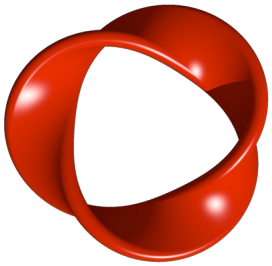

Secfault Security

Bare Pentest

Security Assessment

Report

for

Tether Operations, S.A. de C.V

Colonia Escalón, 87 Avenida Norte, Calle el Mirador, Edificio Torre Futura, Oficina 06, Nivel 11
San Salvador Tether

- hereafter called "Tether" -

Document History

Version	Author	Date	Comment
0.1	Gregor Kopf	2026-06-05	First Draft
0.2	Maik Münch	2026-06-08	Additions
0.3	Dirk Breiden	2026-06-09	Internal Review
0.4	Maik Münch	2026-06-19	Retest
0.5	Gregor Kopf	2026-06-24	Updated Retest Status
1.0	Maik Münch	2026-07-14	Final Report

Table of Contents

1 Executive Summary

Tether tasked Secfault Security to perform a code audit of parts of their minimal and modular JavaScript runtime called Bare.

Bare is a small kernel system designed highly modular. Therefore, Bare is a critical key component of the Tether environment in terms of security. The scope of this engagement included a number of modules with specific priorities assigned by Tether. A complete overview of the scope of this project is described in section 2.1.1 of this document.

Given that all the source code is publicly available, the review was conducted following a white-box approach using static analysis. Section 2.1.2 provides a description of the test procedures applied.

The review has been performed in the time frame from 2026-05-26 to 2026-06-09 in eight (8) person days including retest activities and documentation.

During project execution, a number of mostly medium severity issues have been identified throughout the modules in scope. No critical issues have been identified during project execution. The identified issues range from cryptographic issues such as weak authentication tag lengths or incomplete validations of lengths over integer related issues such as overflows to more general issues such as the use of assertions instead of runtime checks for return values. The details of the corresponding results are documented in detail in section 4 of this document.

Additionally, a general observation regarding uninitialized variables was documented in section 5 of this document.

Tether addressed all issues during project execution. Secfault Security validated all of the respective fixes and considers them to address the corresponding issues. Please note that retesting activities were not part of the engagement, but performed by Secfault Security whenever possible.

Overall, Tether 's solution left a positive impression on Secfault Security. The reviewed areas did not reveal any significant security issues.

Secfault Security would like to thank Tether for the excellent support before and during the security assessment.

2 Overview

The following sections provide an overview of the project execution, the scope of the assessment as well as a brief summary of the test procedures applied during the engagement.

2.1 Project Description

Tether is developing and maintaining Bare, a minimal and modular JavaScript runtime designed for building high-performance apps across desktop and mobile. Tether requested consultation on some the key modules offered by Bare to improve their robustness in terms of security.

The aim of this engagement was to perform a source code review of these key modules to identify common vulnerabilities. Therefore, Tether and Secfault Security agreed on a set of modules and a prioritization.

2.1.1 Target Scope

In scope of this engagement is the Bare solution developed by Tether. Bare is Tether foundational runtime for most of the running code/apps. Bare is a small kernel system, that is highly modular. Therefore, Bare is a critical solution of Tether 's environment in terms of security.

During the project setup communication the following scope and tasks was agreed between Tether and Secfault Security:

- Highest Priority
  - `https://github.com/holepunchto/bare-crypto`
    - All C code - full review
    - JS binding interfaces - the parts that call into C
    - Sensitive data paths - anywhere keys, passwords, auth tags flow through
- High Priority
  - `https://github.com/holepunchto/bare-tls`
    - All TLS/SSL implementation code
    - All C bindings for BoringSSL calls
    - Memory safety and correct BoringSSL API usage, NOT authentication
    - Cipher suite negotiation, Verify no downgrade attacks possible
    - Information Disclosure
    - Session management and resumption
  - `https://github.com/holepunchto/bare-tcp`

    - Focus on C bindings - memory safety, buffer handling
    - Connection state management and race conditions during async operations
    - Input validation before native calls
    - Resource cleanup in error paths and edge cases (especially during reset/destroy)
- Medium Priority
  - `https://github.com/holepunchto/bare-http1`
    - Request smuggling
    - Header and request/status-line injection
    - Connection lifecycle management
    - Agent and socket-pool integrity
    - Resource exhaustion
    - Integration with `bare-tcp`
    - Error handling and cleanup on malformed requests/responses
  - `https://github.com/holepunchto/bare-http-parser`
    - Request smuggling prevention
    - Header injection protection
    - Content-length vs Transfer-Encoding handling
    - Chunk encoding validation and resource limits
  - `https://github.com/holepunchto/bare-url`
    - URL parsing and validation (protocol handling, port parsing, etc.)
- Lower Priority
  - `https://github.com/holepunchto/bare-fetch`
    - Redirect following (open redirect prevention)
    - Sensitive header handling on redirects
    - Input validation
    - Resource exhaustion

Generally, the following objectives were agreed upon:

- C code in `bare-crypto`, `bare-tls`, `bare-tcp`: Full memory safety review
- Parser security: HTTP/1.1 request smuggling
- Input validation: All JS/C boundary crossings

- DoS vectors: Resource exhaustion, algorithmic complexity attacks

2.1.2 Test Procedures

The security assessment of Tether 's JavaScript runtime was conducted using a white-box approach, providing deep insight into the modules' implementation. Tether supplied a list of modules in scope and their corresponding public repositories as well as priority for each of the modules in scope.

As an initial step, Secfault Security analyzed the modules' general architecture to gain an overview of relevant security boundaries such as the JavaScript/C boundary.

