposes a protocol that describes the interaction between a user and a service provider during authentication. The protocol relies on persistent storage of an elliptic curve private key in the Indexed Database API (IndexedDB), such that the private key cannot be extracted from the IndexedDB. Ownership of this private key can be proven using the elliptic curve digital signature algorithm (ECDSA), as a consequence the private key can be used as a factor during two-factor authentication (2FA). By providing this additional factor during authentication, we seek to overcome the insecurity of traditional text-based passwords. This insecurity is primarily caused by the tendency users have to reuse their passwords. We therefore seek to minimize the negative consequences of password reuse by providing an extra factor during authentication. The protocol relies on the IndexedDB to store the private key in a persistent and non-extractable manner. Implementing the protocol has shown that private key, stored in the IndexedDB, can be extracted from the IndexedDB. An alternative for the IndexedDB has to be found, since we conclude the security of the IndexedDB to be insufficient for persistent storage of a private key.

Contents

Chapter 1

Introduction

Strong authentication is needed in order to provide access to sensitive data. Financial services, health-care patient records, or online government are examples of use-cases that require strong authentication. At first single-factor authentication (SFA) was mostly adopted by the community due to its simplicity and user friendliness [25]. Common examples of SFA include the text-based password, and the PIN. However, passwords are the root cause of over 80% of data breaches [4]. One of the reasons for passwords being the root cause of data breaches is password reuse, meaning that users tend to use identical or nearly identical passwords at different service providers. Password reuse introduces a security vulnerability as an attacker who is able to compromise the password used at one service, can compromise other services protected by the same password, reducing overall security to that of the weakest site [13].

In order to raise the bar for an attacker, a logic step forward was the introduction of two-factor authentication (2FA). 2FA uses two distinct or identical factors to connect an individual with the established credentials. We identify three factors that can also be used as evidence for proving your identity [25]:

1. *Knowledge factor* — something the user knows,
   such as passwords or, simply, a "secret".

2. *Ownership factor* — something the user has,
   such as cards, smartphones, or other tokens.

3. *Biometric factor* — something the user is,
   such as biometric data or behavior patterns.

1.1 2FA Using the Browser

Even though 2FA raises the bar for an attacker, it is still questionable if 2FA can be realistically adopted by the majority of Internet users [26]. In this thesis we propose a protocol for 2FA that entirely runs in the browser. The idea is that the first knowledge factor is supplied by the user by using a regular username-password verification protocol. The second ownership factor is supplied by the user's browser by using our protocol. The protocol associates a private key with a specific browser, and thus with a specific user.

The second ownership factor is based on the ownership of a private key. Each private key is mathematically associated with a public key. The combination of a public and private key is called a key pair. Properly generating, storing, and proving ownership of a key pair is achieved in the browser using JavaScript. This means that no additional software, besides a browser, has to be installed in order to use our protocol. Generating a key pair, and subsequently proving ownership over this key pair, is done using the World Wide Web Consortium (W3C) recommended JavaScript interface Web Cryptography API. Proving ownership over the key pair, in particular the private key, is done by using the elliptic curve digital signature algorithm (ECDSA) in combination a challenge-response protocol. The ECDSA is used in accordance with the current recommendations made by the National Institute of Standards and Technology (NIST). The Web Cryptography API expects users to store their key material in the Indexed Database API 2.0 (IndexedDB), which is also recommended by the W3C. In this thesis we make the following contributions:

- We propose a protocol that full encapsulates the use of a second factor in the browser. The Web Cryptography API and Indexed Database API 2.0 will be used to achieve this.
- A proof-of-concept of our protocol will be given. The user side is implemented in JavaScript, while the service provider side is simulated using Python.
- A vulnerability assessment of our protocol is performed, which has identified the IndexedDB to be insufficiently secure to store a private key such that it can be used as an ownership factor in our protocol.
- We will make our own recommendations on the possibilities of deploying the protocol in the field.

1.2 Related Work

In this section we will discuss previous research on the risk of passwords reuse, and password managers that are designed to reduce this risk. Furthermore we will discuss previous research on current two-factor authentication methods.

1.2.1 Password Reuse

One of the pitfalls of user authentication with service providers is the text-based password. Users are estimated to have about 25 different online accounts, these accounts are likely to be protected by reused passwords [13]. To overcome this, and to prevent the use of easily guessable passwords, websites can put so-called password composition policies in place. These policies enforce the use of for example special characters, numbers, and capitalized letters in a password. However, cross-site password-guessing algorithms exist which are able to slightly transform passwords. Because of these transformations, these algorithms have up to 20-30% higher success rate in comparison to traditional cross-site password-guessing algorithms [13, 33].

1.2.2 Password Managers

Password managers are designed to overcome the need to reuse passwords by generating strong passwords that are resistant to dictionary attacks. These stronger passwords are generated uniquely for each service provider, and are protected by an easy to remember user-chosen password. Different types of password managers exist, stand-alone applications, browser plug-ins, browser scripts and bookmarklets. It has however been found that the use of password managers does not come without problems [7].

1.2.3 Two-Factor Authentication

Two-factor authentication was designed to offer enhanced protection during the authentication process. Adding an extra factor during authentication adds extra security to an account. There are numerous ways to achieve two-factor authentication, but still research has shown that some people find it difficult or time consuming to use and set-up 2FA [3, 27]. Also due to the extra steps a user has to complete in order to authenticate to an service provider, 2FA has become unpopular.

Currently the FIDO Alliance [4] is working on a simpler and stronger authentication process. One of the ways attempt to achieve this is using both embedded and external authenticators. Examples of embedded authenticators are biometrics and PIN codes, and examples of external authenticators include the FIDO Security Key (USB device that can perform public key cryptography), mobile devices and wearables.

1.2.4 2FA and the One-Time Password

The one-time password (OTP) can be used as 2FA in many different ways by a service provider to authenticate a user. The OTP was initially introduced by Leslie Lamport [20], and is based on the preimage resistance of hash functions. A well-known example of an OTP is SMS verification, in which a service provider sends the user an OTP to their phone using SMS. This OTP can then be used to authenticate the user. In 2011 an authentication protocol was presented which extends on the initial OTP as introduced by Lamport [14]. In theory, both presented protocols could be implemented in the browser using JavaScript, however no research is performed on how to properly store the secrets that are needed for both OTP protocols in the browser.

1.3 Outline

In Chapter 2 this thesis starts with an introduction on some cryptographic principles such as elliptic curves, security strength, digital signature algorithms and hash functions. This introduction is followed by a brief overview on the Web Cryptography API and the Indexed Database API 2.0, which are used in JavaScript. Thereafter in the Chapter 3 the protocol is defined and implemented. Subsequently, we investigate the underlying storage mechanism of the IndexedDB, and perform an application vulnerability assessment. Chapter 3 ends with a discussion on the limitations, and subsequently the recommendations we have identified for the our protocol. In Chapter 4 we draw our conclusions and address potential avenues for further research.

Chapter 2

Preliminaries

This chapter starts with the cryptographic background on which our protocol relies. This is done by an introduction on elliptic curves in Section 2.1, followed by a definition of security strength in Section 2.2. Hash functions and digital signature algorithms are introduced in Section 2.3 and in Section 2.4 respectively. The previously described sections come together to decide on which digital signature algorithm will be used in our protocol in Section 2.5. The two JavaScript libraries on which our protocol relies are introduced in Section 2.7 and in Section 2.8. Before we will discuss these libraries individually, we briefly describe the how modern websites use JavaScript in Section 2.6.

2.1 Elliptic Curves

Elliptic curve cryptography (ECC) was proposed independently by Neal Koblitz [19] and Victor Miller [23] in 1985. ECC is based on computations using points on an elliptic curve (EC). These computation are performed using different domain parameters, which are introduced below.

- p, denotes a large prime number that limits the number of points on the EC. The EC is defined over a finite field, denoted $\mathbb{F}_p$. A finite field can be seen as a set that contains a finite number of elements. Using these number we define how operations such as addition and multiplication should be performed. These operations are performed modulo p.

- a & b, two integers that define the shape of the EC, and example of an EC is the (short) Weierstrass form: $y^2 = x^3 + a + b$. Different values for a and b will change the shape of the EC, an example curve using $a = -3$ and $b = 3$ has been given in Figure 2.1.

- G, a base point on the EC that is also known as a generator of the subgroup. The subgroup contains a number of points on the EC.

- n, the smallest positive number such that $n \times G$ equals the point at infinity. n is also said to be the order of the group, or the number of elements in a specific group.
- h, also known as the cofactor. h is defined as $h = n/r$, where r is the order of the subgroup and n is the order of the curve. In the case that $n = r$, and thus $h = 1$, we have a subgroup of which the order is equal to the order of the curve. This means that points generated by the generator are equal to all the points on the curve.

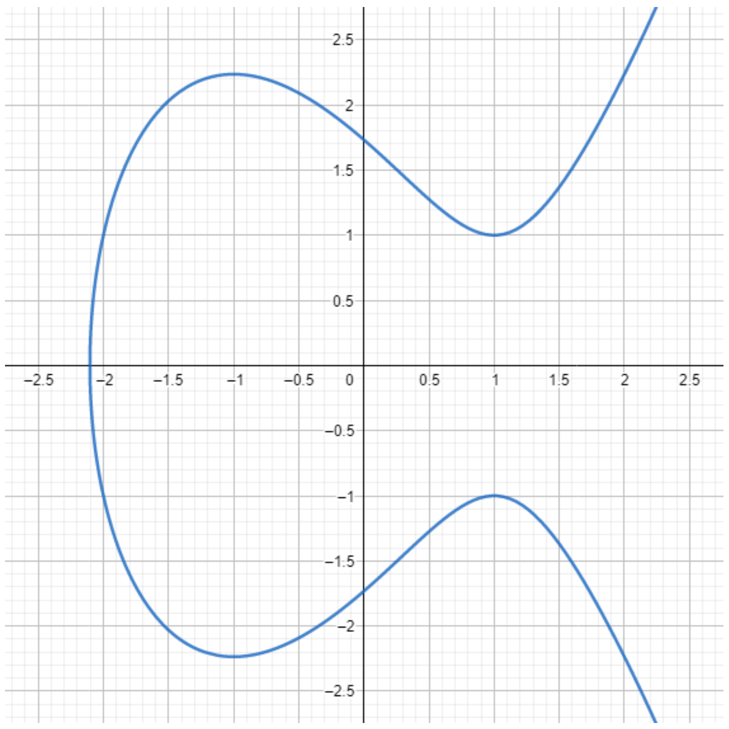


Figure 2.1: Part of the elliptic curve $y^2 = x^3 - 3x + 3$

2.1.1 Security of ECC

The security of elliptic curve based cryptographic algorithms is based on the elliptic curve discrete logarithm problem (ECDLP). ECDLP states that given an elliptic curve E defined over a finite field $\mathbb{F}_p$, a point P on E of order n, and a point Q that is a multiple of P such that $Q \neq P$, one has to find an integer $x \in [0, n-1]$ such that $Q = xP$ [22]. It is considered to be hard to find x given points P and Q. However, given x and P, it is trivial to compute Q.

