# auth.js: Advanced Authentication for the Web

Neophytos Christou[1][0000−0001−7335−9485] and Elias Athanasopoulos[2]

[1] Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678, Nicosia, Cyprus, nchris23@cs.ucy.ac.cy

[2] Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678, Nicosia, Cyprus, eliasathan@cs.ucy.ac.cy

**Abstract.** Several research works attempt to replace simple authentication schemes, where the cryptographic digest of a plaintext password is stored at the server. Those proposals are based on more elaborate schemes, such as PAKE-based protocols. However, in practice, only a very limited amount of applications in the web use such schemes. The reason for this limited deployment is perhaps their complexity as far as the cryptography involved is concerned. Today, even the most successful web applications use text-based passwords, which are simply hashed and stored at the server. This has broad implications for both the service and the user. Essentially, the users are forced to reveal their plain passwords for both registering and authenticating with a service.

In this paper, we attempt to make it easier for any web service to a) enable easily advanced authentication schemes, and b) switch from one scheme to another. More precisely, we design and realize `auth.js`, a framework that allows a web application to offer advanced authentication that leverages sophisticated techniques compared to typical cryptographically hashed text-based passwords. In fact, `auth.js` can be easily enabled in all web applications and supports traditional passwords – however, once enabled, switching to a more elaborate scheme is straight forward. `auth.js` leverages advanced cryptographic primitives, which can be used for implementing strong authentication, such as PAKE and similar solutions, by ensuring that all cryptographic primitives are trusted and executed using the browser's engine. For this, we extend Mozilla Crypto with more cryptographic primitives, such as `scrypt` and the edwards25519 elliptic curve. Finally, we evaluate `auth.js` with real web applications, such as WordPress.

## 1 Introduction

Authentication is vital for the majority of on-line web applications. Through the process of authentication, services can distinguish their users and offer dynamically generated and personalised content. Unfortunately, the authentication process is often an attractive target for attackers. The goal of attackers is to impersonate users by stealing their credentials and therefore have access to their data. Notice that, beyond accessing sensitive data, the attacker can also *generate* information on behalf of the compromised user [21] [6].

Several attacks exist depending on the way authentication is implemented. In the case of text-based passwords, it is common to salt, cryptographically hash, and store them at the server. The mechanics of the password protection, which is based on storing the password hashed at the server, coerces the user to *reveal* their plain password to the server each time they log in, which is very likely to already be used in other services, as well. A malicious server could then use the user's plain password to try to take control of another account of the same user in another service. This can be dramatically augmented due to password reuse [14], where users recycle passwords among different services. Other solutions that combat password reuse, like password managers that auto-generate strong passwords do exist, but unfortunately such solutions have not been thoroughly adopted [8]. Furthermore, the majority of other authentication schemes, like SSO services such as OAuth [15], still make use of plaintext passwords. These services do reduce password reuse, since users can authenticate to many services through a single SSO provider. However, they are still vulnerable to password reuse attacks. If a user registers to a malicious website by reusing the same password as the one they use for their SSO provider, the attacker could then use the password to authenticate as the user at all the other services in which the user is registered with their SSO provider.

On the other hand, advances in cryptography have developed all necessary tools for realizing protocols that do more than simply sending a string to be salted and hashed. For instance, several protocols for Password Authentication Key Agreement (PAKE) [10] permit a password to act as a seed for generating cryptographic keys. Regardless of the actual implementation, such schemes allow users to send a secret to the server for authenticating instead of the password in plain. The secret is cryptographically connected with the password and, therefore, even non-trusted servers must perform cracking attacks for revealing a user's password.

Despite the availability of such protocols, services continue to base their authentication on hashing plain passwords. An exception to this rule is Keybase [1], a service which offers cryptographic functions to users (for instance encrypted chat, filesystem and version control). Keybase assumes that the password (or *passphrase*, as they call it) of the user serves as a seed for generating a pair of keys that belong to an elliptic curve [11]. The private key is generated on the fly by the browser and allows the user to sign a message that is validated using the public key stored at the Keybase server. Thus, the password of the user is never revealed to Keybase, while complex handling of cryptographic keys is not an issue; the keys can be re-generated from the passphrase every time the user logs in (from any device).

Unfortunately, Keybase implements all this functionality, including the cryptographic operations, using its own code and does not use the browser's engine to do so. A web site may advertise that it supports a Keybase-like authentication process, where the password of the user is never revealed to the server, in order to convince users to register with it. However, unless the cryptographic primitives are executed in a secure context, it is unclear whether the aforementioned web site implements the authentication algorithm correctly or deliberately violates it in order to read the user's password.

