Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Review- & Crypto-Audit Report WEaR Library 07.2022

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi

Index

Introduction

"This library has a small set of features to support encryption-at-rest in web apps. Encryption-at-rest for a web app means that sensitive app data is encrypted before it is written to disk or other persistent storage. And when your web app needs to use that same data, it is decrypted and kept in memory."

From https://github.com/erikh2000/web-enc-at-rest/blob/main/README.md

This report describes the results of a cryptographic review and source code audit against the Web Encryption at Rest (WEaR) library written in JS/TS. The work was requested by Erik Hermansen, the maintainer of the WEaR library, in July 2022. Cure53 carried out the project during the same month, namely in CW29.

The assessment was documented as *WER-01*. As for the specific budget and resources, a total of two-and-a-half days were invested to reach the coverage expected for this project. Furthermore, it should be noted that a team of two senior testers were assigned to this project's preparation, execution and finalization.

To make sure that all work can be tracked and comprehensively documented, a work package (WP) was delineated as:

- **WP1**: Cryptography reviews and audits against Web Encryption at Rest (WEaR) library

Given the access level and availability of the material, this *WER-01* project was conducted with white-box methods. More precisely, Cure53 was provided with URLs, sources, as well as all further access and information required to complete the examination.

All preparations were done in July 2022, namely in CW28, so as to make sure that the Cure53 testers can benefit from a smooth start into the testing phase. Communications during the test were done using a dedicated shared Slack channel set up between Erik Hermansen and Cure53. All involved personnel from both parties could take part in the test-relevant discussions on Slack.

The communications proceeding on Slack were very efficient. Not many questions had to be asked because the scope was well-prepared and clear. Hence, no noteworthy roadblocks were encountered during the test.

Cure53 provided frequent status updates about the test's progress and the related findings. Live-reporting was not specifically requested and also not seen as necessary given the manageable severity markers ascribed to the findings.

It should be stated that the Cure53 team managed to get very good coverage over the targets. Four security-relevant discoveries were made, yet all of them were classified to be general weaknesses with lower exploitation potential. In other words, no vulnerabilities have been identified.

On the one hand, due to the rather compact scope examined during *WER-01,* it was expected that the number of issues spotted within this review should be small. On the other hand, the result of only four findings can still be interpreted as pointing to a good state of security characterizing the WEaR library. This is also evidenced by the fact that all of the Cure53 findings were ranked with either *Low* or *Info* severity-marker. In addition, the documented general weaknesses should mostly be trivially easy to address and resolve.

The report will now shed more light on the scope and test setup, alongside providing some more details on the material available for testing. After that, the report will list all general weaknesses in chronological order. Each finding will be accompanied with a technical description, a PoC where possible, as well as mitigation or fix advice.

The report will then close with a conclusion in which Cure53 will elaborate on the general impressions gained throughout this test. Additional, tailored recommendations on further improving the security posture of the Web Encryption at Rest (WEaR) library are also provided.

Scope

- **Cryptography reviews & Code audits against Web Encryption at Rest (WEaR) library**
    - **WP1**: Cryptography reviews & Audits against Web Encryption at Rest (WEaR) library
        - The library offers an API that allows developers to encrypt web application data at rest.
        - The audit focused on the cryptographic guarantees the library makes, especially the creation of "user-accounts", handling of encrypted data protected by these accounts, as well as login & logout processes
    - **NPM URL**
        - https://www.npmjs.com/package/web-enc-at-rest
    - **GitHub URL**
        - https://github.com/erikh2000/web-enc-at-rest#readme
    - **Audited commit**
        - 4192ca66925c6f1c12ad39f279c7ce521c61bf22
    - **Test-supporting material was shared with Cure53**
    - **All relevant sources were shared with Cure53 and/or were available as OSS**

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

WER-01-001 Crypto: Minor improvements to key derivation logic *(Low)*

WEaR's key derivation logic employs PBKDF2 in order to generate credential keys. The passphrase input of PBKDF2 is calculated by concatenating the user's username and password together with a *null* byte as a separator.

While this is not likely to be the source of any problems, it may be sounder to fully separate the input of the username and the password instead. It would be advisable to avoid the use of an arbitrary concatenation byte.

The proposed change may be accomplished by concatenating the username as part of the PBKDF2 salt instead of part of the PBKDF2 passphrase. The password (or an unkeyed hash of the password) could simply be used as the basis for the PBKDF2 passphrase.

Affected file:
src/keyGen.ts

Affected code:
```
function _concatPassphraseFromCredentials(userName:string,
password:string):string {
 return `${userName}\u0000${password}`;
}

const DERIVE_KEY_ITERATIONS = 100000;
async function _generateCredentialKeyBytes(subtle:any, userName:string,
password:string):Promise<Uint8Array> {
 const passphrase = _concatPassphraseFromCredentials(userName, password);
 const salt = getOrCreateDeriveKeySalt();
 const passphraseUint8:Uint8Array = stringToBytes(passphrase);
 const algorithmParams:Pbkdf2Params = { name: 'PBKDF2', hash: 'SHA-256', salt,
iterations:DERIVE_KEY_ITERATIONS };
 const baseKey:CryptoKey = await subtle.importKey('raw', passphraseUint8,
'PBKDF2', false, ['deriveKey']);
 const derivedParams:AesKeyGenParams = { name: 'AES-GCM', length: 128 };
 const credentialKey = await subtle.deriveKey(algorithmParams, baseKey,
derivedParams, true, ['decrypt', 'encrypt']);
 return new Uint8Array(await subtle.exportKey('raw', credentialKey));
}
```

Including the username as part of the salt also helps address concerns related to salt reuse. The WEaR documentation states that the salt reuse is “necessary” in order to allow the same credentials to consistently derive the same keys every time a value is encrypted. Including the username in the salt does not prevent this whilst also allowing for unique salts across usernames.