2.1.2 Elliptic Curve Key Pair

An EC key pair consists of a private key pk, which is represented as an integer and a public key P which is a point on the curve. We derive the pubic key from the private key by computing $P = pkG$. Computing the private key, given points P and G is considered to be hard, because of the previously described ECDLP.

2.2 Security Strength

The security strength is expressed in bits, and represents a number that is associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system. Over time, cryptographic algorithms and their associated key lengths may become more vulnerable to successful attacks, requiring a transition to stronger algorithms or longer key lengths [5]. In other words, the higher the security strength, the stronger the cryptographic algorithm or system and the more likely it is to stay resilient against attacks in the future.

2.3 Hash Functions

A hash function is a function that takes a message of arbitrary length as an input and gives a fixed-length message as an output, the output is also known as the digest of the hash function. In other words, a hash function f is a function that takes a message x, and gives a digest y. The hash function f is said to be deterministic, that is computing $f(x)$ multiple times will always result in the same digest. It is therefore also said that a hash function is a mapping from a message to a digest. Different types of hash functions exist, their difference being most notable in their security strength and the length of their digest. We identify three different attacks, and thus three different attacks to which a hash function can be resistant to. Their definitions as presented below are in accordance with the NIST definitions [31].

1. Preimage resistance.
   Preimage resistance is an expected property of a cryptographic hash function such that, given a randomly chosen message digest, *message_digest*, it is computationally infeasible to find a preimage of the *message_digest*. In other words, given y it is difficult to find an x such that $f(x) = y$.

2. Second preimage resistance.
   Second preimage resistance is an expected property of a cryptographic hash function whereby it is computationally infeasible to find a second preimage of a known message digest. In other words, given x it is difficult to find an x' such that $x' \neq x$ and $f(x) = f(x')$.

3. Collision resistance.
   An expected property of a cryptographic hash function whereby it is computationally infeasible to find a collision. In other words, it is difficult to find both x and x' such that $x' \neq x$ and $f(x) = f(x')$.

2.4 Digital Signature Algorithms

This thesis relies on a technique that is able to provide proof of ownership of a private key. This proof can be provided by using a digital signature algorithm in combination with a challenge. How a digital signature algorithm is incorporated in our protocol can be found in Chapter 3. In this section we will focus on two digital signature algorithms that are standardized by NIST [30]. We will do so by giving a general overview of the two digital signature algorithms, and demonstrate the intuition on how these algorithms operate in Section 2.4.1 and Section 2.4.2. In Section 2.4.3 and Section 2.4.4 the two digital signature algorithm will be introduced, after which we discuss which of the two algorithms is best suited for our protocol in Section 2.4.5.

2.4.1 Introduction

We continue to discuss two different signature verification algorithms that are approved by NIST [30]. These two digital signature algorithms rely on asymmetric cryptography. That is, the user generating a signature has ownership over a private key, and an associated public key. The combination of the public and private key is called a key pair. Each public key is associated with exactly one private key and vice versa. As the names suggest, the private key is to be kept private at all times, while the public key is meant to be shared. When using a digital signature algorithm, a signature is computed over a message and a private key. By doing so, one can provide evidence that it owns its own private key corresponding to its public key. It can therefore be used as a factor. However, one cannot simply send its own private key. This is because we seek to link the ownership of a private key with the identity of a user. Simply "showing" your private key makes it very vulnerable to be stolen. We will therefore use a digital signature algorithm.

Both signature generation algorithms consist of two phases:

1. Signature generation, in which a digital signature is computed over a message and a private key.
2. Signature verification, in which it is determined whether the combination of signature, public key and message is valid. In other words, it is verified whether the signature is correctly computed using both the message and the private key that is associated with the expected public key.

These two phases are graphically represented in Figure 2.2. The result of the signature verification will be used to determine whether the user actually owns the correct private key, without ever revealing said key.

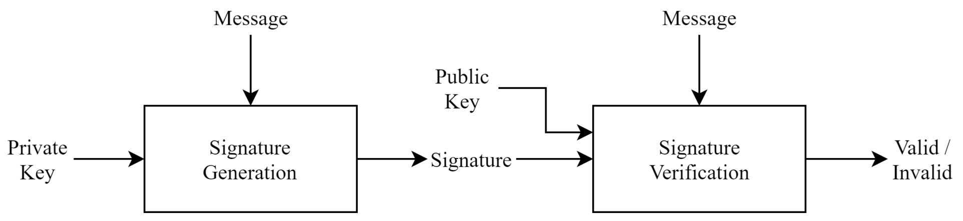


Figure 2.2: Signature generation and verification.

2.4.2 Hash Function

In Figure 2.2 we saw how the message is directly supplied to both the signature generation and verification functions. In reality we first need to compute the hash of the message, and provide the digest to both the signature generation and verification functions. Thus, we generate the signature over the digest and also verify the signature using the digest of the same message. An updated illustration that represents this concept is given in Figure 2.3. The reason of adding this extra layer of complexity is mainly due to an attack called digital signature forgery which will be discussed in the next paragraph.

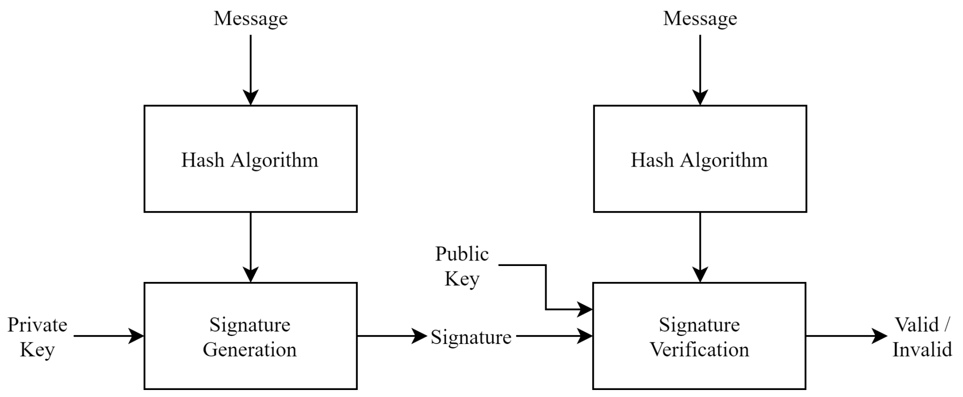


Figure 2.3: Signature generation and verification using a hash function.

Digital Signature Forgery

A hash function is able to transform a message of variable length into a digest of fixed length. In order to understand the importance of this transformation, we need to have a closer look at how digital signature algorithms interpret the digest of a message during signature generation. Digital signature algorithms do not accept a variable length message, instead they will only look at the n left most bits of the provided digest. Not applying a secure hash function before generating a signature will result in an attack being able to perform a digital signature forgery, and thus compromising the overall security of the used digital signature algorithm. A digital signature forgery is the ability to generate a signature for a message, without knowing the private key and thus without using the regular signature generation function. A proof that we can indeed create a forgery knowing only the message and corresponding signature, given that there is no hash function in place, has been in given in Figure B.1.

Digital signature forgery is the reason to use a hash function before signature generation. In the case that a hash function is present, it has to provide collision resistance in order to prevent the ability of creating a digital signature forgery as described in the previous paragraph. Otherwise, two different messages that result in the same digest will also have the same signature and thus it becomes feasible to create a digital signature forgery.

Besides collision resistance a hash function should also offer second preimage resistance. This property is of importance for the same reason as why collision resistance is important. It should be noted that when a hash function is collision resistant, it implies that this hash function is also second preimage resistant.

2.4.3 RSA Digital Signature Algorithm

RSA is an acronym of the surnames of Ron Rivest, Adi Shamir and Leonard Adleman, who first described RSA in 1977 [28]. RSA is based on the so-called prime factorisation problem, which states that given two large prime numbers p and q it is hard to derive both p and q from n such that $n = p \cdot q$.

Key Pair Generation

We present the following steps to generate a RSA key pair, some details in these steps are omitted to make the example clearer. The key pair can then used to generate and subsequently verify signatures using RSA:

1. Generate two distinct prime integers p and q.
2. Compute n such that $n = p \cdot q$.
3. Compute Carmichael's totient function for n, $\lambda(n)$. Since $n = p \cdot q$ we can rewrite $\lambda(n)$ in terms of the least common multiple of p and q, so $\lambda(n) = \text{lcm}(\lambda(p), \lambda(q))$. Because both p and q are prime we can further derive this expression to $\lambda(n) = \text{lcm}(p-1, q-1)$.

4. Choose an integer e such that $1 < e < \lambda(n)$. This e has to be coprime to $\lambda(n)$. Which means that the greatest common divisor of both e and $\lambda(n)$ has to be 1, thus $\gcd(e, \lambda(n)) = 1$.
5. Compute d, such that $d \equiv e^{-1} \pmod{\lambda(n)}$.

After completing these steps we have generated a public key which consists of the tuple (n, e) and is corresponding private key which consists of the tuple (d, e).