In this paper, we build a framework for allowing any web site to offer advanced authentication, where plain passwords are used but are never exposed to any server. In particular, we design, implement and evaluate `auth.js`, an authentication framework with a JavaScript interface, which allows developers to enable any PAKE-like protocol in their apps. As a proof-of-concept, we use `auth.js` to enable Keybase-like authentication to WordPress with just a few code modifications. `auth.js` can be used through JavaScript, however, all cryptographic primitives are enforced by the browser engine, which we assume trusted. For this, we extend Mozilla Crypto with more cryptographic primitives, such as `scrypt` and the edwards25519 elliptic curve.

### 1.1 Contributions

To summarize, this paper contributes:

- we extend Mozilla Crypto with more cryptographic primitives, such as `scrypt` and the edwards25519 elliptic curve –although this is a solely engineering task, we consider it important for enabling new cryptographic capabilities for web applications;
- we design and realize `auth.js`, a framework that allows a web application to offer advanced authentication that leverages sophisticated techniques compared to typical cryptographically hashed text-based passwords;
- `auth.js` can be easily enabled in all web applications and supports traditional passwords – however, once enabled, switching to a more elaborate scheme is straight forward;
- we evaluate `auth.js` with real web applications, such as WordPress. Enabling `auth.js` in WordPress requires modifying about 50 LoCs of the main authentication code and adding 50 LoCs for enabling password recovery and signature validation.

## 2 Background

In this section, we briefly discuss some common authentication schemes supported by most web applications. `auth.js` can easily support all mentioned schemes, as well as more elaborate ones, such as PAKE protocols [10].

### 2.1 Conventional password authentication

The most common authentication scheme used in the web is text-based passwords. A general overview of how this scheme works is the following. Firstly, when a user registers a new account, they send their password over a (usually encrypted) channel to the web server. The web server uses a cryptographic hash function to compute the hash of the user's password and stores the hash, along with other information about the user, such as their username.

When the client wants to authenticate itself to the server, the user is prompted for their password and the password is sent back to the server. At the server, the hash of the password is computed again and compared against the stored hash. If the two hashes match, the authentication is successful and the user is logged in. For storing different cryptographic digests for identical passwords, the server often concatenates a random, non secret, *salt* to the plain password before hashing it.

### 2.2 Public key authentication

An alternative method is public-key authentication. This form is often combined with keys that are derived from a password, in order to simulate the typical text-based password experience. For this authentication scheme, the client does not send their password to the server that it wants to register to. Instead, it generates a key pair consisting of a public key, which is sent to the server, and a private key, which the client stores locally.

For authentication, the client informs the server that it wants to authenticate. The server then sends a message to the client and the client uses their stored private key to sign the message, in order to prove ownership of the private key. The signed message is sent back to the server, and the server verifies the signature using the stored public key of the user. If the verification is successful, the user is logged in.

### 2.3 Keybase authentication

Keybase [1] is a service which offers to its users the ability to prove their identity on social media platforms by mapping their profiles to generated encryption keys. It also offers end-to-end encrypted messaging between its users, an encrypted cloud storage system and other services. Keybase uses a public key authentication system which works as follows. When a new user tries to sign up [3], they firstly type in a password. However, the password is not directly submitted to the server. Keybase uses its signup API call to generate a random salt value and an `scrypt` hash is generated using the password and the salt. Some bytes of the generated hash value are interpreted as an EdDSA private key, which is then used as a seed to another function to generate the corresponding EdDSA public key. This public key is sent to the Keybase server and is stored as the user's credential. At the login phase [2], the EdDSA private key is recomputed similarly to the signup phase. In order to prove ownership of the key, the client recomputes the private key by prompting the user to re-type their password. Using this key, the client creates a signature which is verified by the server using the stored public key of the user.

## 3 Architecture

In this section we provide an overview of the architecture of `auth.js`, as well as the steps needed to be taken by the web application programmer in order to use the framework. We also provide an example of a use case where a server chooses to use an advanced authentication scheme based on public-key cryptography, and specifically based on the authentication scheme of Keybase described in Section 2, to register and authenticate its users. This scheme is referenced as `scrypt_seed_ed25519_keypair` by the `auth.js` API. The cryptographic primitives required to be performed for authentication and registration are handled on the client side by the `auth.js` framework, which uses the client's browser engine to ensure that the cryptographic operations are performed in a secure context.

