Cryptographic Authentication & Authorisation Methods for the Web using Blockchain Technologies

Corey Bothwell
University of Zurich
Department of Informatics
MOEC0532 Final Project
31 May, 2022
Zurich, Switzerland
corey.bothwell@uzh.ch

***Abstract*—This paper investigates novel techniques for common web-based authentication & authorisation workflows using blockchain technology and smart contracts. We begin by enumerating the traditional methods used in web authentication & authorisation and highlight their historical development as well as some of their contemporary limitations. We then propose a novel architecture that supplants traditional password-based authentication workflows using sessions or tokens and instead relies on the growing popularity of browser based signing authorities, or "crypto wallets". We present an architectural overview as well as a prototypical implementation written in Typescript. We conclude by discussing some interesting properties of the system and avenues for further development.**

***Index Terms*—blockchain, cryptography, authorisation, authentication, web security**

I. Introduction

The popularity of blockchain technology has grown tremendously in the past few years, largely driven by the rise of cryptocurrencies. This rise in popularity has a number of second order effects including the rise in familiarity among the general public with Smart Contracts (SC) as well as the growth in use of Browser-Based Signing Authorities, or "wallets" that are used as an extension to popular web browsers such as Firefox, Chrome or Brave.

This presents an interesting opportunity to rethink some of the assumptions underlying web security. The traditional methods used to secure user-facing web services forego the heightened security offered by asymmetric cryptography under the assumption that it harms User Experience (UX): Managing private keys is both cumbersome and dangerous for the average non-technical user. Instead they opt for the ubiquitous password based system, with increasingly more reliance on Two-Factor Authentication (2FA).

A. Problems with Passwords

Password-based authentication comes with its own set of trade-offs. We present an informal overview of some of the most salient limitations.

1) Tension between UX & Security: Passwords present unfortunate compromises between security and UX. A shorter password is easier for the user to remember, yet more vulnerable to brute force attack in which an attacker generates random sequences of possible passwords. This illuminates the justification for the common password requirements that users have become familiar with when registering for popular web services (e.g. "Must be at least 12 characters and contain a symbol"), often to their frustration[1].

2) Vulnerable to Social Engineering: Additionally, the manner in which passwords are used (i.e. inputting them into a web form) makes them more susceptible to social engineering attacks, where users are tricked into providing their password to an attacker imitating the service in question under the pretext of legitimate use.

3) Password Duplication: Passwords are also prone to duplication by users. Ideally, a user would use a separate password for every web service they interact with. However, given the ubiquitous nature of web services in modern life it is an entirely impossible feat for users to remember the unique passwords to the vast amount of services they utilise on a daily basis. The recommended solution is for users to use a special program known as a "password manager", which handles generating and storing passwords on their behalf. However, the use of a password manager has a negative effect on overall UX and presents an additional step in the registration and login flow. Many users are unfamiliar with password managers. Thus, the vast majority of users do without and often elect to duplicate their password(s) across various services [1]. In this scenario, when an attacker compromises a password they gain access to multiple services, increasing the severity of the attack and the scale of the effort required to reconcile compromised

[1]Interestingly enough the more arcane password requirements, such as requiring a special symbol, are misguided ideas aimed at requiring passwords to be less susceptible to brute-force guessing by a determined human. However, they provide virtually no benefit to a machine-based attack, where the *length* of the password is the primary defense. For a humourous take on this see: https://xkcd.com/936/

services.

4) Authorised Services Maintain Opaque Copies: Passwords also suffer from the fact that web services must maintain a copy of a representation of the password in order to compare to the user-supplied input upon authentication. We precisely use the term "representation" here in order to indicate that web services should be using a hashed representation of the password for comparison (see II-C2). Indeed the use of a secure hashed representation of passwords is ubiquitously considered mandatory by the security community, nonetheless there exist many cases in which it is revealed that a web service has failed to do out of negligence. Bauman, Lu, and Lin (2015) found that 11 of the top 500 websites stored passwords in plaintext [2].

Users have no available method to verify that their information has been stored correctly in a secure & hashed format; they must implicitly trust the engineers of the service in question that they have indeed correctly hashed their passwords.

However, even with secure hashing in place, password-based authentication suffers from the very nature that the service maintains a copy of the password representation. Even hashed passwords can be cracked with sophisticated password-cracking mechanisms if they are not created with sufficient entropy [3]. A better method would be *non-custodial* in nature; services should never need to maintain any copy of authentication data. We will see later how our proposed solution provides this.

5) Modern Improvements Further Harm UX: The primary response to the relative weakness of passwords over the past decade has been to utilise Two-Factor Authentication. Two-Factor Authentication involves requiring a "second factor" of authentication during an authentication attempt to mitigate the risk associated with a compromised password. This is most commonly manifested via a one-time code sent to the user in a trusted manner, often to a verified email or phone number (via SMS). More recently, small auxiliary applications can also generate one-time codes known as Time-Based One-Time Passwords (TOTP) based on predetermined parameters and the time-of-day [4]. There are even more secure methods that involve the use of a hardware device, however they remain relatively unpopular due to their negative impact on UX [5].

While providing an additional layer of security, 2FA presents a wide variety of UX challenges for users. Relying on a separate device poses certain challenges. For instance, assuming 2FA via a third party email service, any unavailabilities in the email service could mean the requested web services are inaccessible. In contrast, 2FA via an SMS is dependent on a functioning cell signal, while TOTP's are dependent on the associated mobile device having requisite charge.

B. Alternatives to Password-Based Authentication

There exist alternatives to password-based authentication often used by members of the Information Technology (IT) and Software Development communities[2]. They are most commonly used to authenticate into servers and machines either as a human user or in machine-to-machine communication. These methods usually rely on public-key cryptography and are generally more secure than password based methods [6]. (See section II-A2 for a more detailed introduction.)

If these more secure methods exist, then why have they failed to gain widespread adoption throughout the web?

A complete historical overview is beyond the scope of this paper[3]. Nonetheless we speculate that the dominance of password-based authentication flows sprang from the original convenience that passwords presented in a world with far fewer and less relevant web services.

Additionally, key-based methods also require basic familiarity with encryption protocols and some initial setup that can be intimidating for non-technical users. Many of the traditional methods of setting up public-key authentication methods such as `ssh` are subject to human error which compromises the security of the system[4].

Under this context, it's feasible that simple momentum and familiarity amongst both users and developers have paved the way for the world we have today, with password-based authentication remaining the "default" that users are accustomed to and that developers are comfortable implementing.

However, in a world with ubiquitous and high-value web services that drive large portions of everyday life, it's only natural that web security becomes more and more vital. This is reflected by some of the developments mentioned in I-A5, however the conflicts that these methods present to User Experience are salient and lead us to ask if the IT and development communities are perhaps building on the wrong abstraction.

Indeed, once properly configured, key-based methods are extremely convenient and obviate almost all of the security concerns inherent in password-based authentication. They:

- do not exhibit any tension between UX & security as key generation is an automated process handled by known and audited algorithms. Software on the user's device handles the management of the respective keys, which are much more secure than password-based methods [6].
- are less subject to social engineering attacks, as users are not required to input their private key during any authentication workflow. Users are not even required to **know** their private key in order to authenticate, and in fact for security reasons they shouldn't know, rather it is stored in a secure software on the user's machine. Requests for the private key stand out as abnormal and thus as attacks.
- do not suffer from problems with duplication. Users have no need to memorise their keys which are stored in the managing software.
- do not require that the service being authenticated against maintain a copy of the private key. (Indeed doing so

[2]We speculate that the vast majority of non-technical users are not aware that such methods even exist.