Signature Generation & Verification

Using the generated key pair from the previous section, we can now generate the signature of a message m. In order to generate a signature we first compute the digest h of m using a hash function H, so $h = H(m)$. The signature s can then be computed using the private key such that $s = h^d \pmod{n}$. Subsequently verifying this signature can be done using the corresponding public key, so we verify that $s^e \equiv h \pmod{n}$. These two will be equal if the correct private and public key pair are used, because $s^e = (h^d)^e = h^{de} \equiv h \pmod{n}$.

2.4.4 Elliptic Curve Digital Signature Algorithm

Having discussed the basic principles and key pair generation of EC in Section 2.1, we describe the signature generation and verification of the elliptic curve digital signature algorithm (ECDSA) in this section.

Signature Generation & Verification

The domain parameters used in this section are introduced in Section 2.1. In order to generate a signature we take the following steps:

1. First compute the digest h of m using a hash function H, so $h = H(m)$.
2. Subsequently generate a random k such that $1 < k < (n - 1)$, where n is the order of G.
3. Compute $(x, y) = k \times G$.
4. Compute $r = x \pmod{n}$. If r is equal to zero, we go back to step 2 and select a new k.
5. Compute $s = k^{-1}(h + r \cdot pk) \pmod{n}$. If s is equal to zero, we go back to step 2 and select a new k.

We now have the signature, which is the tuple (r, s). Subsequently verifying the generated signature can be done by taking the following steps:

1. Compute the digest h of m using the same hash function H, so $h = H(m)$.
2. Compute $(x, y) = (h \cdot s^{-1}) \cdot G + (r \cdot s^{-1}) \cdot P$.
3. The signature is valid if r equals x, and invalid otherwise.

2.4.5 Discussion

We decided to use ECDSA in our protocol at the expense of RSA, because for the same key length ECDSA achieves a higher security strength compared to RSA [5]. Besides a shorter key length which will reduce storage overhead on storage limited devices, the length of a signature generated using ECDSA is shorter compared to the length of a signature generated using RSA for the same security strength. The advantage of shorter signatures is that less data has to be send over the network.

2.5 Selection of Elliptic Curve and Hash Function

In this section we discuss which hash function should be used in which situation. Our protocol uses the Web Cryptography API, this API will among other things be used to generate signatures. The ECDSA requires us to specify both an elliptic curve and a hash function in order to generate signatures. The Web Cryptography API supports the following elliptic curves: P-256, P-384, and P-521. Furthermore, the Web Cryptography API supports the following four hash functions: SHA-1, SHA-256, SHA-384, and SHA-512. The choice of hash function largely depends on the security strength of the hash function in relation to the security strength of the chosen elliptic curve. In this section we will discuss the security strength of both the supported elliptic curves and the supported hash functions individually, followed by a discussion on which elliptic curve and hash function will be used in our protocol.

2.5.1 Elliptic Curve Security Strength

The security strength of the available curves has been standardized by NIST [32], and is defined to be about $n/2$. In Section 2.1 n was introduced as the order of the base point G. The curves P-256, P-384, and P-521 have an order of 256, 384 and 521 bits respectively, hence their naming. In accordance with the NIST specification, curve P-256 has a security strength of 128 bits, P-384 has a security strength of 192 bits, and P-521 has a security strength of 256 bits [5].

2.5.2 Hash Function Security Strength

The security strength of a hash function is determined depending on its required properties. In Section 2.4.2 we have stated that the hash function used in our protocol has to be both collision resistant and second preimage resistant. A hash function that is collision resistant implies that this function is also second preimage resistant. Thus, the security strength of our hash function depends on the expected collision resistance of said function. The expected collision resistance strength of a hash function is half the length of the digest, i.e., for a L-bit hash function, the expected security strength for collision resistance is about $L/2$ bits [12].

2.5.3 Discussion

Now that we have defined the basis of the expected security strength of the available elliptic curves and hash functions, we can determine the actual expected security strength. This is represented in Table 2.1.

Security Strength	Hash Function used for Digital Signatures	ECDSA Curve
≤ 80	SHA-1	
128	SHA-256	P-256
192	SHA-384	P-384
256	SHA-512	P-521

Table 2.1: Security strength of the supported hash functions and curves.

In our protocol we decided to use the SHA-512 hash function in combination with the P-521 curve. The main reason for this choice is future use. Our protocol relies on storing a private key. Since this key will be used during future login attempts, we wish our protocol to not only be secure right now, but also in the foreseeable future. NIST recommends that a security strength of 128 bits is acceptable for the year 2031 and beyond [5]. However, the higher the security strength of the cryptographic algorithms our protocol relies on, the longer our protocol is expected to remain secure. We therefore opted for the highest security strength that is available to us, which is the SHA-512 hash function in combination with the P-521 elliptic curve.

2.6 JavaScript

In Section 2.7 and Section 2.8 the two required JavaScript interfaces will be introduced. JavaScript is a programming language that allows a web-page

to have additional functionalities beyond what is possible with HTML and CSS. HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) are two typically used languages to define the appearances of a website. Typically the JavaScript is defined in a file with the `.js` suffix, where HTML and CSS typically have the `.html` and `.css` suffix respectively. These three files combined form a website.
We will first introduce the Web Cryptography API, on which the our protocol relies for ECDSA signature generation and EC key pair generation. Our protocol uses the Indexed Database API to handle the storage of the generated key pair.

2.7 Web Cryptography API

In order to perform ECDSA in the browser, we need to find a reliable implementation that is able to perform two operations. Firstly it has to be able to generate EC key pairs, secondly it should also be able to perform the previously discussed ECDSA signature generation and verification. These two operations should furthermore be fully supported in JavaScript. Such an implementation is provided by the Web Cryptography API, which has been recommended by the W3C since January 2017 [9]. A consequence of this recommendation is that the Web Cryptography API is currently compatible with all major browsers [10]. In this section we will provide coding examples on how to generate an EC key pair, and how to perform ECDSA signature generation and verification using the generated key pair.

2.7.1 Generating a Key Pair

Using the Web Cryptography API we can generate an EC key pair using the function `generateKey`, which has three arguments. First it takes an `EcKeyGenParams` object which specifies the type of the key pair. The type of a key pair consists of a name, in our case `ECDSA`, and the name of the elliptic curve to use which in our case is `P-521`. The second argument of the function `generateKey` is a boolean, which determines whether or not the private key is extractable. That is, can the private key ever be revealed from the key pair, or can it only be used for certain computations. Which kind of computations are allowed to use the key pair is determined by the third argument of the function `generateKey`. The argument takes a String array which contains either or both the strings `sign` and `verify`, which indicate that the key pair can be used to both perform signature generation and signature verification respectively. In Figure 2.4 a key pair is generated that is able to both sign and verify signatures.

```javascript
window.crypto.subtle.generateKey(
    {
        name: "ECDSA",
        namedCurve: "P-521",
    },
    extractable,
    ["sign", "verify"]
)
.then(keyPair => {
    // Handle the generated key pair
});
```

Figure 2.4: Generating an ECDSA key pair.

2.7.2 Proving Ownership

In Figure 2.4 we have seen that we can generate a key pair using the Web Cryptography API. Using the same API we can also sign a message and subsequently verify the signature. Signature generation can be performed by executing the `sign` function which, as expected, takes a private key and a message as its arguments. Besides these arguments it also takes an `EcdsaParams` object. In this object we specify the name of the digital signature algorithm, in our case `ECDSA`, and the hash to be used, which is `SHA-512` in our case. A coding example of ECDSA signature generation signature is given in Figure 2.5.

```javascript
window.crypto.subtle.sign(
    {
        name: "ECDSA",
        hash: {name: "SHA-512"},
    },
    privateKey,
    message
)
.then(signature => {
    // Handle the signature
});
```

Figure 2.5: Generating an ECDSA signature.

After signing a message, we can determine if the generated signature is valid. This can be done by executing the function `verify`, which also takes an `EcdsaParams` object as an argument. The same name and hash are to be used during signature generation. Furthermore the function `verify` takes a public key, signature and message and will return a boolean. This boolean tells us whether or not the signature is valid. A coding example is given in Figure 2.6.

```
    window.crypto.subtle.verify(
        {
            name: "ECDSA",
            hash: {name: "SHA-512"},
        },
        publicKey,
        signature,
        message
    )
    .then(isValid => {
        // Handle isValid boolean
    });
```

Figure 2.6: Verifying an ECDSA signature.

2.8 Indexed Database API 2.0

The Indexed Database API, or IndexedDB, has been standardized by the World Wide Web Consortium (W3C) since January 8th, 2015. It has been designed to be a database of records holding simple values and hierarchical objects. Each record consists of a key and some value. Moreover, the database maintains indexes over records it stores [8]. The IndexedDB is a persistent databases, meaning that whatever is stored using the API will be stored for future use. In this section different aspects of the documentation will be highlighted. These aspects are required as basic background knowledge for this thesis.

2.8.1 Database Object

A request has to be made in order to open or create a database. This request takes a name variable, and optionally a version number which can be almost any integer greater than one. The version number is set to 1 in the case it has been left unspecified. In Figure 2.7 the boilerplate code for opening and creating a database object can be found. One or more of the four open requests, `onupgradeneeded`, `onsuccess`, `onerror` or `onblocked` may be called to indicate the progress. These four events may be triggered when both opening or deleting a database.

```javascript
const request = indexedDB.open(name, number);

request.onupgradeneeded = event => {
    // Handle on Upgrade Needed
}
request.onsuccess = event => {
    var db = event.target.result;
    // Handle on Success
}
request.onerror = event => {
    // Handle on Error
}
request.onblocked = event => {
    // Handle on blocked
}
```

Figure 2.7: Opening/creating a database object.

If the `onsuccess` function is called, a so-called “connection” has been made with the database. The connection will be closed whenever the user leaves the site, or the variable `db` goes out of scope. Each connection contains a set of object stores, which in turn are the holders of the actual data. This concept will be substantiated in the subsequent section.