## 3.1 Overview

`auth.js` provides simple API calls for the programmer that wants to use advanced authentication techniques in their web application, without needing to worry about the underlying implementation. This is especially important for the various cryptographic elements, which may be leveraged during authentication. First, the programmer does not need to re-implement any cryptographic primitives and, second, all primitives are enforced by the web browser, which we consider trusted.

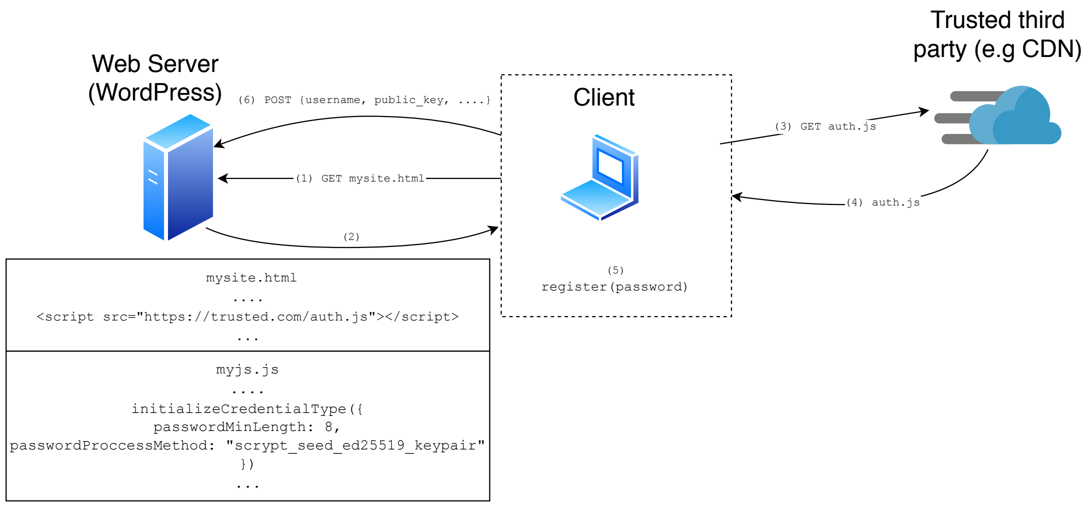
When a client requests a web application, the web server will direct the client to retrieve a copy of `auth.js`. The library can be provided to the client either by the web server directly, or via a trusted third party such as a Content Distribution Network, as seen in Figure 1. Ideally, a user can even pre-install `auth.js`, eliminating any need to retrieve it through the web for each authentication. After retrieving the library, the client is able to start the registration or authentication process. In particular, our library provides two API calls, `authenticate` and `register` that, when called, will use the client's browser Web Crypto API to perform the correct cryptographic operations depending on the chosen authentication scheme. For example, in the case of the `scrypt_seed_ed25519_keypair` scheme, the library will use the implemented `scrypt` hash function and the `Ed25519` key generation to create a key pair using the user's password. For authentication, it will use the generated private key to sign a nonce sent by the server using the `Ed25519` signature scheme, to prove ownership of the private key.

Our library currently supports traditional plain password authentication, as well as the more advanced public key authentication scheme based on the Keybase authentication. It can be extended to support any authentication scheme, as long as the browser supports the corresponding cryptographic primitives.

## 3.2 auth.js API

**Usage** Our JavaScript library provides an easy-to-use API that can be used by the web application programmer with minimal effort. The library will be used as follows:

– The server that wants to use our library includes `auth.js` in the web application's source.



```
mysite.html
....
<script src="https://trusted.com/auth.js"></script>
...
```

```
myjs.js
....
initializeCredentialType({
passwordMinLength: 8,
passwordProccessMethod: "scrypt_seed_ed25519_keypair"
})
...
```

**Fig. 1.** Overview of the architecture of `auth.js`. The client requests the web application from the server (1). The server responds by sending the html file, which directs the client to retrieve `auth.js` from a trusted third party, as well as with a JavaScript file in which the authentication options are initialized (2). The client then retrieves `auth.js` from the trusted source (3 and 4). The client's browser prompts the user for a password and the register API call from `auth.js` is used to generate the correct credential (5). In this case the generated credential is the user's public key, which is generated based on the password which the user provided. Finally, the credential is sent to the server (6), where it will be verified.