[3]This work was performed in the context of a 3-credit course at the University of Zurich in the Spring of 2022.

[4]E.g. accidentally copying the private key.

would be impossible to support, as knowledge of the private key essentially "unlocks" the public key and any authenticated services it has access to. See II for more details.)
- do not conflict with, but are rather compatible with modern 2FA methods[5].

C. The Role of Blockchains

Blockchains have risen to prominence over the past decade primarily driven by the popularity of cryptocurrencies like Bitcoin and Smart Contracts like those present in the Ethereum ecosystem. These systems represent a completely open distributed ledger which is shared between many different machines. The machines use consensus protocols in order to determine the "true state" of the ledger, and machines are not required to trust each other in order to agree on consensus. The distributed ledger maintains a record of state changes made since the initialisation of the system, or the "Genesis Block". See II for more background. Given their open nature, they are completely dependent on public-key cryptography for authenticating changes proposed to the distributed ledger. In order to mutate the ledger (i.e. to write to the blockchain), users must send a *transaction* representing the change to be made along with a *signature* which is linked to their key-pair. Only with a valid signature is the transaction accepted by the nodes in the system.

The growth of the popularity of these systems has led to enormous demand for interacting with blockchains. Given the open and permissionless nature, there is generally no hard requirements on how a state-changing transaction must be initiated, insofar as it is eventually sent to a node in the network according to a predefined and valid interface, it is accepted. Indeed users are free to use any key management software to generate transactions and send them to a valid node. Users could theoretically even generate signed transactions manually with the help of common encryption libraries.

Most users however, demand something more convenient than sending raw transactions to a node in the network. Indeed the vast majority of Smart Contract users utilise **Browser-Based Signing Authorities** or "browser wallets" for common interactions with the blockchain. These wallets are designed as extensions to common web browsers (Chrome, Firefox, Brave, etc.) that manage the user's keys securely on their behalf and allow the user to interact with the blockchain directly in the web-browser by visiting web applications provided by SC or blockchain developers known as "decentralised applications" or "dApps". The wallets present a simple interface for sending transactions defined by the application and approved & signed by the user.

It is important to note here that the public-key authentication methods employed by browser wallets are fundamentally no different from their more traditional and obscure use in authenticating servers & machine-to-machine communication commonly employed by the software community. This presents a very interesting opportunity for the web security community to rethink some of the foundational assumptions that underpin traditional password-based authentication. Namely, if the fundamental argument for passwords is that they provide familiarity to the user and that key-based methods are too obscure, the growth of these browser-based wallets may offer a solution. Indeed the very use of a "dApp" represents the immature[6] fusion of a traditional modern web service with a distributed ledger replacing the role of the backend database. If blockchains continue to rise in popularity and use, then it is safe to assume that the use of these browser wallets will continue to expand. Widespread adoption would mean that web users around the world would have a simplified, secure, and convenient method to access public-key cryptography. Almost orthogonal to their use for blockchains, the growth of these browser wallets offers a chance to completely rethink the traditional web security model from the ground up.

D. Paper Structure

We follow this introduction with a brief formal overview of the required technical background which includes topics related to blockchains and techniques used in web authentication. Thereafter we present a novel architecture for web authentication using a a Browser-Based Signing Authority known as Metamask[7] along with a simple Smart Contract. We provide a prototypical working code implementation along with screenshots demonstrating its use. We evaluate some interesting properties of the system and then turn our focus towards avenues for further development.

II. Technical Background

We now turn our attention towards some brief technical background. We note that this portion is written informally in order to be accessible to a wide audience. Readers familiar with the concepts enumerated below can freely skip to III.

A. Public Key Cryptography

Public key cryptography or asymmetric cryptography is a method of cryptography that is *asymmetric* in that depends on a pair of keys known as a *key pair* which contains a private key and a public key. The security of the scheme depends on the security of the private key, whereas the public key can revealed without consequence. Public key cryptography has a number of desirable properties, notably that encrypted messages can be sent between parties without the need for any party to reveal the private key. Public key cryptography also facilitates the creation of verifiable digital signatures, which can be independently confirmed as having originated from the signer [7]. See II-A1.

[5]They may however obviate the need. The UX challenges of 2FA naturally remain; we postulate that with the greater security posture provided by key-based methods, 2FA is unnecessary except for highly-sensitive applications.

[6]Immature in the sense of novel and yet to reach its full potential

[7]https://metamask.io/

1) Digital Signatures: Digital signatures are a method of signing and verifying data. Public key cryptography facilitates the creation of digital signatures and proceeds in the following manner: If Alice wants to send verified data to Bob, she sends the data along with a signature generated from her private key. Bob uses Alice's public key to verify that the signature is indeed valid (i.e. that it only could have originated from Alice and nobody else). Refer to [7] for more details.

The digital signature mechanism utilised in public key cryptography is also the foundational method utilised by blockchains for verifying transactions sent to the network. Paraphrasing from [8] (citations maintained): "The Elliptical Curve Digital Signature Algorithm is the digital signature algorithm used by the Ethereum Virtual Machine and serves to provide proof that owners of a wallet have authorised transactions originating from that wallet. (They also serve two further purposes, to establish non-repudiation and unmodifiability, see [9] for more details) Given a signature, a serialized transaction, and public key of the supposed originator of the transaction, the ECDSA allows for verification of a transaction, i.e. to say 'Only the owner of the private key that generated this public key could have signed this transaction' [9]. The ECDSA algorithm also allows the recovery of the public key given a message hash and associated signature."

2) SSH: Secure Shell (SSH) is a protocol for accessing remote machines over a network in an encrypted manner. Importantly, one of the methods of authentication in the SSH protocol is via public key signature verification. The client submits a signed message which the remote server can then recover the public key from. If the public key matches one of the authorised keys, the client is allowed to access the remote machine and a connection is established. For more details see [10]. We will see how our solutions resembles SSH in some aspects.

B. Blockchains

Blockchains are a category of distributed ledger technology. Blockchains maintain a ledger, or a "system state" that is distributed over many copies among various nodes in the network. No single node maintains the system state, and new entries are added to the ledger after a period of determining concensus amongst all nodes in the network via a *consensus mechanism*.

State transitions in the network, commonly called "transactions", are sent via participants and verified to have originated from their key pair by validating the signature alongside the ledger entry. In order to prevent abuse by sending infinitely many transactions, users usually need to pay a small fee represented by tokens in the network for each transaction they wish to commit. The fee is paid to "miners", which are nodes that verify the incoming transactions and participant in the associated consensus mechanism. In the Ethereum ecosystem, this fee is known as "Gas" [8].

Blockchains can provide the capability for Smart Contracts, which represent arbitrary computations stored on the network which can be initiated with an incoming transaction. Smart Contracts have led to the growth of Browser-Based Signing Authorities like Metamask[8] which provide users with a satisfying interface for interacting with Smart Contracts in a standard web browser. These "browser wallets" maintain an asymmetric key-pair (i.e. a public/private key pair) used for signing blockchain transactions; co-opting this key pair for authentication purposes is fundamental to our proposed solution.

One defining characteristic of blockchains that lends our solution interesting properties is their open and public nature: anyone can read the state of the system at any time. We will see later on how we maintain the privacy of user data in our solution despite storing it on the blockchain.