2.8.2 Object Store

As briefly introduced in the previous section, an object store is the primary storage mechanism for storing data. An object store has a list of records which hold the data stored in the object store. Each record consists of a key and a value. The list is sorted according to the key in ascending order. There can never be multiple records in a given object store with the same key [8]. No Structured Query Language (SQL) is required in order to interact with the IndexedDB, since the database contains objects that are indexed by key value pairs.

2.8.3 Transaction

Figure 2.8 displays how the connection with the database can be used to create a transaction and subsequently an object store. A transaction is needed whenever we want to read or write to the database. It takes both a scope and a mode as arguments. Using the scope argument we determine which object store(s) this transaction can use, while the mode determines the type of interactions with these object store(s). The mode can either be `readonly`, `readwrite` or `versionchange`.

```
    var transaction = db.transaction(scope, mode);
    var objectStore = transaction.objectStore(name);
```

Figure 2.8: Creating a transaction for an object store.

2.8.4 Storing and Retrieving a Key Pair

Once we have actually create a transaction, we can continue with storing the key pair we have created in Figure 2.4. In this example we will store the generated key pair `keyPair` from Figure 2.4, using the variable `objectStore` used in Figure 2.8. In order to do so we need the `readwrite` permission as a mode of the transaction. A coding example of this procedure is given in Figure 2.9. As can be seen in this example, an optional key could be provided whenever we wish to store a JavaScript value in the IndexedDB. This key can be used to identify the value, or in this case the keyPair, we are adding to the database.

```
    var request = objectStore.add(keyPair, key);
    request.onsucces = event => {
        // Handle on success
    }
    request.onerror = event => {
        // Handle on error
    }
```

Figure 2.9: Adding a key pair to IndexedDB.

Retrieving an object from the IndexedDB can be done in a similar fashion as adding an object to the IndexedDB. For example, the key pair we have added to the database in Figure 2.9 can be retrieved by calling the function `get` on the variable `objectStore`. This has been demonstrated in Figure 2.10. While doing so we need to ensure that the transaction we are using transaction either has the mode `readonly` or `readwrite`.

```
    var request = objectStore.get(key);
    request.onsucces = event => {
        // Handle on success
    }
    request.onerror = event => {
        // Handle on error
    }
```

Figure 2.10: Extracting a key pair to IndexedDB.

2.8.5 Security Considerations

In the previous sections we have given a brief overview of how the IndexedDB can be used in combination with the Web Cryptography API. In this section we will discuss some of the security considerations that are implemented by the IndexedDB to ensure that, in our case, our key material originating from the Web Cryptography API stays secure. All of these considerations attempt to restrict access such that only the scripts that are supposed to access specific databases can do so.

Same Origin Policy

The same origin policy (SOP) ensures that only certain origins can access databases that are assigned to that specific origin. An origin is defined as the combination of the application layer protocol, domain and port number [21]. Each origin has its own corresponding set of databases, and thus the SOP prevents an origin access to the database(s) of another origin.

DNS Spoofing

The domain name system (DNS) is a system which maps domain names to an IP-address, which is used to identify and locate servers on the internet. DNS spoofing is the act of maliciously modifying this mapping from domain name to IP-address. This can be done in various ways, however these techniques will not be discussed in this thesis. It is important to run our protocol over TLS (HTTPS) instead of HTTP. HTTPS is a more secure version of HTTP which are both used to communicate between a server and a client over the Internet.

Chapter 3

Proposed Protocol

In this chapter we propose our protocol. We will do so in Section 3.1, and give a minimal implementation in Section 3.2. Based on this implementation, we investigate how the IndexedDB implementation of the Google Chrome browser stores JavaScript objects on the computer in Section 3.3. An assessment is made to identify any possible vulnerabilities in Section 3.4. Based on our findings we described the limitations and our recommendations of our protocol in Section 3.5 and Section 3.6 respectively.

3.1 The Protocol

In this section we will present the theoretical framework of using the browser as an ownership factor during authentication. The protocol describes the interaction between a user and the service provider during authentication, and relies on the Web Cryptography API and the Indexed Database API in the browser on the client side. We define a protocol on how to use the IndexedDB as an ownership factor during authentication. The protocol can be found in Protocol 1.

1. A service provider asks a user to authenticate.
2. The user and the service provider execute the regular username-password verification protocol.
3. If the browser does not have a key pair, it will generate one.
4. The service provider sends a challenge C to the user.
5. The user computes an ECDSA signature S, based on C, its password, and its private key located in the IndexedDB.
6. The user sends S and its public key to the service provider.
7. The service provider performs ECDSA signature verification if the public key is known.
8. If one of the steps 2-7 fails, the protocol fails and otherwise it succeeds.

Protocol 1: Using the browser as an ownership factor.

We elaborate on the steps of Protocol 1.

1. *"A service provider asks a user to authenticate."*
   The request to authenticate marks the start of this protocol, such a request be made whenever the service provider deems authentication of the user to be necessary.

2. *"The user and the service provider execute the regular username-password verification protocol."*
   During this step a protocol that authenticates the user based on a username and password combination is used. Said protocol is simplified down to a single step. More information on this step can be found in Section 3.1.2.

3. *"If the browser does not have a key pair, it will generate one."*
   This steps marks the first new additional step compared to single factor authentication. In the case that the specific browser has never been used before to authenticate to the service provider, it has to generate a key pair which it will store in the IndexedDB.

4. *"The service provider sends a challenge C to the user."*
   It is critical that the challenge C is unpredictable, in order to prevent possible replay-attacks by malicious users.

5. *"The user computes an ECDSA signature S, based on C, its password, and its private key located in the IndexedDB."*
   This step is performed in the browser using the Web Cryptography API, based on the key pair that is stored in the IndexedDB. The message that is to be signed is equal to the password appended to the challenge, which is provided by the service provider.

6. *"The user sends S and its public key to the service provider."*
   The signature S is sent to the service provider, so that it can be used in the subsequent step. By always sending the public key along with the signature we prevent a malicious user from gaining knowledge on whether or not the user is able to authenticate from multiple browsers. Additional information on authenticating with multiple browsers can be found in Section 3.1.3.

7. *"The service provider performs ECDSA signature verification if the public key is known."*
   The penultimate step of the protocol. During this step the service provider performs ECDSA signature verification, and thus checks if the user has ownership over the correct private key. If the service provider does not recognize the public key as a valid public key for this user, then this step will fail.

8. *"If one of the steps 2-7 fails, the protocol fails and otherwise it succeeds."*
   This final step determines whether or not the authentication process was successful. It should be noted that even if step 2 fails, that is when an unknown username-password combination is provided to the service provider, steps 3 till 7 are still executed. This has be done to prevent a malicious user from obtaining information on whether step 2 or 7 has failed. In other words, using this protocol, a malicious user cannot differentiate between an incorrect username-password combination and a failed ECDSA signature verification. The user is notified of the result of the protocol.

After a successful authentication, the service provider may give the user the ability to authenticate using another browser, besides the browser that was just used to authenticate with. The support of multiple browsers is described in Section 3.1.3.

3.1.1 Network Diagram

Before we will give an implementation of our protocol, we will discuss the network interactions of the protocol. The network interactions are described between a user and a service provider to which the user has to authenticate itself to. The order and effects of these interactions highlight the basic innerworking of our protocol. In Figure 3.1 we give the network diagram of our protocol that is described in the previous section.

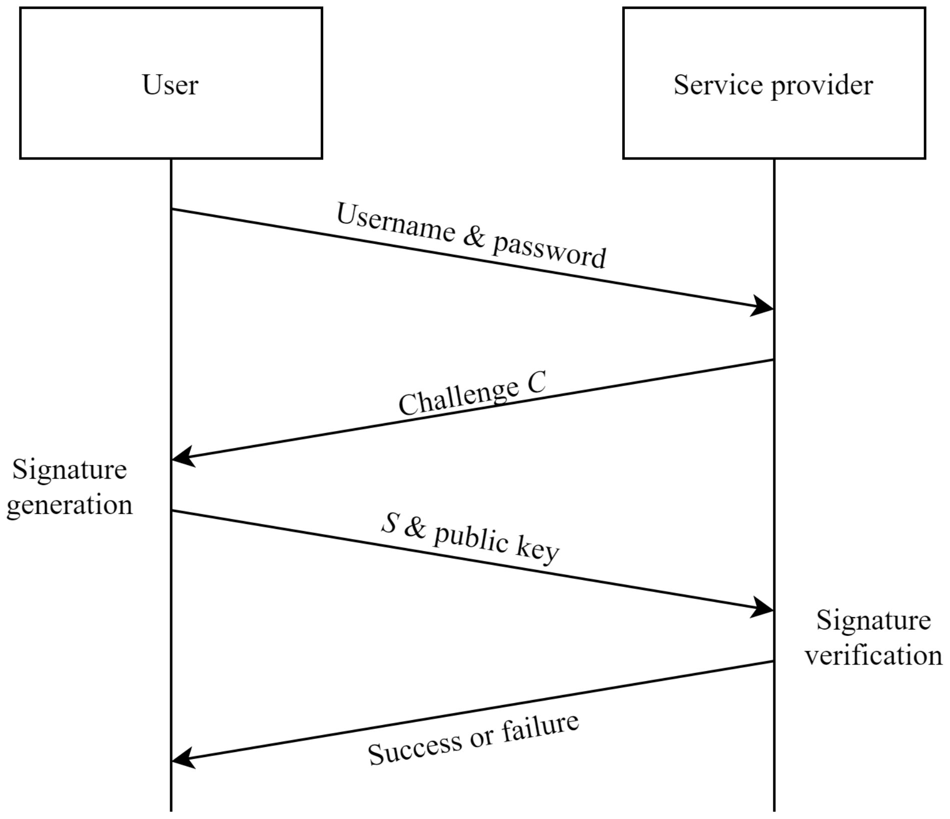


Figure 3.1: Network diagram of our protocol.

3.1.2 Authentication Protocol

The network diagram in Figure 3.1 starts with the user sending a request to log in. It does so by providing a username and password combination. However, the network diagram represents authentication using a username and password in a rather naive way. It does so by using the password authentication protocol (PAP), which is the most basic authentication protocol one can imagine, the user sends a username and password combination to the service provider which on its turn responds whether or not this combination is correct. PAP should not be used, since it is very insecure to not encrypt your credentials before sending them over the network. Furthermore PAP does not offer any security against replay attacks. In the rare instance that PAP is used, our protocol should append the text-based password to the challenge before it is signed.