– The desired authentication options must be initialized by the web programmer using the `initializeCredentialType` API call in the main web application (e.g. in the JavaScript file served by the web server), as depicted in Listing 1.2. This call takes as an argument a JSON object describing the authentication options. The library currently supports two options. First, the `passwordMinLength` option allows the server to choose the minimum password length it can accept. The second option, `passwordProcessMethod`, enforces the use of one of the supported authentication schemes. The currently supported schemes are `plain`, which is the traditional text-based password and `scrypt_seed_ed25519_keypair`. If the `initializeCredentialType` call is not used, the library will use the default values of no minimum password length and the `plain` authentication scheme.
– After initializing the options, the `authenticate` and `register` calls can be used. Those calls are placed in the web application's JavaScript source by the web programmer, to be called when the user tries to perform a authentication or registration action. The `register` function takes as an argument the password which the user typed and returns the corresponding credential based on the chosen authentication scheme, to be sent to the server. The `authenticate` function also takes as an argument the user's password and,

in the case where an advanced public-key based authentication scheme is used, the optional message argument, which is the nonce that should be signed using the user's private key. The function generates the private key based on the password, signs the message if needed, and returns the signed message. In the case of the plain authentication scheme, the two functions simply return the user's password.
– The web application sends the generated credential to the server. If the authentication or registration is successful, the user can continue using the web application as usual.

**Example** In the following example, we depict how a server chooses to use the `scrypt_seed_ed25519_keypair` authentication scheme, with a minimum of 8 characters for the password. The web application HTML code directs the user to retrieve `auth.js` from a trusted source, as seen in Listing 1.1. The API calls of `auth.js`, `register` and `authenticate`, are then used to generate the correct credentials that the web application can now send to the server.

```
<html>
<head>
...
<script type = "text/javascript" src = "https://trusted.com/
    auth.js"></script>
<script type = "text/javascript" src = "myjs.js"></script>
...
</head>
<body>
/* Registration and login form */
</body>
</html>
```

**Listing 1.1. Web application html file.** The client is directed to get `auth.js` from a truted source.

```
  initializeCredentialType({
    passwordMinLength: 8,
    passwordProccessMethod: "scrypt_seed_ed25519_keypair",
  });
    let password = document.getElementById("password");
      /* On registration action */
    let credential = register(password);
      /* On login action */
    let message = document.getElementById("nonce");
    let credential = authenticate(password, message);
  /* Send credential and other necessary information to the
    server */
```

**Listing 1.2. Web application JavaScript file.** The minimum password length and authentication scheme are initialized. The register and authenticate API calls are called when a user tries to register to or authenticate with the server. auth.js generates the

correct credential based on the user's password, and the credential is then sent to the server along with other necessary information, such as the user's username

## 4 Implementation

Since modern web browsers do not yet provide support for the cryptographic primitives needed for offering advanced cryptographic capabilities, we extended Mozilla's Network Security Services, which is the set of cryptographic libraries used by Mozilla, to support the use of the `scrypt` cryptographic hash function, the creation of `Ed25519` public and private keys and the use of the `Ed25519` signature scheme. Firefox's Web Crypto API also needed to be extended, so as to enable the option to make use of the new cryptographic primitives through the browser. By adding those capabilities, the client does not need to rely on untrusted external sources to perform the aforementioned cryptographic operations, since their own browser's engine executes the cryptographic primitives in a secure context.

### 4.1 Extending Mozilla's Network Security Services

**Adding the `scrypt` cryptographic hash function** We added a new cryptographic hash function based on the implementation of `scrypt` taken from Tarsnap [5] into the NSS. The new function is added in NSS similarly to other existing cryptographic hash functions, such as the implementation of `SHA256`. An example of how the new `scrypt` works, along with the existing `SHA256`, is depicted in Listing 1.3.

```c
void
SHA256_End(SHA256Context *ctx, unsigned char *digest,
           unsigned int *digestLen, unsigned int maxDigestLen)
{
    unsigned int inBuf = ctx->sizeLo & 0x3f;
    unsigned int padLen = (inBuf < 56) ? (56 - inBuf) : (56 +
    64 - inBuf);
    ...
    /* SHA256 implementation */
}
void
SCRYPT_End(SCRYPTContext *ctx, unsigned char *digest,
           unsigned int *digestLen, unsigned int maxDigestLen)
{
    /* Set scrypt parameters */
    _crypto_scrypt(...);
}
```

**Listing 1.3.** sha512.c in Mozilla's NSS implementation, which contains the implementation of existing hash functions. SCRYPT_End calls the _crypto_scrypt function (part of the Tarsnap scrypt implementation) to perform the hashing.