C. Web Authentication Flows

In order for a user to access a protected web service, they must first authenticate. **Authentication** is the process of proving identity, or that the person attempting to access the service is indeed who they claim to be. This is distinct from **authorisation**, which is the process of determining what exactly the user has access to within the service (e.g. Is the user an administrator or simply a normal user?). Since authorisation is application-specific, we don't consider it in great detail here. The focus of this work is on a novel method for authentication that also facilitates authorisation.

1) Basics of Password Authentication: Traditional password-based authentication proceeds in the following manner: The user submits a form containing a unique identifier that is associated with their identity to the service (usually a username or email address) along with a password. The values are sent by the browser in an HTTP request to the service's authentication server which performs a lookup in their database according to the user identifier. If the server successfully finds the user, it retrieves the account information of the user, which includes a value that represents the hash of the original password provided by the user at registration. The server then hashes the password value that the user provided when attempting to login (see II-C2). If these two values match, the user has successfully provided their password and the server returns a token or a cookie (see below) to the user's browser indicating a login success. If the attempt was not successful, the server notifies the browser that the attempt was not successful. Usually the user may retry, although here the implementation is application specific, so this step might vary.

This token/cookie is stored in the user's browser and used to allow access to protected resources during future requests. It is sent alongside all further requests (until logout) to the service to identify the user to the server and to demonstrate that the user has successfully authenticated prior to the request. Importantly, the token/cookie is cryptographically signed by the server, which allows the server to ensure that the only place the token/cookie could have originated was the server itself.

[8]https://metamask.io/

2) Password Hashing: Password hashing is an important concept in password-based authentication. The concept is as follows: in the event of a data breach targeting a web service attackers may secure access to the password database. If the passwords are stored in plain text, the attacker has access to their raw values and may use them to access the service on behalf of any user. Furthermore, since we know that users often re-use passwords across multiple services, further accounts at other services are likely compromised as well.

The solution is to store passwords in the database in a special representation known as a **hash** which is generated by a predetermined hash function. Informally, a hash function is a fundamental concept in mathematics and computer science that describes a special function which can only be computed in one direction. Essentially, it is a "one-way street", and can be used to generate hash values from a plaintext input, however the plaintext cannot be recovered with knowledge of the hash value. There are other desirable properties of hash functions such as "deterministic" (the same output for the same input) and "collision-free" (two different input values do not return the same hash). See Figure 1 for a visual overview[9].

Because we can always compute the hash value from the plaintext input (but never the reverse) we can use this to harden our password database against breach. Instead of storing the raw password value we store a *hash* of its value. When the user attemtps to login, we simply re-hash the provided password input and compare it to the stored hash value. If the hash values match, and assuming our hash function has acceptable properties, we can be sure that the user provided the original password. If the password is ever to become breached, the attacker only has access to the hashed values. Submitting the hashed value in an attempt to authenticate will never result in success since the server would then "hash the hash" resulting in a different value and an invalid comparison.

Despite the benefits offered by hashing passwords, there are still sophisticated methods available that allow attackers to reverse engineer their inputs in a practice known as "password cracking" [3]. Additionally, users have no guarantees that web services correctly hash their passwords in order to harden their password databases. Users are completely dependent on the respective software engineers to correctly implement password hashing [2].

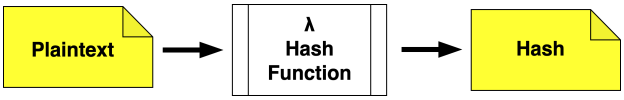

Fig. 1. Visualising a hash function.

3) Stateful vs. Stateless Authentication: Once the user has been authenticated, they must be provided with some small value that represents their successful login. For our purposes we will refer to this as a token. This token value can be used in lieu of re-authenticating for every request. (Imagine if you needed to enter your username and password every time you refreshed the page!) As long as the browser sends the token value alongside every request, the server can interpret the value and determine that the request came from a user who had previously authenticated[10][11].

There exist a wide variety of methods for performing this essential task, however they generally fall into two main categories: **stateful** or **stateless**. This distinction concerns how the server interprets the token value sent alongside the request.

In stateful authentication, the server sends a unique identifier in the token value. The identifier is interpreted by the server and linked to a value in the database that represents the users current authentication status commonly referred to as a "session". If the session value is valid the server knows that the request is authenticated and can continue to perform the requested action. Additional information about the user, such as their authorisation level, can also be stored in the session database. Stateful authentication is the traditional method used in web services and offers a fine-grained level of control over which users have access to the system at any given time. For example, if the service wanted to revoke access to a specific user, they can do so immediately by altering their value in the session store. However, many software developers describe stateful authentication as having downsides at very high scales, as each request must query the database for the session status. At high amounts of users this may slow down the system.

In contrast, stateless authentication places all of the required information regarding the user in the token itself. The popularity of stateless authentication has grown over the proceeding decade, largely driven by the adoption of standards such as JSON Web Token (JWT) [11]. In this case, the token sent by the server upon successful authentication represents the entire package of data necessary to validate the user's request. See Figure 2 for a visual representation of a JWT[12].

Upon receiving a request from the browser with a token, the server simply needs to parse the data, interpret it and verify that the signature is valid. It can then respond immediately with the requested data without the need to query a separate session store. This can potentially provide benefits for the throughput of the system although it comes with certain considerations, since the token represents a completely self-contained and authenticated claim to the server. Because possession of the token grants access to protected resources, it is especially important that it is not compromised, and developers need to take great care in selecting the expiry time of the token. Additionally, since the token is stateless, the service owners lose a portion of the natural fine-grained control available with stateful authentication. For instance, it is not possible to immediately revoke a user's access (since

[9]Image created by the author and inspired by: https://khalilstemmler.com/img/blog/data-structures/what-is-hashing/hash-function.png

[10]Browsers mostly handle this process automatically, however web developers can often override this behaviour to handle the process manually.

[11]For the avoidance of doubt, recall that the server issued and signed the token. By verifying the signature it can be sure that it has not been forged. See II-A1

[12]Provided courtesy of: https://jwt.io/

the token is self-contained and represents an authorised claim) without sophisticated techniques[13].

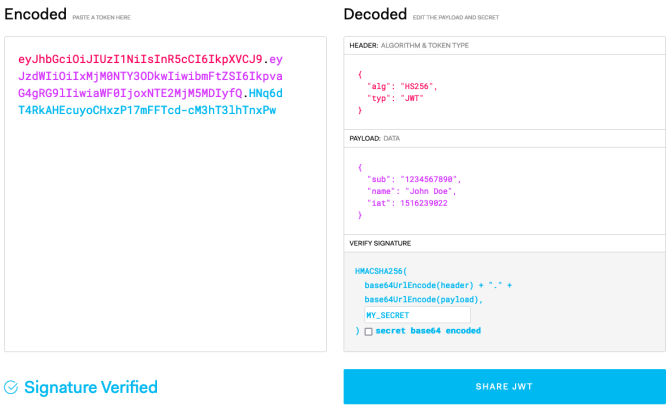

Fig. 2. A visual representation of a JWT. The left side shows the encoded data, while the right side shows the decoded version. Note the "Signature Valid" text which indicates the signature given by the JWT was successfully verified to have originated from the provided secret.

4) Token Storage Mechanisms: Once a token has been generated it needs to be sent to the user's browser and stored for successive requests. Here too the software engineers have a decision to make.