Challenge-Handshake Authentication Protocol

A more robust authentication protocol for a username and password combination is the challenge-handshake authentication protocol (CHAP). Whenever CHAP is used, the service provider sends an unpredictable challenge to the user, which has to hash this challenge together with the password. The digest of this hash computation is sent back to the service provider, which performs the same hash computation and compares the digest with the result that is provided by the user. The user is authenticated only if both digests are equivalent. The service provider may send such challenge at random intervals, and not only during login. Whenever a variant of CHAP is used in combination with our protocol, the user should append the digest to the challenge before it is signed, instead of appending the password to the challenge.

3.1.3 Multiple Browser Support

Some users might find it frustrating when they are only able to authenticate using one specific browser. For instance, one might want to authenticate to a service provider from both a personal computer and a mobile phone. We therefore define the option to authenticate from a limited number of browsers. Each browser that a user wishes to authenticate from has a unique key pair. As a consequence, the service provider has to keep track of which public keys could be used to verify the signature of which specific user. In Protocol 2 we define a protocol that should be used whenever a user wants to authenticate from a new browser, but already uses another known browser. In this protocol we use the following terminology:

- *New browser*, a browser that has never successfully completed the protocol as defined in Protocol 1. However, the user wants to use this specific browser as defined in Protocol 1.

- *Known browser*, a browser that has successfully completed the protocol as defined in Protocol 1. This browser thus has a key pair that can be used to authenticate with.

1. The user will notify the service provider, using a known browser, that it wants to authenticate using a new browser.
2. The user performs Protocol 1 using a new browser.
3. The user has the option to trust the new browser, using a known browser.

Protocol 2: Using an additional browser as an ownership factor.

We will elaborate on the steps of Protocol 2.

1. *"The user will notify the service provider, using a known browser, that it wants to authenticate using a new browser."*
   After successful authentication using a known browser, the user is given the option to inform the service provider that the user wishes to authenticate with a new browser. Note that the user only has the option to do so if Protocol 1 has been performed successfully.

2. *"The user performs Protocol 1 using a new browser."*
   Using a new browser Protocol 1 is performed. By doing so the new browser will generate a key pair and store it in the IndexedDB, however Protocol 1 will fail on step 7 because the newly generated public key is not known by the service provider. However, if the previous step was performed successfully, the service provider will temporarily store the public key whilst awaiting approval in the subsequent step.

3. *"The user has the option to trust the new browser, using a known browser."*
   Trusting the new browser will tell the service provider that the public key, that was supplied to the service provider in the previous step, can be used to perform ECDSA signature verification with. Protocol 2 is only successful if the user decides to trust the new browser in this final step.

After completing Protocol 2, the user is able to execute Protocol 1 successfully using the new browser.

3.2 Implementation

In Figure A.1 till Figure A.4 a minimal working proof-of-concept is given. The already introduced Web Cryptography API (Section 2.7) and IndexedDB (Section 2.8) are used to execute our protocol in the user's browser. On the other hand, the server side of our protocol relies on the use of two Python libraries called Flask [29] and Cryptography [11]. Flask is used to start a local server, which is able to simulate the service provider side of the network diagram given in Figure 3.1. The Cryptography library is used to perform the ECDSA signature verification on the signature and public key, and the corresponding challenge as provided according to our protocol. A `.html` file and `.css` file can also be found in these figures, these two files represent a generic login window. These two libraries, and their corresponding `.html` and`.css` files, will not be discussed in detail, since their use is merely to provide a proof-of-concept and not to provide a production ready codebase.

3.3 IndexedDB Files

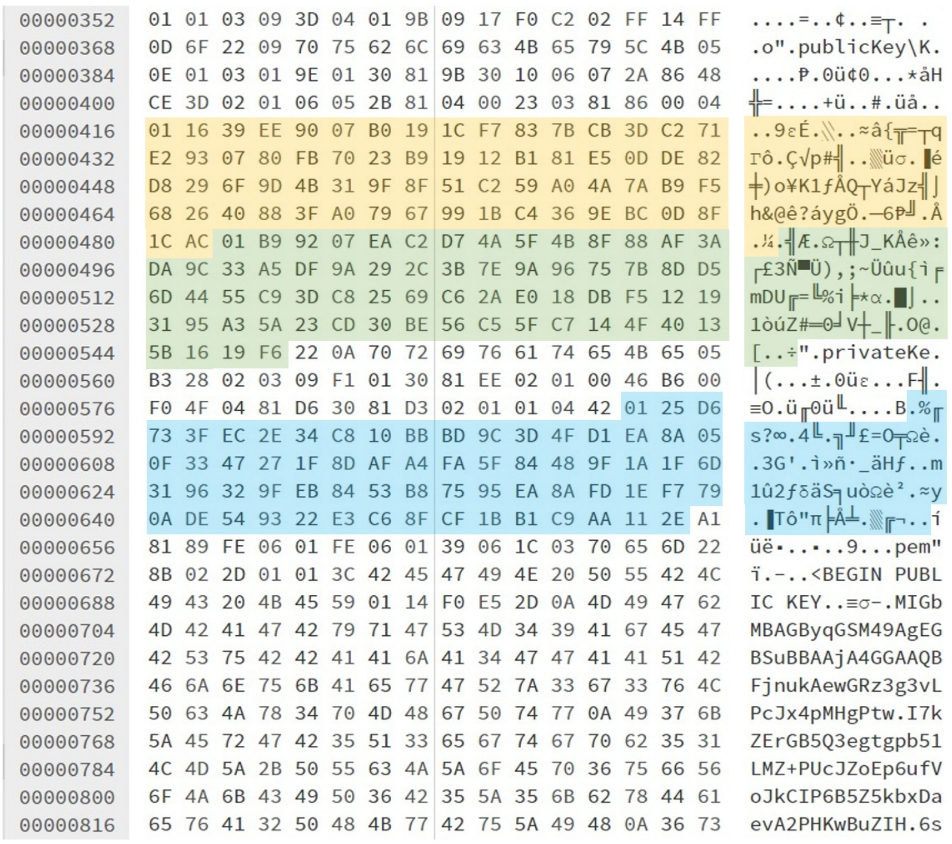
The security of Protocol 1 is based on the assumption that the private key, which is stored in the IndexedDB, can be used to uniquely identify a browser. This identification can only occur if there is no way to extract or reveal the private key after it has been generated. In the JavaScript environment this property is guaranteed by the Web Cryptography API, which allows the programmer to set the `extractable` Boolean to false upon key pair generation, which as a consequence means that the private key cannot be revealed in the JavaScript environment. In order to further investigate the strength of the connection between the private key and the browser, we abstract away from the JavaScript environment and investigate the IndexedDB on a lower level. We investigate how the IndexedDB stores the key pair on the drive for persistent storage. Even though the use of the IndexedDB is standardized by W3C, the underlying structure is not. For example Google Chrome uses the LevelDB [16], while Mozilla Firefox uses SQLite [24]. In this section we will investigate how Google Chrome stores a key pair, using the LevelDB.

3.3.1 LevelDB

Once we request to store a key pair using JavaScript, a `.ldb` file is created in which the key pair is stored. The process of converting a JavaScript object to bytes, such that they can be stored in a `.ldb` file is called serialization. The process of converting these bytes back into the same JavaScript object is called deserialization. During serialization both the public and private keys are not compressed nor encrypted, they can thus be found in clear text in the `.ldb` file. The private key seems to always follow the hex string `0xD63081D30201010442`. In Appendix A.5 a proof-of-concept is given using Python that can, if provided a `.ldb`, extract the private key.