**Adding the `Ed25519` EdDSA signature scheme** In a similar fashion, we added support for the Ed25519 signature scheme. In particular, we added the functionality to create a public-private key pair based on a given seed, as well as the signing functionality of the scheme. For this cryptographic primitive, we used parts of the SUPERCOP benchmarking tool's implementation of Ed25519 [4].

## 4.2 Extending Mozilla's `Web Crypto API`

Apart from extending the NSS library, we also needed to extend Mozilla's Web Crypto API, in order to enable the use of the newly added cryptographic primitives through JavaScript API calls. Similarly to the NSS extension, we located the files containing the calls to other cryptographic primitives and extended them to also provide calls to the newly added operations. With this addition, the client's browser can use the Web Crypto API to perform password hashing using the `scrypt` hash function, as shown in Listing 1.4, generate `Ed25519` keys and sign messages using those keys.

```
const encoder = new TextEncoder();
//Get scrypt hash of password
const passwordEncoded = encoder.encode(password);
const hashScrypt = crypto.subtle.digest("SCRYPT",
    passwordEncoded);
```

**Listing 1.4.** `scrypt` hash function called from Firefox using Mozilla's Web Crypto API.

## 4.3 WordPress

WordPress is one of the most popular open-source web management systems. It is written in PHP and is widely used for building various websites, ranging from simple blog spots to professional websites. Since it is open-source, we modified the source code to incorporate our authentication and registration system, by extending the current WordPress functionality.

The current default login and registration system of WordPress works as follows. When users wish to register to the website, they provide their user name and email. The user then receives an email with what is essentially link to a reset password form, where they can set their first password. After the user chooses a password, it is sent to the server, where it is salted and hashed with the MD5 hash function and stored.

At the login phase, the user fills in their user name or email and their password in the login form, which is submitted to the server. There, the hash of the submitted password is checked against the stored hashed password and, if they match, the user is logged in.

A web developer that wishes to use `auth.js` in a WordPress site can do so by making minor tweaks to the WordPress source code. The number of changes needed to be made depend on the authentication scheme that is chosen to be used. Simply adding `auth.js` in a WordPress website that wishes to continue using its current authentication system is as simple as adding a few lines of code, while switching to the public key authentication scheme requires some extra steps, such as the addition of a few more functions using the hooks provided by WordPress, in order to extend the functionality of the authentication system. Both of the aforementioned additions are demonstrated below.

**Using `auth.js` with the current WordPress authentication system** A web developer can choose to add `auth.js` to a WordPress website without wishing to change the default authentication scheme. To do so, the following steps are required:

– Include `auth.js` in the list of the scripts which are loaded along the log in and reset password pages. Note that as discussed in Section 3, this could also be done by loading the file from a trusted third party, such as a CDN.
– Modify the log in and reset password form to make `auth.js` intervene before the form submission, in order to change the typed user password to the corresponding credential for the chosen authentication method. Even though no modification will be made on the password field when the `plain` (default) authentication scheme is chosen, adding this will make it easier to switch between authentication schemes in case the web developer wishes to change to a more advanced authentication scheme in the future.

Adding the `auth.js` file can easily be done using the `login_enqueue_scripts` hook provided by WordPress, as shown in Listing 1.5. This should be added in the `wp-login.php` file, which handles the login, reset password and registration forms.

```php
add_action( 'login_enqueue_scripts', 'enqueue_authjs' );

function enqueue_authjs( $page ) {
	wp_enqueue_script( 'auth', home_url() . '/wp-includes/js/auth.js', null, null, true );
}
do_action( 'login_enqueue_scripts' );
```

**Listing 1.5.** Using the `login_enqueue_scripts` hook to enqueue auth.js.

To modify the reset password form, a script that temporarily stops the form submission must be added. We demonstrate how this can be done using JQuery in Listing 1.6. The minimum password length and authentication scheme must be initialized using the `initializeCredentialType` call. Before eventually submitting the form, the script uses the `auth.js` API to generate the correct credential and change the credential value which will be submitted. Similarly to the reset password form, a script can be added to change the submitted password value on the login form. In the case of the `plain` authentication scheme, the typed password length is checked and the password is submitted as is.

Both the reset password and log in form scripts can be saved in the site's resources in the `wp-includes/js` folder and enqueued in the same way the `auth.js` file is enqueued, by including them in a JavaScript file in the website resources and then using the `login_enqueue_scripts` hook in the `wp-login.php` file.