The traditional manner for storing tokens in the browser is via cookies. Most users are familiar with the nature of cookies from their use of common websites. A cookie is a small file that is sent via the server and stored by the browser. The browser automatically sends the cookie upon further requests to the same service. The server receives the cookie and process the data contained therein. Cookies are generally secure and seen as the favoured method for storing sensitive user data in the browser. Because the browser normally sends cookies automatically to the server whenever a request is made, cookies make users subject to an attack vector known as *Cross-Site Request Forgery* (CSRF) (see II-C5), in which requests from another domain send requests to the web service without the user's knowledge. However, modern browsers support an attribute set on the Cookie called "SameSite" [12] which can be used to specify that cookies are only to be sent from the same domain as the service itself. This combined with other techniques[14] make Cookies generally the preferred manner of storing sensitive user data such as the authentication token.

Some web developers prefer storing authentication tokens in a separate storage mechanism offered by the browser known as "LocalStorage", or its cousin "SessionStorage". These storage mechnisms are accessible to JavaScript, the browser-based programming language used to control many aspects of the User Experience of a website. They therefore provide more flexible control to the developer over exactly how the application utilises the token in comparison to Cookies, which are primarily handleded by the browser itself.

The downside of this storage mechanism is that because the token is stored in a manner accessible to JavaScript, it is more vulnerable to being intercepted by malicious code placed by an attacker. (Most of the time, Javascript has no access to Cookie values, whereas JavaScript has access to LocalStorage & SessionStorage by definition.) This is known as *Cross-Site Scripting* (XSS). For this reason, storing authentication tokens in LocalStorage/SessionStorage is generally seen as less secure, however many developers still do so due to the flexibility it gives them for application development.

5) Common Web Attacks: We describe two common web attacks related to the use of authentication tokens. The first is *Cross-Site Request Forgery*, enumerated above. CSRF is an attack the takes advantage of the browser's behaviour of sending cookies automatically along with requests. An attacker simply needs to trick the user into making a request to the service, usually by embedding the request in a malicious site or via a phishing attempt[15]. Refer to Figure 3 for a visual overview[16].

CSRF attacks can be widely mitigated by setting the "Same-Site" attribute on the authorisation Cookie, which instructs the browser to only send the Cookie when the request originates from the same domain. Additional techniques include setting a one-time "CSRF Token" into the browser's form elements on the web service's site. When the server receives a request it validates the special "CSRF Token" has been sent alongside the request and matches the expected value. This technique mitigates the ability of an attacker to send a forged request as they cannot generate the token value expected by the server.

The second attack we describe is known as *Cross-Site Scripting*. XSS is a much more general form of attack on a web service and is performed by injecting malicious JavaScript code into the browser's page. This can happen if web developers fail to verify user provided data is free from any malicious content. In essence the browser has no way of determining which JavaScript is deemed "safe", once it has been injected in the page, the browser will execute it no matter the origin. Since the injected JavaScript can perform arbitrary operations, this attack is inherently very general. It is important to note that the malicious code also has access to any values stored in the browser's storage API's like LocalStorage/SessionStorage, potentially including any authorisation tokens stored therein.

There are a variety of other potential attack vectors when it comes to web security. We have enumerated two of the most common when it comes to the storage of authorisation tokens and we will see how our proposed solution offers some novel approaches for mitigating them.

[13]We note that the debate in software engineering circles about the relative merits of stateful vs. stateless authentication is ongoing. For more information see: https://news.ycombinator.com/item?id=22354534 and https://news.ycombinator.com/item?id=21783303

[14]Notably the use of a "CSRF Token" on requests.

[15]Assumably, the attack benefits the attacker in some way, by revealing data or transferring funds etc.

[16]Diagram created by the author with inspiration from M. Lengyel at: shorturl.at/cqORW

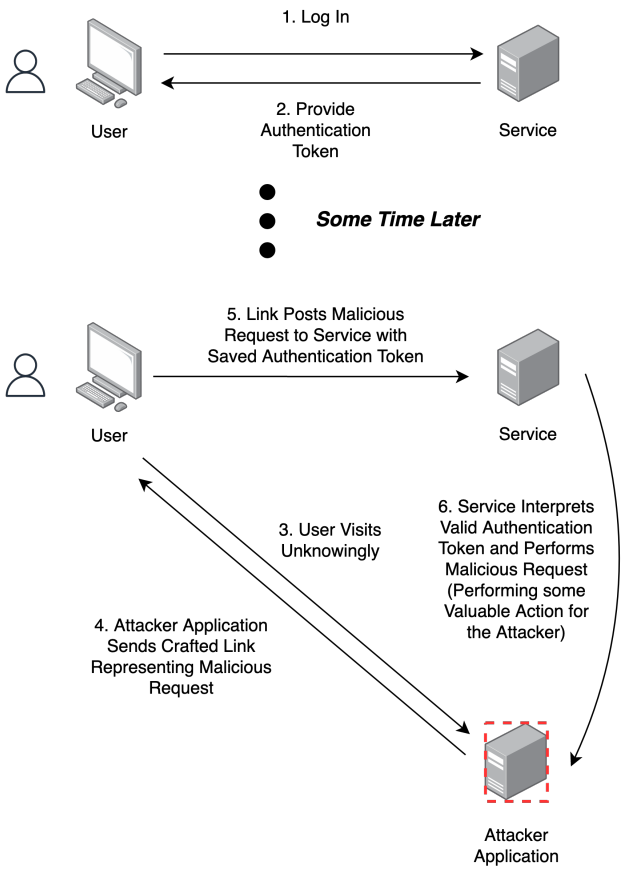

Fig. 3. A visual overview of Cross-Site Request Forgery.

III. Architecture

We now turn our attention to describing the architecture & implementation details of our proposed solution. Our system is comprised of three major components: A client, a server, and a smart contract.

The ***client*** is responsible for presenting information to the user as well as signing and verifying authentication data provided to the server. Integral to our solution is the presence of a browser-based signing authority in the user's browser that the client can use.

The ***server*** is responsible for receiving authentication information from the client, verifying & encrypting it, and providing access to protected resources. The server verifies the authentication attempts from the client by validating signatures sent in the request.

The ***smart contract*** is responsible for storing the authentication claim as determined by the server for the particular user. The smart contract's open nature means that the user can independently verify that the information has been properly encrypted and is the valid signed data the user originally provided.

We will now provide a general overview of the entire authentication flow facilitated by the system. Afterwords, we will turn our attention to describing the implementation of each component in detail.

A. The General Authentication Flow

The general authentication flow takes place in two steps: *Registration* which takes place once, and *Login*, which is a repeated process.

1) Registration: In order to register for a service, the user navigates to the server web page and is presented with a standard web form. The user fills in the required information; thereafter the service's web application can add any required authorisation data to the set of fields provided by the user.

The user is then presented with an overview of the complete information and is then requested to *sign* the information, producing a verifiable digital signature. Additionally, the user is requested to provide their public key (which is safe to distribute), so that the message may be encrypted by the service. The user then sends the signed information along with their public key to the service. These important interactions happen via the browser wallet, on which our implementation is completely dependent.

The server receives the signed information from the client and *encrypts* it with its own private key along with the public key provided by the user. Thereafter the server sends the encrypted information to the smart contract, where it is stored. The server[17] accordingly pays the necessary gas fee for this transaction. Importantly, the stored information is linked to the user's *public key*[18] which is unique and thus serves as an identifier. We use the term *claim* to refer to this stored information, as it represents the user's claim to resources on the server.

Once the transaction has completed, the server returns a response to the client with a location where the client can query the blockchain to retrieve the encrypted data. This is preferred to simply returning the encrypted data directly; the client now has the opportunity to independently verify the data on-chain. The client can retrieve the stored information and verify it by decrypting it[19]. If the decryption succeeds and the decrypted data produces a message with a valid signature, the user can be certain that the information has been successfully encrypted and is unaltered (as otherwise the signature would be invalid).