Example

Using the implementation as described in Section 3.2, we generate a key pair which we store in the IndexedDB using the Google Chrome (version 87.0.4280.88) browser. The key pair is stored as a JavaScript object that has three properties, the public key, the private key, and the public key formatted in PEM format as formalized in RFC 7468 [17]. The subsequently generated `.ldb` file is inspected, and shown in Figure 3.2.



```
00000352  01 01 03 09 3D 04 01 9B 09 17 F0 C2 02 FF 14 FF  ....=..¢..≡┬. .
00000368  0D 6F 22 09 70 75 62 6C 69 63 4B 65 79 5C 4B 05  .o".publicKey\K.
00000384  0E 01 03 01 9E 01 30 81 9B 30 10 06 07 2A 86 48  ....₧.0ü¢0...*åH
00000400  CE 3D 02 01 06 05 2B 81 04 00 23 03 81 86 00 04  ╬=....+ü..#.üå..
00000416  01 16 39 EE 90 07 B0 19 1C F7 83 7B CB 3D C2 71  ..9εÉ.░..≈â{╦=┬q
00000432  E2 93 07 80 FB 70 23 B9 19 12 B1 81 E5 0D DE 82  Γô.Ç√p#╣..▒üσ.▐é
00000448  D8 29 6F 9D 4B 31 9F 8F 51 C2 59 A0 4A 7A B9 F5  ╪)o¥K1ƒÅQ┬YáJz╣⌡
00000464  68 26 40 88 3F A0 79 67 99 1B C4 36 9E BC 0D 8F  h&@ê?áygÖ.─6₧╝.Å
00000480  1C AC 01 B9 92 07 EA C2 D7 4A 5F 4B 8F 88 AF 3A  .¼.╣Æ.Ω┬╫J_KÅê»:
00000496  DA 9C 33 A5 DF 9A 29 2C 3B 7E 9A 96 75 7B 8D D5  ┌£3Ñ▀Ü),;~Üûu{ì╒
00000512  6D 44 55 C9 3D C8 25 69 C6 2A E0 18 DB F5 12 19  mDU╔=╚%i╞*α.█⌡..
00000528  31 95 A3 5A 23 CD 30 BE 56 C5 5F C7 14 4F 40 13  1òúZ#═0╛V┼_╟.O@.
00000544  5B 16 19 F6 22 0A 70 72 69 76 61 74 65 4B 65 05  [..÷".privateKe.
00000560  B3 28 02 03 09 F1 01 30 81 EE 02 01 00 46 B6 00  │(...±.0üε...F╢.
00000576  F0 4F 04 81 D6 30 81 D3 02 01 01 04 42 01 25 D6  ≡O.ü╓0ü╙....B.%╓
00000592  73 3F EC 2E 34 C8 10 BB BD 9C 3D 4F D1 EA 8A 05  s?∞.4╚.╗╜£=O╤Ωè.
00000608  0F 33 47 27 1F 8D AF A4 FA 5F 84 48 9F 1A 1F 6D  .3G'.ì»ñ·_äHƒ..m
00000624  31 96 32 9F EB 84 53 B8 75 95 EA 8A FD 1E F7 79  1û2ƒδäS╕uòΩè².≈y
00000640  0A DE 54 93 22 E3 C6 8F CF 1B B1 C9 AA 11 2E A1  .▐Tô"π╞Å╧.▒╔¬..í
00000656  81 89 FE 06 01 FE 06 01 39 06 1C 03 70 65 6D 22  üë■..■..9...pem"
00000672  8B 02 2D 01 01 3C 42 45 47 49 4E 20 50 55 42 4C  ï.-..<BEGIN PUBL
00000688  49 43 20 4B 45 59 01 14 F0 E5 2D 0A 4D 49 47 62  IC KEY..≡σ-.MIGb
00000704  4D 42 41 47 42 79 71 47 53 4D 34 39 41 67 45 47  MBAGByqGSM49AgEG
00000720  42 53 75 42 42 41 41 6A 41 34 47 47 41 41 51 42  BSuBBAAjA4GGAAQB
00000736  46 6A 6E 75 6B 41 65 77 47 52 7A 33 67 33 76 4C  FjnukAewGRz3g3vL
00000752  50 63 4A 78 34 70 4D 48 67 50 74 77 0A 49 37 6B  PcJx4pMHgPtw.I7k
00000768  5A 45 72 47 42 35 51 33 65 67 74 67 70 62 35 31  ZErGB5Q3egtgpb51
00000784  4C 4D 5A 2B 50 55 63 4A 5A 6F 45 70 36 75 66 56  LMZ+PUcJZoEp6ufV
00000800  6F 4A 6B 43 49 50 36 42 35 5A 35 6B 62 78 44 61  oJkCIP6B5Z5kbxDa
00000816  65 76 41 32 50 48 4B 77 42 75 5A 49 48 0A 36 73  evA2PHKwBuZIH.6s
```

Figure 3.2: `.ldb` file including the public key x-coordinate (yellow), y-coordinate (green), and the private key (blue).

In Figure 3.2 the bytes that represent the public key are marked in yellow for the x-coordinate (bytes 416-481) and green for the y-coordinate (bytes 483-547). In the same figure the private key is marked in blue (bytes 589-654). Starting around byte number 672 we can see the start of the public key in PEM format. We can conclude that the IndexedDB is not an appropriate storage mechanism to securely store a private key that is used to uniquely identify a browser. This is because using the `.ldb` file, the private key can easily be extracted from the browser, even though the private key is marked as non-extractable in the JavaScript environment.

3.4 Application Vulnerability Assessment

In the following sections we report on our findings of the application vulnerability assessment (AVA). Our target of evaluation (TOE) is our protocol as defined in Protocol 1. During this assessment, emphasis will be placed on the underlying use of the IndexedDB and the Web Cryptography API. The

AVA will be performed on the basis of five categories, using the criteria as defined in [1, 2].

3.4.1 Bypassing

Bypassing can occur in different forms, all of which include an attacker being able to avoid any security enforcement. Two different potential vulnerabilities are identified that bypass a security measurement. The first vulnerability is introduced by the underlying representation of the IndexedDB. This vulnerability allows an attacker to extract the private key by exploiting how the IndexedDB serialises JavaScript objects. This vulnerability has already been discussed in Section 3.3, because of its critical importance to security of the protocol. In this section we will discuss what an adversary would be able to do once the same origin policy is bypassed.

Same Origin Policy

This section will describe what an adversary could do when it has successfully bypassed the same origin policy (SOP). The SOP regulates cross-object access control in the browser using domain names [18]. The intents of SOP is to let a user access malicious websites, without giving these malicious websites access to honest websites. The IndexedDB is also protected by the SOP. Under normal circumstances we can assume that a malicious site is not able to access the IndexedDB of another, distinct domain. However, an adversary will have full access to the IndexedDB of a specific domain in the case that it's able to successfully bypass the SOP. In the subsequent section we discuss what operations could be performed in the event that the SOP is bypassed, in relation to the IndexedDB that is used by our protocol. Below, a summation is given in which we discuss two different operations which an adversary could perform once the SOP is bypassed. In Figure A.6 and Figure A.7, coding proof-of-concepts are given in JavaScript that implement these two operations.

- Deletion of the database.
  While it is completely valid for a user to delete the IndexedDB, it could also be done by an adversary that has bypassed the SOP. After the database, including the key pair, is deleted, the user can no longer authenticate using our protocol. Thus, as a consequence the user would be denied access to the service provider. The deletion of the database is a permanent operation, and no backup system is offered by default.

- Generation of signatures.
  When an adversary has access to the IndexedDB which stores the key-pair, it could in theory generate infinitely many signatures. Generating many signatures does not weaken the underlying ECDSA al-

gorithm, given that the pseudo random number generator of the Web Cryptography API does not leak information about any of the bits of the pseudo random number k. This number k is introduced in Section 2.4.4, and is used during signature generation. In fact many types of nonuniformities in k can reveal the private key, given sufficiently many signatures [6]. In the case that the exact same k is used twice to sign two distinct messages, the private key will be leaked. A proof of computing the private key if the same k is used twice is given in Figure B.2.

3.4.2 Tampering

Tampering includes any attack based on an adversary attempting to influence the behaviour of our protocol. We have not identified any tampering vulnerabilities, since the protocol is fully automated and thus not influenced by the user. Furthermore, we do not identify any vulnerabilities in performing the protocol in a different order, or by purposefully delaying or sending incorrect messages to the service provider.

3.4.3 Direct Attacks

In this section we identify one direct attack in our protocol, which might occur in three different forms. The security of the our protocol relies on the use of a strong pseudo random number generator. Below we will discuss each of the three instances in which a pseudo random number has to be generated, and what the consequences are if the number can be predicted, ie. that the number is easy to distinguish from random.

- Key pair generation.
  Upon the first use of our protocol in the browser, an EC key pair has to be generated. In order to do so, a pseudo random number has to be generated. This number represents the private key, from which the corresponding public key is derived. Whenever this pseudo random number becomes predictable, ie. is no longer random, the private key and thus also the public key become predictable. This would undermine the security of our protocol.

- Challenge generation.
  A pseudo random number generator is used in order to ensure that a challenge, which is generated on the server side, is in fact unpredictable. However, our protocol becomes vulnerable to a replay attack if the challenge becomes easy to predict.

- Signature generation.
  As described in Section 2.4.4 the generation of an EC signature requires a random integer k to be generated. If enough bits of k are

leaked, it becomes possible to extract the private key from a signature, given sufficiently many signatures [6]. A proof of computing the private key if the same k is used twice is given in Figure B.2.

3.4.4 Monitoring

Monitoring the challenges of our protocol over the network, as described in Section 3.1.1, will not diminish the security of our protocol. If the challenges are monitored, then the protocol will not offer any non-reputability. This is due to the fact that the user has to sign a certain challenge, and as a consequence the user cannot deny that it tried to authenticate to a specific service provider. This is because an attacker that is monitoring the network can see the challenge, and the signature that is used to sign that specific challenge. However, an attacker monitoring the authentication process does not limit the overall security of the protocol.

3.4.5 Misuse

Misuse can occur when a user is not using our protocol in accordance with the predefined procedures. We argue that the chance of misuse is very slim, because our protocol is fully automated using JavaScript. We therefore do not identify a possibility for a user to incorrectly use the protocol.

3.5 Limitations

We have noted that our protocol has several limitations. These limitations will be discussed in this section, after which we will give our recommendation on how these limitations could be handled.

3.5.1 LevelDB

In Section 3.3 we saw that even though marking the private key as non-extractable upon generation using the Web Cryptography API, the private key could still be extracted from the underlying storage that is used by the IndexedDB. As a consequence it cannot be guaranteed that the private key can be used to uniquely identify a browser, and thus the private key should not be stored using the IndexedDB.

3.5.2 Persistence of the IndexedDB

At any point in time the IndexedDB might be deleted, this can be done by the user itself, the browser, or by other applications. As a consequence, our protocol must be aware of this specific property of the IndexedDB. In other words, our protocol relies on the expectation that a user owns a private key, however the user could lose this key at any point of time. Therefore a fallback option must be put in place to overcome the loss of the private key.

3.5.3 Private Browsing

Directly related with the previous limitation, most browser have a built-in option which allows users to use the Internet without the browser storing information about their browsing session. During such private browsing sessions, the browser will not store, among other things, caches, cookies and browsing history. Furthermore, private browsing sessions create a new, temporary, IndexedDB for the specific private browsing sessions. Whenever users use our protocol in a private browsing sessions, this temporary IndexedDB version will be used. Once the private browsing session has ended, the temporarily created IndexedDB and the private key will be deleted.

3.5.4 Shared Browsers

Since our protocol uses the browser to uniquely identify a user, the browser should only be used by that specific user to authenticate to the service provider. Giving access to a private key, by sharing a browser, defeats the purpose of our protocol. The browser is unable to uniquely identify a user once the browser is shared between users.

3.6 Recommendations

We recommend that our protocol as defined in Protocol 1 should not be used as two factor authentication. This is because said protocol relies on a strong connection between the browser and the generated private key. This is essential because if a private key can be extracted from browser, and then be used in another browser, the private key can no longer be used to uniquely identify a user. As a consequence, ownership over the private key can no longer be used as a factor during two-factor authentication. We have shown that a private key stored in the IndexedDB can be retrieved in the Google Chrome browser by inspecting the underlying LevelDB file.