**Using `auth.js` with the public key authentication scheme** In order to switch to the more advanced public key authentication scheme, the following additional steps must be made, apart from the steps described above:

- Whenever the `initializeCredentialType` is used to set the options for the credential generation, use `scrypt_seed_ed25519_keypair` as the value for the `passwordProccessMethod` field.
- Modify the login form to include a random token that will be utilized as a nonce and get signed with the user's private key in order to perform authentication.
- Add the same nonce as a cookie that will be submitted along with the form, in order for the server to have the original value of the nonce and be able to verify the signature.
- Modify the default authentication check of WordPress to make it verify the submitted signed nonce using the stored public key.

```
jQuery("#resetpassform").on("submit", function (e) {
     e.preventDefault(); //Stop form submission
     let self = jQuery(this);
     initializeCredentialType({
      passwordMinLength: 8,
      passwordProccessMethod: "plain",
     });
     let password = jQuery("#pass1").val();
     let public_key = register(password); //Generate the
     credential using auth.js
     public_key.then( (pk) => {
          console.log(pk);
          jQuery("#pass1").val(pk); //Set the new credential
     value to be submitted
          jQuery("#pass2").val(pk);
          jQuery("#resetpassform").off("submit");
          self.submit();//Submit the form
     })
});
```

**Listing 1.6.** JavaScript code that uses `auth.js` API to generate the credential and submit the reset password form

To use the public key authentication scheme in the log in and reset password forms, the `passwordProccessMethod` field seen in Listing 1.6 needs to be changed to `scrypt_seed_ed25519_keypair`. When this authentication scheme is chosen, the `register` API call of `auth.js` will use the browser's Web Crypto

API and perform the necessary cryptographic operations to change the value of the typed password to the corresponding Ed25519 public key, which is generated using the `scrypt` hash of the password as a seed. The log in script will use the `authenticate` API call to sign the nonce placed in the login form using the private key corresponding to the public key mentioned earlier. The submitted value will be the public key concatenated with the generated signature. Note that the server must have a way to get the original value of the cookie, in order to be able to verify the signature.

Next, the nonce that will be utilized as a message and get signed using the user's private key needs to be added. A simple way to do so is to generate a nonce on the server and attach this nonce in a hidden field in the login form and also add the same value as a cookie. This way, the server does not need to keep the state of each session, since the original value of the nonce before it was signed can be retrieved from the cookie. This addition is demonstrated in Listing 1.7 and should again be made in the `wp-login.php` file.

```
# Create nonce and set it as a cookie
$token = bin2hex(openssl_random_pseudo_bytes(16));
setcookie("nonce-message", $token, time() + 60 * 60 * 24);
...
# Add the nonce as a hidden field in the login form
<input type="hidden" id="nonce-message" name="nonce-message"
    value="<?= $token ?>" />
```

**Listing 1.7.** Add a nonce as a cookie, as well as in the log in form as a hidden field

```
function wp_authenticate_username_password( $user, $username,
    $password ) {

  if ( ! wp_check_password( $password, $user->user_pass, $user
    ->ID ) ) {
     return new WP_Error(
       'incorrect_password',
       sprintf(
         /* translators: %s: User name. */
         __( '<strong>ERROR</strong>: The password you entered
    for the username %s is incorrect.' ),
         '<strong>' . $username . '</strong>'
                    ...
     }
...
}
```

**Listing 1.8.** wp_authenticate_username_password, one of the default authentication functions used in WordPress

Finally, the authentication check in the WordPress server side needs to be modified. To do this, the `authenticate` hook can be used to add a new function to authenticate the user. This hook should be added in the `default-filters.php` file, in the `wp-includes` folder. We added the new user authentication function, called `authjs_authenticate`, in the `user.php` file. `authjs_authenticate` functions similarly to the default authentication functions[3] used by WordPress, except that, for checking the user's credentials, it does not call the default `wp_authenticate_email_password` function. Instead, it calls a new function called `check_public_key`. The differences between the two functions can be seen in Listings 1.8 and 1.9.

```
function authjs_authenticate( $user, $username, $password ) {
...

  if ( !check_public_key( $password, $user->user_pass, $user->ID ) ) {
    return new WP_Error(
      'incorrect_public_key',
      sprintf(
        __( '<strong>ERROR</strong>: Wrong public key' ),
    }
}
```

**Listing 1.9.** The `authjs_authenticate` function which is used in place of the default authentication function of WordPress