2) Login: Once the user has successfully registered, they can authenticate to access protected resources from the server. The client authenticates in the following way: When the client makes a request to the server, it sends alongside it another message containing the *it's own signed address*[20] along with a timestamp of the request. The server takes this request information and recovers the public key from the signature. If the recovered address from the signature matches the provided

[17]More accurately, the account associated with the server's private key.

[18]Technically, it is linked to their address, which is a truncated hash of the public key.

[19]Recall that only the client and the server can decrypt the message. The client has provided their public key; the server has encrypted the information with their own private key and the client's public key. See II if required.

[20]Again, for our purposes, the public key and the address of the user's account are largely interchangeable. We will refer to them interchangeably throughout the text.

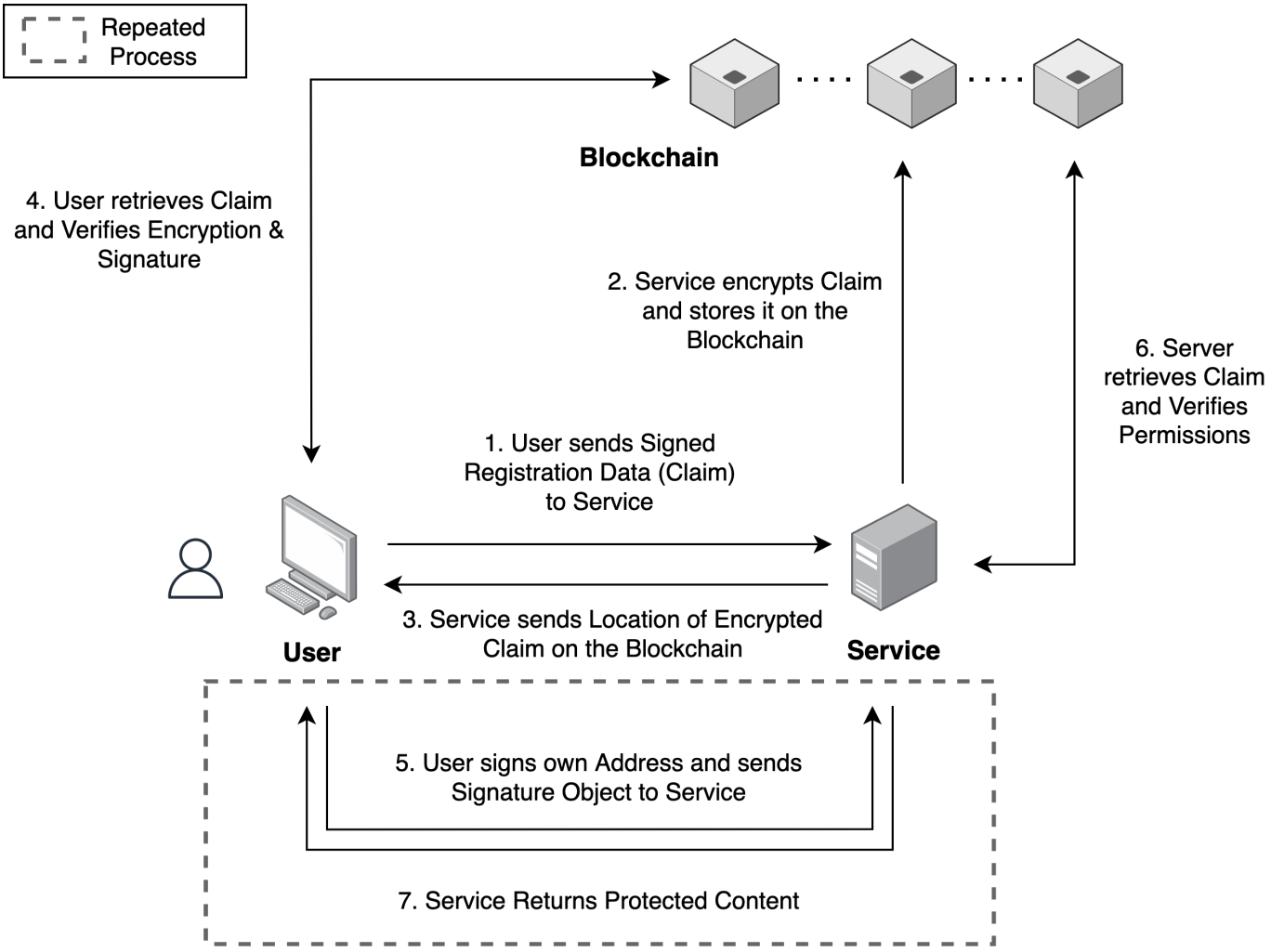

Fig. 4. The general architecture of our authentication workflow.

address then the user is authenticated; only that specific key pair could have produced that signature.

Thereafter the server can query the blockchain for the user's claim information and decrypt it. The server proceeds to read the contents which contains information on the appropriate permissions of the user or any other application-specific information. The server can then return the response as required by the application. See Figure 4 for a visual overview.

B. The Client

The client is responsible for presenting the application interface to the user and interfacing with the browser-based signing authority to send signed messages on behalf of the user. The client also verifies the data on the blockchain to ensure that is has been encrypted and is unaltered.

Our prototypical implementation is built using React.js[21], a library for building user interfaces on the web. We use Ethers.js[22] to interface with the browser wallet. Our implementation was tested using Metamask.

As mentioned previously, the client displays the initial registration page to the user. The user is asked to connect their wallet (if not previously done) and to provide their public key. The user is presented with a basic form asking for a few mock details like their name. Upon form submission, we instruct the browser wallet to request a signature from the user on the data to be sent to the server. The client implementation is free to add any additional data required by the application server to register the user; in our implementation we include a mock "permissions" field that indicates the user is authorised to access images from the server. Figure 5 demonstrates our client-side code that signs the provided values. The `_signTypedData` function provided by the Ethers.js library is compliant with ERC-712 [13].

The client posts the data to server using the browser native `fetch` API and then awaits the response. The client expects a response that contains the *contractAddress* as well as a

[21]https://reactjs.org/

[22]https://docs.ethers.io/v5/

```
export type Claim = {
  subject: {
    email: string;
    name: string;
  };
  permissions: {
    pictures: boolean;
  };
};

// All properties on a domain are optional
export const domain: Domain = {
  name: "Claim Demo",
  version: "1",
  chainId: 31337,
  verifyingContract: "0
      xCcCCccccCCCCcCCCCCCcCcCccCcCCCcCccccccccC"
      , // Okay for demo implementation, not
      verified chain-side
};

// The named list of all type definitions
export const claimTypes: Record<string,
    TypedDataField[]> = {
  Subject: [
    { name: "email", type: "string" },
    { name: "name", type: "string" },
  ],
  Permissions: [{ name: "pictures", type: "bool"
      }],
  Claim: [
    { name: "subject", type: "Subject" },
    { name: "permissions", type: "Permissions" },
  ],
};

export const sign = async (signer: JsonRpcSigner,
     value: Claim) => {
  return await signer._signTypedData(domain,
      claimTypes, value);
};

export const verify = async (value: Claim,
    signature: string) => {
  return await ethers.utils.verifyTypedData(
    domain,
    claimTypes,
    value,
    signature
  );
};
```

Fig. 5. Our client-side implementation that signs the registration data provided by the user and the service. This data is sent to the server which encrypts it and then forwards it to our smart contract.

claimAddress. The *contractAddress* represents the location of the smart contract where the encrypted data has been stored. The *claimAddress* represents the specific location of all of the claim objects associated with the particular web service. Finally, using it's own address (provided by the browser wallet), the client can query the exact location where the encrpyted claim has been stored. See III-D for more details on the simple architecture of our smart contract.