Chapter 4

Conclusions

This thesis proposes a protocol that describes the interaction between a user and a service provider during authentication. The protocol provides two-factor authentication in the browser, which optionally supports multiple browsers to be used for authentication. Before the protocol, that is based on ECDSA, can operate in a secure manner a strong and reliable connection between the browser and the private key is crucial. It should not be possible to extract the private key from the browser. In our protocol the Web Cryptography API is used for elliptic curve key pair generation, and ECDSA signature generation. The generated key pair could not be extracted in the JavaScript environment, however we have shown that a private key stored in the IndexedDB could be extracted. Thus, the underlying storage mechanism of the IndexedDB in the Google Chrome browser can be exploited to extract the private key. Therefore the IndexedDB should not be used to securely store a private key.

4.1 Further Research

In this section we will touch upon some of the topics that remain unanswered in this thesis.

- Private key storage in the IndexedDB.
  As of now the most critical flaw in our protocol is the storage of the private key. We have shown that the private key can easily be extracted from the Google Chrome browser's underlying LevelDB. Another technique for storing the private key, such that it can easily be used in the browser, has to be found. Another option that could be researched is how the IndexedDB could be altered in such a way that it does provide the needed connection between the browser and the private key.

- Persistence of the IndexedDB.
  We described how the IndexedDB may, for several reasons, loose the data that was previously stored. This also includes the private key. To overcome the in-persistence of the IndexedDB, a mechanism must be put into place to overcome the loss of a private key. This mechanism has to be secure and resistant to any possible exploitations. Further research is needed to overcome this issue.

- Electronic IDentification Authentication and trust Services.
  Once a viable solution is found to store the private key such that it cannot be extracted from the browser, an assessment of the assurance level of the protocol could be made on the basis of the electronic IDentification Authentication and trust Services (eIDAS). eIDAS is an European regulation that specifies the minimum technical specifications and procedures for assurance levels for electronic identification [15]. The eIDAS regulation could be used to allow our protocol to be used in combination with European electronic identity cards to authenticate European passport holders.

Bibliography

[1] ISO/IEC 15408-3. Information technology — security techniques — evaluation criteria for it security — part 3: Security assurance requirements.

[2] ISO/IEC 18045. Information technology — security techniques — methodology for it security evaluation.

[3] Claudia Ziegler Acemyan, Philip Kortum, Jeffrey Xiong, and Dan S Wallach. 2FA Might Be Secure, But It's Not Usable: A Summative Usability Assessment of Google's Two-factor Authentication (2FA) Methods. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 62, pages 1141–1145. SAGE Publications Sage CA: Los Angeles, CA, 2018.

[4] Fido Alliance, 2020. `https://fidoalliance.org`.

[5] Elaine Barker. PUB 800-57 Part 1 Revision 5 - Recommendation for Key Management: Part 1 – General, February 2009.

[6] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 3–20. Springer, 2019.

[7] Sonia Chiasson, Paul C van Oorschot, and Robert Biddle. A Usability Study and Critique of Two Password Managers. In *USENIX Security Symposium*, volume 15, pages 1–16, 2006.

[8] World Wide Web Consortium. Indexed Database API 2.0, 2018. `https://www.w3.org/TR/IndexedDB-2`.

[9] World Wide Web Consortium. Web Cryptography API, 2020. `https://www.w3.org/TR/WebCryptoAPI`.

[10] MDN contributors. Web Crypto API, 2019. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API`.

[11] Cryptography. Version 3.2.1. `https://github.com/pyca/cryptography`.

[12] Quynh Dang. PUB 800-107 Revision 1 - Recommendation for Applications Using Approved Hash Algorithms, August 2012.

[13] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[14] Mohamed Hamdy Eldefrawy, Khaled Alghathbar, and Muhammad Khurram Khan. OTP-based two-factor authentication using mobile phones. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 327–331, 2011.

[15] EUR-Lex. Commission Implementing Regulation (EU) 2015/1502 of 8 September 2015.

[16] Sanjay Ghemawat and Jeff Dean. leveldb, 2020. `https://github.com/google/leveldb`.

[17] Simon Josefsson and Sean Leonard. Textual Encodings of PKIX, PKCS, and CMS Structures. *IETF, RFC 7468*, 4 2015.

[18] Chris Karlof, Umesh Shankar, J Doug Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 58–71, 2007.

[19] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[20] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.

[21] Mozilla MDN Web Docs. IndexedDB API Basic concepts. `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB`.

[22] Alfred Menezes. Evaluation of security level of cryptography: the Elliptic Curve Discrete Logarithm Problem (ECDLP). *University of Waterloo*, 2001.

[23] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[24] Mozilla. Avoid SQLite In Your Next Firefox Feature, 2020. `https://wiki.mozilla.org/Performance/Avoid_SQLite_In_Your_Next_Firefox_Feature`.

[25] Aleksandr Ometov, Sergey Bezzateev, Niko Mäkitalo, Sergey Andreev, Tommi Mikkonen, and Yevgeni Koucheryavy. Multi-factor authentication: A survey. *Cryptography*, 2(1):1, 2018.

[26] Thanasis Petsas, Giorgos Tsirantonakis, Elias Athanasopoulos, and Sotiris Ioannidis. Two-factor authentication: is the world ready? Quantifying 2FA adoption. In *Proceedings of the eighth european workshop on system security*, pages 1–7, 2015.

[27] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons. A usability study of five two-factor authentication methods. In *Fifteenth Symposium on Usable Privacy and Security*, 2019.

[28] Ronald L Rivest, Adi Shamir, and Len Adleman. On Digital Signatures and Public-Key Cryptosystems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1977.

[29] Armin Ronacher. Flask. Version 1.1.2. `https://palletsprojects.com/p/flask`.

[30] U.S. DEPARTMENT OF COMMERCE, Institute of Standards and Technology. FIPS PUB 186, May 1994.

[31] U.S. DEPARTMENT OF COMMERCE, Institute of Standards and Technology. PUB 800-106 - Randomized Hashing for Digital Signatures, February 2009.

[32] U.S. DEPARTMENT OF COMMERCE, Institute of Standards and Technology. FIPS PUB 186-4 - Digital Signature Standard (DSS), July 2013.

[33] Zhiyang Xia, Ping Yi, Yunyu Liu, Bo Jiang, Wei Wang, and Ting Zhu. GENPass: a multi-source deep learning model for password guessing. *IEEE Transactions on Multimedia*, 22(5):1323–1332, 2019.

Appendix A

Implementations

```python
from flask import Flask, render_template, redirect, request
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric.utils import encode_dss_signature
from cryptography.exceptions import InvalidSignature
from secrets import randbits

app = Flask(__name__)
_correctCredentials = False
_challenge = None
_password = ""

# Convert random bits to Uint8Array for convenience
def getUint8Array(number):
    array = []
    for _ in range(64): # 512/8
        array.insert(0, number % 256)
        number = number >> 8
    return array

@app.route('/')
def index():
    return redirect("/login", code=302)

@app.route('/login')
def login():
    return render_template("login.html")
```

```python
# Listens for the username and password of the user
# Responds with a 512 bits Uint8Array challenge
@app.route('/credentials', methods=['POST'])
def credentials():
    global _correctCredentials, _challenge, _password
    username = request.json.get("username")
    password = request.json.get("password")

    # Check the username and password, in this case hardcoded
    if username == "admin" and password == "admin":
        _correctCredentials = True
        _password = "admin"

    _challenge = randbits(512)
    print(_challenge)
    # Send a random (512 bits) challenge to the user
    return {"challenge": str(getUint8Array(_challenge))}

# Verifies the signature, and responds wether or not the login attempt was successful
@app.route('/verify', methods=['POST'])
def verify():
    global _correctCredentials, _challenge
    # Public key should be compared with the expected public key in a database
    publicKey = request.json.get("publicKey")
    r = request.json.get("r")
    s = request.json.get("s")

    publicKey = load_pem_public_key(bytes(publicKey, encoding="utf-8"))
    signature = encode_dss_signature(int(r, base=16), int(s, base=16))

    try:
        publicKey.verify(
            signature,
            _challenge.to_bytes(64, byteorder='big') + _password.encode(encoding="utf-8"),
            ec.ECDSA(hashes.SHA512())
        )
    except InvalidSignature:
        _correctCredentials = False

    # Both signature and login credentials are correct
    if _correctCredentials:
        return {"result": True}
    return {"result": False}

if __name__ == '__main__':
    app.run(debug=True)
```