The `check_public_key` function is added in the `pluggable.php` file. Listing 1.10 shows how `check_public_key` verifies that the submitted signature is correct. In particular, it parses the received credentials to get the public key and signature values and checks if the hash of the public key submitted by the user matches the stored public key hash. Then, it uses the submitted signature along with the Ed25519 public key and the original nonce value to verify the signature. We implemented this check as an external Python script, which uses the PyNaCl library to verify that the given signature is correct. After the signature is verified, the user is successfully logged in.

```
        // Get the original value of the nonce from the cookie, so we can verify the signature
        $message = $_COOKIE["nonce-message"];
        $public_key = substr($credentials, 0, 64);
        $signature = substr($credentials, 64);
        // Check if the user sent the correct public key
        $check = hash_equals( $stored_pk, md5( $public_key ) );
        /* Run python script to verify signature */
        ...
        return apply_filters( 'check_password', $check, $credentials, $stored_pk, $user_id );
```

**Listing 1.10.** The `check_public_key` function that verifies the submitted signature using the user's stored public key

[3] To be precise, WordPress has three default authentication methods: one using username and password, one using email and password and one using a cookie.

# 5 Evaluation

In this section we evaluate the performance of `auth.js` and particularly the overhead that the public key authentication system adds over the traditional password authentication method.

## 5.1 Setup

For the following measurements, we used two Linux machines running Ubuntu 18.04 LTS. The first machine run a dummy server with minimal functionality. The second machine run a fork of Mozilla Firefox Nightly 73.0a1, compiled with the disable optimizations and enable debug options.

## 5.2 Average time for posting credentials on the server and getting a reply

We measured the average time for generating and posting a user's credentials using the two authentication methods, traditional password authentication and public key authentication, from the machine running Firefox to the machine running the dummy server. For checking the password, the dummy server simply checked if the posted password matched the user's stored password in its database. For checking the posted signature, the server run the Python script mentioned in Section 4. Table 5.2 presents the average time for 1,000 repetitions.

**Table 1.** Average time for posting key pairs and signatures.

| Credential posted | Average time |
|---|---|
| Password | 260 ms |
| Signature | 328 ms |

## 5.3 Average time for key pair and signature generation

We measured the performance of `auth.js` for creating Ed25519 key pairs and signing messages using the private key of the pair. We split the measurement in 3 parts: the time for only generating key pairs with a given password, the time for only signing a given message with a given key pair, and the time for both generating a key pair using a given password and signing a given message with the generated private key. Table 5.3 presents the average time for these three measurements for 10 thousand repetitions.

**Table 2.** Average time for generating key pairs and signatures.

| | Average time |
|---|---|
| Generate key pair | 30.9 ms |
| Sign message | 29.5 ms |
| Generate key pair + sign message | 59.3 ms |

## 6 Related Work

### 6.1 Advanced authentication schemes

Apart from the public key authentication scheme we presented, various more authentication methods exist. PAKE protocols such as SRP [22] allow clients to authenticate themselves to a server and exchange a secret securely, without needing to send their actual password. Even though certain PAKE protocols have seen some adoption, many of them have not been successfully deployed yet. Other password-based authentication mechanisms which are based on PAKE protocols, such as [23], are also starting to get proposed. `auth.js` can serve as a single framework from which such protocols can be deployed. As long as the cryptographic primitives needed for a protocol are implemented in the client's browser, `auth.js` can securely enforce their usage, assuming of course that the browser is not compromised. A web programmer who wishes to use another scheme for authenticating users can do so simply by changing the `passwordProccessMethod` field in their forms to the authentication scheme of their choosing and transparently switch to a new authentication method, assuming that the server also supports the use of a chosen protocol. The autentication scheme mentioned in this paper is based on the authentication scheme used by Keybase [1]. The major difference is that Keybase uses its own source code to perform the cryptographic operations, while `auth.js` uses the cryptographic primitives that are built in the user's browser, ensuring that the operations will be performed securely.

### 6.2 Cryptographic primitives

In the recent years, many improvements have been made and many new cryptographic primitives have been introduced, which are not yet implemented by the major web browsers. For our work, we added the scrypt [18] hash function as well as the Curve25519 elliptic curve [11] to Mozilla Firefox and specifically in the Web Crypto API, in order to use them for our authentication scheme. We expect that those cryptographic primitives, as well as more primitives such as the bcrypt [19], Argon2 [12] and blake2 [9] hash functions or new elliptic curves such as the FourQ curve [13] will eventually be implemented in the major web browsers and will be available to use. As more and more cryptographic primitives are added, `auth.js` can be modified to support the usage of these primitives to create new authentication schemes. Other projects have also explored the extension of the Web Crypto API functionality to add support for other operations, such as document signing [16]. New types of cryptographic primitives are also starting to get implemented. For example, Microsoft's SEAL [20] provides an API that can be used to perform homomorphic encryption.