The client can retrieve the claim and can once again use the browser wallet to decrypt it. The user is prompted by the wallet for decryption and can preview the contents. Upon confirmation, if the decryption succeeds the client validates the recovered signature address matches the wallet's own address. If so, the client displays a message to the user that the signature is valid. The user can be sure that the provided registration data has not been tampered with and has been safely stored on-chain in an encrypted manner[23]. Figure 6 demonstrates the simple API exposed by Metamask for decrypting data. See Figure 7 for a screenshot demonstration of the user successfully verifying the signature of retrieved data.

```
const decrypt = async () => {
    if (!!cipher) {
      // This line retrieves the decrypted values
          from the browser wallet
      const plaintext = await provider.send("
          eth_decrypt", [cipher, account]);
      setAppState((s) => ({ ...s, plaintext,
          state: State.Complete }));
    }
};
```

Fig. 6. Using the browser wallet's (Metamask) API to request the user to decrypt the encrypted claim retrieved from the smart contract.

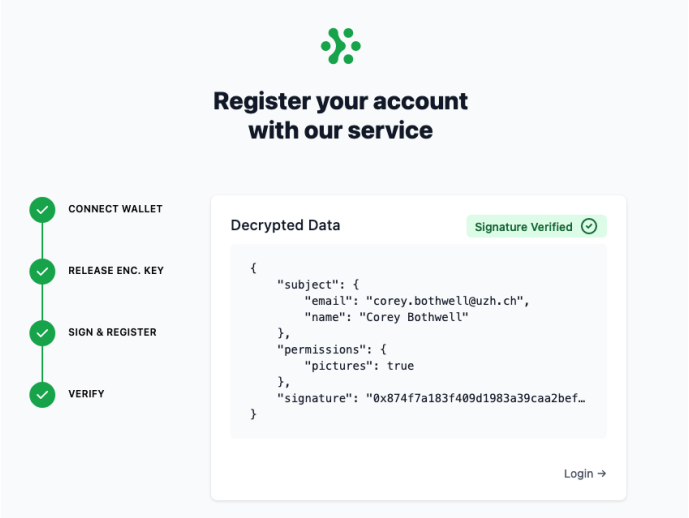

Fig. 7. A demonstration of a user successfully verifying the signature of a decrypted claim object.

Finally, once the registration is successful, the user can authenticate on further requests to access protected resources. In our implementation we simulate this with a protected photograph of a wonderful Swiss landscape.

In order to authenticate, the user simply needs to click a button and sign their own address. We call this the *signature object*; recovering the address contained in the signature and comparing it to the content of the message (the address itself) allows the identity of the sender to be established. If the addresses match, only the user with the corresponding private key could have possibly generated the signature in question. The user is thus authenticated.

[23]Of course, there is nothing stopping a hypothetical service from storing the data in another location. We touch on this concern in the Evaluation portion.

Fig. 8. A successfully authenticated user has been granted access to the protected content, in this case an image of a beautiful Swiss landscape.

The server is responsible for verifying the signature and validating any application-specific permissions stored in the claim data, which the server independently retrieves[24]. If the server successfully validates the signature, the user is successfully authenticated and the server can safely return the requested content.

Figure 8 shows the user's view upon a successful authentication & retrieval of the protected content.

C. The Server

The server is responsible for registering and encrypting claim objects which are sent to the blockchain. The server is also responsible for validating the signatures that users provide upon successive requests for content, as well as determining any application-specific permissions stored in the claim object as required.

The server's role during the registration flow was implied in the previous section and is repeated here in more detail. The server receives a request containing the claim object (filled with values from the provided form on the client-side, as well as any programatically added values), a signature of the claim object provided by the client which indicates the user approves of the data to be sent, and the public key[25] of the user's key pair (provided by the user's browser wallet).

The server encrypts this data with the server's private key, which is kept secret throughout the entire process. We use the open source library `TweetNaCl.js`[26] for the implementation of our encryption procedures. The encrypted value is converted into a format acceptable for storage in the smart contract and is then sent to the chain by the server and is stored in the smart contract at a specific location tied to the server's and user's addresses. The server then returns this location to the client, indicating that the registration was successful and the client is free to verify the stored data. See Figure 9 for details.

Upon further requests for protected resources, the server expects to receive a signed copy of the user's address. If the recovered address from the signature matches the address in the provided message, the user is authenticated. The server can then optionally proof the claim data stored on chain and return

[24]For latency reasons, web servers would likely cache the permission data locally instead of retrieving it from the chain every request. The evaluation portion touches on this consideration

[25]As well as the address derived from their public key.

[26]https://tweetnacl.js.org/#/

```
app.post("/register", async (req: Request, res: Response) => {
  const { claim, claimantAddress, publicKey, signature } = req.body;

  const encrypted = encrypt({
    publicKey,
    data: JSON.stringify({ ...claim, signature: signature }),
    version: "x25519-xsalsa20-poly1305",
  });

  const hexEncrypted = JSON.stringify(encrypted)
    .split("")
    .map((c) => c.charCodeAt(0).toString(16))
    .join("");

  const cipher = `0x${hexEncrypted}`;

  const setClaim = await contract.setClaim(claimantAddress, cipher);
  await setClaim.wait();

  res.send({
    contractAddress,
    claimAddress: signer.address,
  });
});
```

Fig. 9. The request flow for a new registration in our prototypical server implementation. The server receives data from the client and encrypts the received claim along with the signature provided by the user. The server then converts the encoding of the encrypted data into a format acceptable to the smart contract and then stores it there at a location tied to the user's address. The server then returns the location where the encrypted claim can be retrieved.

any requested resources to the client. Figure 10 demonstrates the login code.

We use the open source `Express.js` library to provide the framework for our server implementation[27].

D. The Smart Contract

In contrast the our client and server implementations, the smart contract implementation is decidedly minimal. Our smart contract only performs one initial function: storing the encrypted claim objects on-chain so that they may be verified by the client. We use a nested public mapping `claims` in order to provide a tree like structure. The outer-mapping maps services to their respective inner-mapping; here services can store the encrypted claims for each user indexed by their respective address.

The nested mapping is completely open and publicly writable; protection is afforded by ensuring that that index variable in the outer mapping is *not* able to be set by the caller - instead it corresponds to the address of the transaction sender. This ensures that services can only write the claims of their own users and not maliciously overwrite others.

In order to interact with the smart contract, the server and client require complementary pieces of information from one another. The server requires the address of the user from their wallet in order to correctly store the associated claim

[27]See https://expressjs.com/

```
app.post("/login", async (req: Request, res: Response) => {
  const { address, date, publicKey, signature } = req.body;
  const verifiedAddress = await recoverLogin({ address, date }, signature);

  if (address !== verifiedAddress) {
    res.status(403).send("Not Authorized");
  }

  const hasPermission = await verifyClaim(
    contract,
    signer.address,
    address,
    publicKey
  );

  if (!hasPermission) {
    res.status(403).send("Not Authorized");
  }

  res.send({ content: img });
});
```

Fig. 10. The server validates incoming requests by first ensuring that they have originated from the requester which checks their provided signature. This corresponds to the traditional authentication check performed in password-based authentication when the service would normally check the hash passwords to ensure they match. Afterwords the server can verify the claim object associated with that address stored on-chain to ensure they possess any required permissions. This corresponds to the traditional authorisation check in which the server would typically query a traditional database.

in the inner mapping - its address in the outer mapping will be set automatically by the smart contract upon sending the transaction. The client knows their address (provided by the wallet); it requires from the server the server's address[28] so that it can successfully index to the correct inner mapping and retrieve its own claim therein[29].