Figure A.1: Minimal server implementation.

```
const databaseName = "loginDatabase";
const databaseVersion = 1;
const objectStoreName = "crypto";
const indexName = "keyPair";
const keyName = "key"
var firstUse = false;
var numTries = 0;

// Solution:
// https://stackoverflow.com/questions/27359748/how-to-add-initial-data-in-indexeddb-only-once/57534229

// Gets called when the user presses the login button.
// The function takes the following steps:
// 1. If this is the first login attempt by the user, the IndexedDB and a key pair will be generated.
// 2. Else the (existing) IndexedDB will be openend.
// 3. If no errors occur in step 1 and 2, the function sendCredentials() will be called.
function login(){
    if (!window.indexedDB) {
        alert("IndexedDB is not supported by this browser, login unsuccessful!");
        return;
    }

    if (!window.crypto || !window.crypto.subtle) {
        alert("Web Cryptography API is not supported by this browser, login unsuccessful!");
        return;
    }

    if (!window.isSecureContext) {
        alert("The implementation is not run in a secure environment, login unsuccessful. See https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts for more info.");
        return;
    }

    const request = indexedDB.open(databaseName, databaseVersion);

    request.onupgradeneeded = event => {
        var db = event.target.result;
        // First time opening the database, and thus the first time to use our algorithm.
        if (!db.objectStoreNames.contains(objectStoreName)) {
            var objectStore = db.createObjectStore(objectStoreName, {autoIncrement: true});
            objectStore.createIndex(indexName, indexName, {unique: true});
            firstUse = true;
        }
    }
    request.onsuccess = event => {
        var db = event.target.result;
        if (firstUse) {
            window.crypto.subtle.generateKey(
                {
                    name: "ECDSA",
                    namedCurve: "P-521",
                },
                false,
                ["sign"]
            )
            .then(keyPair => {
                window.crypto.subtle.exportKey('spki', keyPair.publicKey)
                .then(function(spki) {
                    pem = spkiToPEM(spki);
                    var object = {
                        publicKey: keyPair.publicKey,
                        privateKey: keyPair.privateKey,
                        pem: pem
                    }
                    // Add the generated key pair to the database.
                    var transaction = db.transaction([objectStoreName], "readwrite");
                    var objectStoreRequest = transaction.objectStore(objectStoreName)
    ;

                    var request = objectStoreRequest.add(object, keyName);
                    request.onsuccess = event => {
                        firstUse = false;
                        sendCredentials(db);
                    }
                    request.onerror = event => {
                        alert("Error adding the key pair to the IndexedDB.");
                    }
                });
            });
        } else{
            sendCredentials(db);
        }
    }
    request.onerror = event => {
        alert("IndexedDB open error: " + request.error);
    }
    request.onblocked = event => {
        alert("IndexedDB request was blocked.");
    }
}
```

```
// Receives an database object from the function login().
// Send the username and password to the server, and waits for a response.
function sendCredentials(db) {
    var username = document.getElementById("username").value;
    var password = document.getElementById("password").value;

    // Send the username and password wrapped in a JSON object to the server.
    fetch('http://127.0.0.1:5000/credentials', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({
            username: username,
            password: password
        }),
    })
    .then(response => response.json())
    .then(data => {
        // Receive a challenge that has to be signed.
        signMessage(db, new Uint8Array(JSON.parse(data.challenge)).buffer, password);
    })
}

// Sign the challange, send it to the server, and wait for its response.
function signMessage(db, challenge, password) {
    var transaction = db.transaction([objectStoreName], "readonly");
    var objectStore = transaction.objectStore(objectStoreName);
    var request = objectStore.get(keyName);

    request.onsuccess = event => {
        keyObject = event.target.result;

        var utf8Encode = new TextEncoder();
        var pass = utf8Encode.encode(password);
        var messagePassword = new Uint8Array(challenge.byteLength + pass.byteLength);
        messagePassword.set(new Uint8Array(challenge));
        messagePassword.set(pass, challenge.byteLength);
        window.crypto.subtle.sign(
            {
                name: "ECDSA",
                hash: {name: "SHA-512"},
            },
            keyObject.privateKey,
            messagePassword
        )
        .then(signature => {
            var r = new Int8Array(signature.slice(0, 66));
            var s = new Int8Array(signature.slice(67));
            r = buf2hex(r);
            s = buf2hex(s);

            fetch('http://127.0.0.1:5000/verify', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    r: r,
                    s: s,
                    publicKey: keyObject.pem
                }),
            })
            .then(response => response.json())
            .then(data => {
                // result tells us whether the login was successful or not.
                // However, sometimes the signature verification fails when it should
    not.
                // So we retry for a maximum of 10 times.
                if (data.result){
                    numTries = 0;
                    alert("Successful login.");
                }
                else{
                    if (numTries >= 10){
                        alert("Try again!");
                        numTries = 0;
                        return;
                    }
                    numTries+=1;
                    sendCredentials(db);
                }
            })
        });
    }
    request.onerror = event => {
        alert("An error occured" + event.target.result);
    }
}
```

```
// Helper functions:
// https://stackoverflow.com/questions/40314257/export-webcrypto-key-to-pem-format
function spkiToPEM(keydata){
    var keydataS = arrayBufferToString(keydata);
    var keydataB64 = window.btoa(keydataS);
    var keydataB64Pem = formatAsPem(keydataB64);
    return keydataB64Pem;
}

function arrayBufferToString( buffer ) {
    var binary = '';
    var bytes = new Uint8Array( buffer );
    var len = bytes.byteLength;
    for (var i = 0; i < len; i++) {
        binary += String.fromCharCode(bytes[i]);
    }
    return binary;
}

function formatAsPem(str) {
    var finalString = '-----BEGIN PUBLIC KEY-----\n';

    while(str.length > 0) {
        finalString += str.substring(0, 64) + '\n';
        str = str.substring(64);
    }

    finalString = finalString + "-----END PUBLIC KEY-----";

    return finalString;
}

// https://stackoverflow.com/questions/40031688/javascript-arraybuffer-to-hex
function buf2hex(arrayBuffer) {
    return Array.prototype.map.call(new Uint8Array(arrayBuffer), x => ('00' + x.
    toString(16)).slice(-2)).join('');
}
```

Figure A.2: Minimal JavaScript implementation.

```html
<!-- https://www.w3schools.com/howto/howto_css_login_form.asp -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Login Example Page</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <script type="text/javascript" src="{{ url_for('static',filename='script.js') }}"
    ></script>
  </head>
  <body>
      <div class="container limitWidth">
        <label for="username"><b>Username</b></label>
        <input type="text" placeholder="Enter Username" id="username" required>

        <label for="password"><b>Password</b></label>
        <input type="password" placeholder="Enter Password" id="password" required>

        <button onclick="login()">Login</button>
    </div>
  </body>
</html>
```

Figure A.3: HTML login page.

```css
/* https://www.w3schools.com/howto/howto_css_login_form.asp */
body {font-family: Arial, Helvetica, sans-serif;}
.limitWidth {
  border: 3px solid #f1f1f1;
  max-width: 900px;
  margin: 0 auto;
}

input[type=text], input[type=password] {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  box-sizing: border-box;
}

button {
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  cursor: pointer;
  width: 100%;
}

button:hover {
  opacity: 0.8;
}

.container {
  padding: 16px;
}
```

Figure A.4: CSS page.

```python
def main():
    inFile = open("XXXXXX.ldb", "rb")
    data = inFile.read()
    inFile.close()
    # Private key starts after 0xD63081D30201010442
    start = bytes.fromhex("D63081D30201010442")
    startPrivateKey = data.index(start) + len(start)
    privateKey = data[startPrivateKey:startPrivateKey+66]
    print(int(privateKey.hex(), base=16))

if __name__ == "__main__":
   main()
```

Figure A.5: Locating and printing the private EC key in a `.ldb` file.

```javascript
    indexedDB.databases().then(database => {
        // For each database
        database.forEach(element => {
            // Delete it
            indexedDB.deleteDatabase(element.name);
        });
    });
```

Figure A.6: Deleting a database.

```javascript
indexedDB.databases().then(databases => {
    databases.forEach(element => {
        const request = indexedDB.open(element.name, element.version);

        request.onsuccess = event => {
            var db = event.target.result;
            var objectStoreNames = db.objectStoreNames;

            for (i=0; i<objectStoreNames.length; i++){
                var name = objectStoreNames.item(i);
                var transaction = db.transaction(name, "readwrite");
                var objectStore = transaction.objectStore(name);

                objectStore.getAll().onsuccess = event => {
                    event.target.result.forEach(index => {
                        // If this index holds a keypair,
                        // it can then be used to generate signatures.
                        keyObject = index;
                        window.crypto.subtle.sign(
                            {
                                name: "ECDSA",
                                hash: {name: "SHA-512"},
                            },
                            keyObject.privateKey,
                            new Uint8Array()
                        )
                        .then(signature => {
                            // Handle signature
                        });
                    })
                }
            }
        }
    });
});
```

Figure A.7: Computing signatures.

Appendix B

Proofs

Proof that a digital signature forgery can be performed if no hash function is used.

- Let M be a message of length m bits, so $|M| = m$.
- Let F be a signature generation function, which takes a private key pk and M.
- Let n be a maximum message length in bits of which F computes the signature over.
- Let S be a signature generated by F.

The function F interprets M on its length m compared to n, we make a case distinction based on m relative to n:

1. $m < n$. The message M is prepended with $(n - m)$ 0s. Thus $|M| = n$, and we are in case 3.
2. $m > n$. The rightmost $(m - n)$ bits are truncated. Thus $|M| = n$, and we are in case 3.
3. $m = n$. We know S and M such that $S = F(pk, M)$.
    - Let M' be M appended with an arbitrary bit sequence of length greater than 0.

   We can then created infinitely many signatures S', such that $S = S'$, by using M'. This is because any of the bits that are appended to M are truncated again by F:
   $S = F(pk, M) = F(pk, M') = S'$.

Figure B.1: Digital signature forgery when no hash function is used.

Proof that ECDSA is vulnerable to private key recovery. The private key can be recovered, given that two signatures use same integer k to generate a signature.

- Let (r, s) and (r, s') be two distinct signatures.
- Let m and m' be two distinct messages.
- Let h and h' be a digest of a hash function H such that $h = H(m)$ and $h' = H(m')$.
- In this proof we use the EC domain parameters as defined in Section 2.1.

From the ECDSA signature generation algorithm we get:

- $s \equiv k^{-1} \cdot (h + r \cdot pk) \pmod{n}$ and
  $s' \equiv k^{-1} \cdot (h' + r \cdot pk) \pmod{n}$

Multiplication by k gives us:

- $s \cdot k \equiv (h + r \cdot pk) \pmod{n}$ and
  $s' \cdot k \equiv (h' + r \cdot pk) \pmod{n}$

Subtraction gives us:

- $(s \cdot k) - (s' \cdot k) \equiv (h + r \cdot pk) - (h' + r \cdot pk) \pmod{n}$
  $(s - s') \cdot k \equiv h - h' \pmod{n}$

Rewriting for k gives:

- $k \equiv (s - s')^{-1} \cdot (h - h') \pmod{n}$

We can know derive k from s, s', h, h', and n. Given k we can compute the private key pk, by using either s or s'. Since $s \equiv k^{-1}(h + r \cdot pk) \pmod{n}$, we compute pk in the following way:

- $s = k^{-1}(h + r \cdot pk)$
  $s \cdot k = h + r \cdot pk$
  $s \cdot k - h = r \cdot pk$
  $pk = r^{-1}(s \cdot k - h)$

Figure B.2: Computing the private key if the same k is used.