### 6.3 Cryptography frameworks

Other frameworks have also tried making advanced cryptography more accessible and easier to use. For example, Let's Encrypt [7], [17] makes it easy to obtain a TLS certificate without the need of human intervention. Keybase is another web service that offers advanced cryptography to simple users, such as an advanced authentication scheme, end-to-end encryption, public identity verification and encrypted storage.

## 7 Conclusion

In this paper we designed, implemented and evaluated `auth.js`, a framework that allows web developers to integrate any authentication scheme in their applications. `auth.js` allows a developer to express the authentication policy in JavaScript and realize complex schemes, that leverage modern cryptographic primitives, in the browser environment. Moreover, the framework makes sure that cryptographic operations are not implemented in JavaScript, but are instead carried out using the browser's internal engine, which is considered trusted. For this, we extended Mozilla Crypto with the `scrypt` hash function and the edwards25519 elliptic curve in order to easily implement the authentication used in Keybase. In the same fashion, `auth.js` can support other cryptographic-based authentication schemes, such as PAKE. Enabling `auth.js` in existing web application is trivial and, once the framework is in place, switching from one authentication to another is straight forward. For demonstrating this, we enabled `auth.js` in a popular open-source web application, namely WordPress. Our modifications do not exceed 50 LoCs for the main authentication code in WordPress and require additionally 50 LoCs for enabling password recovery and signature validation.

**Acknowledgements** We thank the anonymous reviewers for helping us to improve the final version of this paper. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct), No. 830929 (CyberSec4Europe), and No. 826278 (SERUMS), and by the RESTART programmes of the research, technological development and innovation of the Research Promotion Foundation, under grant agreement ENTERPRISES/0916/0063 (PERSONAS).

## References

1. Keybase.io. https://keybase.io/.

2. Keybase.io login api documentation. https://keybase.io/docs/api/1.0/call/login.
3. Keybase.io signup api documentation. https://keybase.io/docs/api/1.0/call/signup.
4. Supercop benchmarking tool. https://bench.cr.yp.to/supercop.html.
5. Tarsnap scrypt 1.3.0. https://www.tarsnap.com/scrypt/scrypt-1.3.0.tgz.
6. S. Abu-Nimeh, T. Chen, and O. Alzubi. Malicious and spam posts in online social networks. *Computer*, 44(9):23–28, 2011.
7. M. Aertsen et al. How to bring https to the masses? measuring issuance in the first year of let's encrypt. 2017.
8. N. Alkaldi and K. Renaud. Why do people adopt, or reject, smartphone password managers? 01 2016.
9. J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. Blake2: simpler, smaller, fast as md5. pages 119–135, 06 2013.
10. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE, 1992.
11. D. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. volume 2, pages 124–142, 09 2011.
12. A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 292–302, 2016.
13. C. Costello and P. Longa. Fourq: four-dimensional decompositions on a q-curve over the mersenne prime. 06 2015.
14. S. Gaw and E. W. Felten. Password management strategies for online accounts. In *Proceedings of the Symposium on Usable Privacy and Security*, SOUPS, 2006.
15. E. Hardt, D. The OAuth 2.0 Authorization Framework. Internet Requests for Comments, October 2012.
16. N. Hofstede and N. V. D. Bleeken. Using the w3c webcrypto api for document signing, 2013.
17. A. Manousis, R. Ragsdale, B. Draffin, A. Agrawal, and V. Sekar. Shedding light on the adoption of let's encrypt. *CoRR*, abs/1611.00469, 2016.
18. C. PERCIVAL. Stronger key derivation via sequential memory-hard functions. 01 2009.
19. N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
20. Microsoft SEAL (release 3.4). https://github.com/Microsoft/SEAL, Oct. 2019. Microsoft Research, Redmond, WA.
21. K. Thomas, C. Grier, D. Song, and V. Paxson. Suspended accounts in retrospect: An analysis of twitter spam. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, page 243–258, New York, NY, USA, 2011. Association for Computing Machinery.
22. T. D. Wu et al. The secure remote password protocol. In *NDSS*, volume 98, pages 97–111. Citeseer, 1998.
23. Z. Zhang, Y. Wang, and K. Yang. Strong authentication without temper-resistant hardware and application to federated identities. 01 2020.