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "hardhat/console.sol";

contract Auth {
    mapping(address => mapping(address => bytes)) public claims;

    function setClaim(address claimant, bytes calldata claim) public {
        claims[msg.sender][claimant] = claim;
    }
}
```

Fig. 11. Our smart contract implementation. The nested `claims` mapping is where the claim objects are stored, indexed first by the service address, and then user address.

[28]The address derived from their public key associated with the service, **not** their IP address etc.

[29]Most likely, a more sophisticated implementation would hardcode the server's address on the client-side to obviate the need to send it back and forth over the network.

IV. Evaluation

Our approach to authentication exhibits some desirable properties.

Before we continue, it is important to note that our approach is composed of two largely orthogonal techniques. The first is the manner of authentication practiced by the server upon receiving a request. The server validates the signed message containing the user's address and verifies the signature; it can be said that this technique alone acts a replacement for password-based authentication.

The second is the manner in which we store the user's data encrypted on the blockchain. This is not strictly necessary to facilitate the authentication flow we propose; web services could feasibly store the data anywhere e.g. in a traditional database. We take this a step further and store it on the blockchain in order to demonstrate a workflow in which the user can independently verify their data; web services can also verify authentication data without the need to independently store it[30].

A. Improvements over Password-Based Authentication Offered by Our Method

Let us revisit some of the deficiencies of password-based authentication that we enumerated in I.

We described that passwords:

1) exhibit an inherent conflict between security & UX
2) are vulnerable to social engineering attacks
3) are subject to duplication by users
4) are copied and stored by services which require them
5) suffer from attempts to improve their security with methods like 2FA

Our solution provides considerable advantages over password-based authentication in all of these areas.

1) Essentially No Conflict between Security & UX: Our method offers a greatly improved security posture alongside *improved* UX. When registering an account, the user no longer needs to conform with arbitrary password requirements aimed at improving password strength (e.g. a required length or character set). The user's authentication mechanism is contained entirely within their private key which is managed on their behalf by their browser wallet. As an orthogonal benefit, the user is given the opportunity to sign and later verify that their data has been stored securely.

When logging in, the user only needs to click a button in their browser wallet to provide the required signature. The user no longer needs to remember any passwords or employ the use of a third-party password manager.

2) Much Less Vulnerable to Social Engineering: Our method provides provides a mechanism that is more robust to social engineering attacks. The common need for users to enter their password in password-based authentication means that this is an attractive vector for attackers. Users must be able to distinguish between legitimate and illegitimate requests for their password; anyone familiar with a "Phishing" attack has experienced this.

The equivalent in our method would be the private key stored in the browser wallet, something the user should *never* be asked to reveal. While there are legitimate cases for exporting the private key[31], this is an operation that can be reasonably communicated as being very dangerous, such that any social engineering attacks which attempt to goad the user into doing so can be reasonably identified.

3) No Requirement for Duplication: The UX challenges presented by password-based authentication lead users to often duplicate their passwords across services. Our method completely eliminates this weakness. Since the authentication mechanism takes place via signature verification, there is nothing to duplicate - the signature itself provides proof of possession of the private key and thus identifies the user.

It may be argued that some form of duplication still takes place in the sense that, if the user's private key were to be compromised, all accounts with web services linked to the corresponding public key would be exposed. This has a relatively straightforward solution: using a separate key-pair for every service, a capability that would be relatively trivial for browser wallets to integrate. Wallets already integrate a tree structure of separate key pairs derived from a single master seed, a technique that characterises them as *hierarchical deterministic* (HD) wallets. See Chapter 5 of [9] for more details.

4) No Copy Maintained by Services: One of the most marked benefits offered by our solution is that it obviates the need for web services to store any vulnerable copy of the user's password. In our solution, there is simply nothing to store - authentication is performed by verifying the signature produced by the user's wallet, which is a stateless operation. Only the application specific data related to the user's permissions in the service are stored, in our case on-chain.

Thus, if we assume a breach of a web service's user data, only information related to that specific service is compromised. While grave in its own right, attackers have no way to infiltrate other services. Users do not need to trust that a service has securely implemented their password storage with hashing; there is simply no password to store.

5) Compatible with 2FA Methods: We posit that the increased security posture offered by our approach obviates the need for common 2FA methods for most applications, which were introduced as a response to the security deficiencies of password-based authentication. Nonetheless, for very high-security applications, our method is fully compatible with existing 2FA methods.

Table I summarises these points.

B. A Platform to Reduce XSS & CSRF Vulnerabilities

Interestingly, our architecture also lays the groundwork for new methods to reduce the attack surface of common web

[30]Although this work is prototypical, a valid criticism of our approach would be that even if the user successfully validates the signature of their released data, the server is free to tamper with it and store it in another location. We discuss this observation below.

[31]Mostly regarding transferring the key-pair to another software.

TABLE I
IMPROVEMENTS OFFERED BY OUR AUTHENTICATION METHOD OVER PASSWORD-BASED AUTHENTICATION

Criterion	Password-Based Authentication	Our Method
Conflict between Security & UX?	Inherent	Both Security & UX *Improved*
Social Engineering Risk?	High	Low to Moderate
Encourages Duplication Across Services?	Yes	No. Individual keys for each service possible via HD Wallets [9]
Services Maintain a Copy?	Yes	No
Compatible with 2FA Methods?	Yes, required for acceptable security	Yes, but not required unless very high security requirements

vulnerabilities, like XSS & CSRF. The advantages enumerated here are not present in this prototypical implementation and would be dependent on further development of browser-based signing authorities to natively integrate this authentication method. Nonetheless, we believe it is valuable to discuss them here.

1) XSS: XSS is a wide-ranging and general phenomenon that concerns foreign and malicious JavaScript which is injected into a webpage by an attacker. This is relevant to authentication as credentials (like a authentication token sent by a server) may be stored in LocalStorage/SessionStorage, which is accessible to JavaScript and thus to the attacker.

Our prototypical implementation requests the user's signature directly via the browser wallet's JavaScript API. Our implementation does not store the generated signature object sent to the server, but it's feasible that other implementations might do so, presumably in LocalStorage/SessionStorage. Thus, if malicious JavaScript were injected into the page, an attacker would be able to access the signature object, granting the malicious code access to the service.

However, with further development of browser wallets, we believe that these risks could be heavily mitigated. The procedure for generating signatures of the wallet's address could be standardised and hidden behind a formal API and sent to the service by the browser directly. If the generated object were inaccessible to the JavaScript on the page, XSS vulnerabilities would have a far smaller surface area for intercepting authentication credentials. Presumably, the attacker would only be able to falsify a signature object by indirectly requesting the user to generate it. To combat this, we would propose that a robust implementation of such an API would only allow such objects to be created from the same domain as the service's web server, similar to the "SameSite" attribute used in modern day web cookies. Developers of web services would thus only be responsible for ensuring the absence of XSS vulnerabilities on their own sites.

In conclusion, our authentication flow proposed here does nothing to prevent XSS attacks *in general*, but it does suggest an authentication flow that might one day *reduce* the surface area for XSS attacks to access authentication credentials.

2) CSRF: Our technique also suggests a future where CSRF-style attacks may be all but eliminated outside of more esoteric implementations[32].

[32]E.g. Websites like forums which display user provided content. Here, CSRF tokens would still be required.