WER-01-002 Crypto: *Tamper detection* cryptographic API ineffective (*Info*)

WEaR includes a *tamper detection* mechanism which tries to detect whether the WebCrypto API core functionality has been overwritten after the module was loaded.

Due to the malleable nature of the JavaScript runtime, this sort of tamper detection is unlikely to yield any effective results, especially against attackers who are aware of its existence. Being able to override the WebCrypto API namespace implies concurrent capacity to override the namespace occupied by the WEaR functionality itself, as well as other namespaces across the execution environment.

While leaving the *tamper detection* functionality intact is unlikely to cause harm, removing it may reduce the overall technical debt. This would happen at no cost to the effective security or performance of the cryptographic library.

WER-01-003 Crypto: Native Base64 parsing advantages (*Info*)

WEaR provides its own Base64 encoding and decoding functionalities. While these appear to be soundly implemented, utilizing built-in browser functionality in order to convert byte arrays to Base64 and vice-versa may reduce the overall technical debt present in the project. It furthermore reduces the potential for the introduction of future software errors.

A *Uint8Array* may be converted to a Base64 string (and vice versa) using much simpler JavaScript code which takes advantage of the built-in functionality:

```
arrayToBase64String(a) {
	return btoa(String.fromCharCode(...a));
}

base64StringToArray(s) {
	let asciiString = atob(s);
	return new Uint8Array([...asciiString].map(char => char.charCodeAt(0)));
}
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

WER-01-004 Crypto: Advantages of migrating password hashing to *Scrypt* (*Low*)

WEaR utilizes PBKDF2 as the password-based key derivation function. This function is configured with 100.000 iterations and with SHA-256 as the underlying algorithm. On the one hand, this particular usage of PBKDF2 is relatively unlikely to present significant security issues in the future and the employed PBKDF2 parameters are relatively strong. On the other hand, PBKDF2 has been vulnerable to increasingly sophisticated attacks enabled by specialized hardware since 2014[1][2][3] onwards. The high availability of GPU-powered and ASIC-powered hashing can also be leveraged to instigate an attack on PBKDF2.

Affected file:
src/keyGen.ts

Affected code:
```
const DERIVE_KEY_ITERATIONS = 100000;
async function _generateCredentialKeyBytes(subtle:any, userName:string,
password:string):Promise<Uint8Array> {
 const passphrase = _concatPassphraseFromCredentials(userName, password);
 const salt = getOrCreateDeriveKeySalt();
 const passphraseUint8:Uint8Array = stringToBytes(passphrase);
 const algorithmParams:Pbkdf2Params = { name: 'PBKDF2', hash: 'SHA-256', salt,
iterations:DERIVE_KEY_ITERATIONS };
 const baseKey:CryptoKey = await subtle.importKey('raw', passphraseUint8,
'PBKDF2', false, ['deriveKey']);
 const derivedParams:AesKeyGenParams = { name: 'AES-GCM', length: 128 };
 const credentialKey = await subtle.deriveKey(algorithmParams, baseKey,
derivedParams, true, ['decrypt', 'encrypt']);
 return new Uint8Array(await subtle.exportKey('raw', credentialKey));
}
```

To mitigate this issue, it is recommended to eventually migrate to the *Scrypt* password hashing function. *Scrypt* is a competing password hashing function that offers superior security to PBKDF2 at the same password margins. Crucially, *Scrypt* is not susceptible to the attack scenarios that affect PBKDF2 due to the fact its security is based on memory-hardness.[4]

[1] https://link.springer.com/chapter/10.1007/978-3-319-26823-1_9
[2] http://mcs.csueastbay.edu/~lertaul/PBKDFBCRYPTCAMREADYICWN16.pdf
[3] https://www.usenix.org/conference/woot16/workshop-program/presentation/ruddick
[4] https://eprint.iacr.org/2016/989

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

It can be concluded that the WEaR library exposes a strong security posture. This Cure53 summer 2022 project demonstrated that only minor improvements can be considered and implemented to further increase the soundness of the WEaR library's security premise.

More broadly, the WEaR cryptographic library provides a minimalist set of features to support encryption-at-rest in web applications. Its design goals aim to keep it low-risk, light-weight and easy to understand. Thus, the scope targeted by Cure53 in this project was highly specific.

Cure53 was tasked with a full cryptographic audit of the WEaR library. Given the library's exceptionally simple, common-sense design and small codebase, the audit was completed quickly and without outstanding issues. This was further assisted by detailed code documentation, which ensured that the audit could be completed in the timely manner.

Within this *WER-01* project, Cure53 closely examined key derivation, encryption, decryption, authentication, encoding, decoding, storage, as well as all utility functionalities included as part of the WEaR library. Two testers from the Cure53 team were unable to detect any outstanding vulnerabilities in the WEaR library during this July 2022 audit. However, the examination has indicated several miscellaneous issues in the WEaR library that could be considered to ameliorate the overall soundness of the design.

Among areas where improvements may be made were the optional changes to the key derivation logic (WER-01-001) and the possible need to reconsider the currently inefficient *tamper detection* features in WEaR (WER-01-002). In addition, WER-01-003 discusses the lack of necessity for custom Base64 parsing and serializing code. Finally, WER-01-004 proposes an eventual migration to more future-proof password-based key derivation primitives. In conclusion, the WEaR cryptographic library appears to soundly implement cryptographic constructions. Considering the findings of this July 2022 audit, Cure53 believes that the WEaR cryptographic library is suitable for production use.

Cure53 would like to thank Erik Hermansen, the library maintainer, for his excellent project coordination, support and assistance, both before and during this assignment.