We again assume further development of browser-based signing authorities to incorporate a version of the authentication workflow our work proposes. As discussed prior, a robust implementation would include a "SameSite" property similar to modern browser cookies, in which the signature objects are only generated for requests sent to the some domain. We discuss this further in V.

C. Economic Properties

Our approach also exhibits some very interesting economic properties. Namely, the initial registration step requires a write transaction to be sent to the blockchain which involves the payment of Gas (or similar transaction fees in other blockchains). In our implementation, this is performed by the service, and the encrypted claim object is sent from the service's web server to the chain.

This means that service operators have an economic incentive to ensure that users present a certain minimum economic value to the service, otherwise user registration leads to economic loss. We initially speculated that this characteristic threshold might lead to less low quality web services. Here, we mean low quality in the sense of "delivers low economic value". However, it's difficult to make any substantiated claims without further investigation into this property which is unfortunately beyond the scope of this paper. In fact, we propose that a more robust implementation that builds on our architecture presented here would actually "swap" the roles of the user and the service, with the user paying the required fee to commit the claim object to the blockchain. See V for more details.

D. Weaknesses of Our Technique

Our approach maintains a few characteristic weaknesses.

Our implementation doesn't validate the timestamps of the signature objects sent to the server, which means that a compromised signature object is valid *indefinitely*, and can be used to authenticate as the user by an attacker. However, this is solely a characteristic arising from the prototypical nature of our implementation. A more robust implementation would include various hardening mechanisms like short-lived timestamps and service-specific nonces applied to signature objects.

More intrinsically, our approach centers around providing the user with a manner to verify the security of the user data that the service has stored. Hence we allow the user to verify their own signature of the data which has been encrypted on-chain. However, in general, there is no method to ensure that

the service has not stored a copy, altered the data, or co-opted it for purposes not in the interest of the user. It's likely safe to assume that most services *would* in fact do so in order to cache the claim objects, as it is unattractive to query the blockchain to validate every request because it would require a network round trip or running a local blockchain node and for many services we can assume this is not feasible & presents an unnecessary development burden. Rather than attempt to combat this reality (in futility), we embrace it; some of our suggestions in V center around this point.

In this context we thus conclude that our work maintains many aspects ripe for refinement and further development.

V. Further Development

The most salient manner to further improve our proposed method is to build the capabilities directly into the browser-based signing authorities, or perhaps even directly into web browsers themselves. It's quite feasible that the process of sending the signature object to the server could be completely automated by the browser wallet or browser, a user would simply need to grant login access one time; thereafter the wallet could ensure that the signature object is sent with every request to the server. As mentioned previously, we propose that a mandatory "SameSite" functionality would provide the foundation for a very secure implementation and help combat CSRF-style attacks. Hardening the API and disallowing any direct JavaScript access would similarly reduce the attack surface for common XSS attacks targeting authentication details.

Of course, an unanswered question remains: How do services request cross-site resources directly from the client (i.e. the browser)? Server side rendering largely obviates the need for this use case, as the server is responsible for requesting third party content on the user's behalf and then sends it to the browser. However, for the case where client side cross-site requests are required, we hypothesise that it may be possible to integrate a "whitelist" of allowed cross-site domains which could be integrated into the third-party service's web server. The browser would be required to send an initial request to determine if the origin domain is included in this whitelist, if so, the browser sends the actual request, including the signature object[33].

Finally, we turn our attention to the aforementioned economic aspects of our authentication framework. We initially determined that the it should be the responsibility of the services to cover the transaction fees required to send the encrypted claim to the blockchain. This was done with the reasoning that the economic burden should not be placed onto the user; services should be required to cover this economic cost as an acknowledgement of the value of the data they are receiving.

However, we believe that a more sophisticated implementation may reverse this. In this scenario the users would encrypt the data and send it to the blockchain themselves. While this places the economic burden on the users, this ensures they have complete control over the data released to the service. If the scalability of modern blockchains continues to increase, we believe that this cost could be sufficiently minimised to the point where the self-custody of data becomes the standard and secure default.

[33]Readers with a web development background will notice this is conceptually identical to Cross-Origin Resource Sharing (CORS).

VI. Conclusion

We have investigated two innovative techniques that contribute to a novel web authentication workflow, namely:

1) Utilising the growth of browser-based signing authorities as a vector to introduce a more secure authentication workflow for the web based on public key cryptography with improved UX
2) Experimenting with the retention of user data on the blockchain to provide open verification of its secure storage and retrieval

We have seen that this authentication framework provides marked benefits over traditional password-based authentication mechanisms and is more secure by nature while offering a better user experience. We also determine that further development of this framework may help reduce the attack surface for common web attacks, namely CSRF and XSS.

We have introduced some of the interesting economic properties inherent our framework which dictate that some party in the registration process must bear the transaction fee required to store the data on the blockchain; we conclude that the ramifications of this are not fully explored and ripe for more investigation, and we propose that placing the economic burden on the user may allow them to fully control their data. Finally, we conclude that this authentication framework deserves continued research attention and possibly, direct integration into browser-based crypto wallets, or even browsers themselves.

Acknowledgment

The author would like to thank Professor Dr. Thomas Puschmann for the opportunity to write the course paper on this topic as well as allowing it to be written in the provided IEEE format.

References

[1] "Password manager and vault 2021 annual report: Usage, awareness, and market size." https://www.security.org/digital-safety/password-manager-annual-report/, Dec 2021.

[2] E. Bauman, Y. Lu, and Z. Lin, "Half a century of practice: Who is still storing plaintext passwords?," in *Information Security Practice and Experience* (J. Lopez and Y. Wu, eds.), (Cham), pp. 253–267, Springer International Publishing, 2015.

[3] L. Bošnjak, J. Sreš, and B. Brumen, "Brute-force and dictionary attack on hashed real-world passwords," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1161–1166, 2018.

[4] M. View, J. Rydell, M. Pei, and S. Machani, "TOTP: Time-Based One-Time Password Algorithm." RFC 6238, May 2011.

[5] K. Krol, E. Philippou, E. De Cristofaro, and M. A. Sasse, ""They brought in the horrible key ring thing!" Analysing the Usability of Two-Factor Authentication in UK Online Banking." https://arxiv.org/abs/1501.04434, 2015.

[6] O. Spaniol and D. Thießen, "Security in communication systems: Authentication lecture." http://www-i4.informatik.rwth-aachen.de/content/teaching/lectures/sub/sikon/sikonSS07/, April 2007.
[7] R. W. Shirey, "Internet Security Glossary, Version 2." RFC 4949, Aug. 2007.
[8] C. Bothwell, J. Faessler, and M. Gamba, "Group 07: ERC-721 NFT with ERC-20 & Off-Chain Signing / Report," tech. rep., University of Zurich, Faculty of Business, Economics and Informatics, Blockchain Programming Seminar, 2021.
[9] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: building smart contracts and DApps*. O'Reilly Media, Inc., 1st ed., 2018.
[10] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Authentication Protocol." RFC 4252, Jan. 2006.
[11] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519, May 2015.
[12] "Samesite cookies - http: Mdn." https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite, May 2022.
[13] R. Bloemen, L. Logvinov, and J. Evans, "EIP-712: Ethereum typed structured data hashing and signing, Ethereum Improvement Proposals, no. 712." https://eips.ethereum.org/EIPS/eip-712, 2017.

VII. Code

The full code implementation used and described by this work is available at: https://github.com/cmbothwell/blockchain-auth.