During the engagement, the assigned priorities were respected. Therefore, most of the effort was put in the high-priority modules such as `bare-crypto`, `bare-tls` and `bare-tcp`. An in-depth code review has been performed, which focused on memory-related issues that potentially arise during data crossing the JavaScript/C boundaries. Further, cryptographic primitives and implementations were inspected for common problems such as truncation or potential information leaks.

It has to be noted that the C layer was not inspected for issues that might arise from directly using the C bindings. Tether explicitly mentioned that direct use of the bindings, as for example possible in the scenario of executing untrusted code, is not supported by Bare at all.

In a next step, the less prioritized modules have been reviewed briefly. Due to the results identified in the high priority modules, a thorough analysis of the less prioritized modules could not be performed during project execution. Therefore, a best effort approach was used to review the remaining modules and therefore the respective coverage cannot be deemed comprehensive enough to evaluate their overall security.

During project execution, Tether applied mitigations to all the identified issues. All of these mitigations were reviewed by Secfault Security during project execution and all reviewed fixes were considered to mitigate the respective issues.

The identified vulnerabilities and general observations were documented, providing a detailed technical analysis, potential attack vectors, and specific recommendations for mitigation.

2.2 Project Execution

The project has been executed in the time frame from 2026-05-26 to 2026-06-09 in eight (8) person days including retest activities and documentation.

The consultants assigned to this project were:

- Gregor Kopf
- Maik Münch

3 Result Overview

An overview of the project results is provided in the following table.

Description	Chapter	Type	Exploitability	Attack Impact	Status
Using assert to Detect Error Conditions	4.1	Code	Medium	Medium	Fixed
Missing HMAC_CTX_cleanup Causes Memory Leak / Missing Zeroization	4.2	Code	Medium	Medium	Fixed
PBKDF2 key_len not Validated	4.3	Code	Medium	Medium	Fixed
PBKDF2 Iterations Silently Truncated	4.4	Code	Low	High	Fixed
Multiple Issues in Web Crypto HMAC generateKey Produce Wrong Key Material	4.5	Code	Medium	Medium	Fixed
Cipher and AEAD Construction Accept Too-Long IV/Nonce Values	4.6	Code	Low	High	Fixed
AEAD Cipher Accepts Weak Authentication Tag Length	4.7	Code	Medium	Medium	Fixed
Zeroization-Related Issues	4.8	Code/Design	Medium	High	Fixed
Missing CRLF Validation Allows Header Injection and Response Splitting	4.9	Code	Medium	Medium	Fixed
Header Storage Accessing Prototype Properties	4.10	Code	Medium	Medium	Fixed
HMAC Verification Uses Non-constant-time Comparison	4.11	Code	Medium	Medium	Fixed
Integer Overflow in Port Parsing	4.12	Code	Medium	High	Fixed
No Connection Limit on TCPServer Allows Remote Resource Exhaustion	4.13	Code	Medium	Medium	Fixed
Missing Port Range Validation Allows Silent Truncation to Unintended Port	4.14	Code	Low	Medium	Fixed
Server ALPN Negotiation Uses	4.15	Code	High	Medium	Fixed

Client Preference Order					
Inconsistent Integer Width Handling When Passing Sizes to OpenSSL APIs	4.16	Code	Medium	High	Fixed
Null Handle Dereference via alpnProtocol Getter After Destroy	4.17	Code	Medium	Medium	Fixed
ASCII Case Conversion Corrupts Non-Alphabetic Characters	4.18	Code	Medium	Low	Fixed
Null Byte Injection Allows Path Truncation at C Boundaries	4.19	Code	Medium	High	Fixed

Each identified issue is briefly described by its title, its type, its exploitability and by the impact of a successful exploitation. Technical details for the individual issues are provided in the respective sections of chapter 4 of this document. Details regarding the vulnerability rating scheme used in this document are provided in section 6.

4 Results

The issues identified during the project are described in detail in the following sections. For each finding, there is a technical description, recommended actions and - if necessary and possible - reproduction steps. For details regarding the used vulnerability rating scheme, please refer to section 6 of this document.

4.1 Using assert to Detect Error Conditions

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Medium	Medium	Fixed

Technical Description

During a brief analysis of `bindings.c` it was identified that multiple return values were checked using `assert()` as shown exemplary in the following excerpt:

```
static js_value_t *
bare_crypto_aead_init(js_env_t *env, js_callback_info_t *info) {
  ...

  int64_t tag_len;
  err = js_get_value_int64(env, argv[4], &tag_len);
  assert(err == 0);

  ...

  err = EVP_AEAD_CTX_init(&aead->context, algorithm, &key[key_offset], key_len,
tag_len, NULL);
  assert(err == 1);

  return handle;
}
```

It can be observed that `assert` is used to ensure that `EVP_AEAD_CTX_init` succeeded. However, as one can observe in the implementation of this function, there is a possibility for it to actually fail:

```
static int aead_aes_gcm_init_impl(struct aead_aes_gcm_ctx *gcm_ctx,
                                  size_t *out_tag_len, const uint8_t *key,
                                  size_t key_len, size_t tag_len) {
...
  if (tag_len == EVP_AEAD_DEFAULT_TAG_LENGTH) {
    tag_len = EVP_AEAD_AES_GCM_TAG_LEN;
```

```
  }

  if (tag_len > EVP_AEAD_AES_GCM_TAG_LEN) {
    OPENSSL_PUT_ERROR(CIPHER, CIPHER_R_TAG_TOO_LARGE);
    return 0;
  }
...
}
```

In case the `tag_len` exceeds `EVP_AEAD_AES_GCM_TAG_LEN`, the function will return `0`.

Using `assert` will not work for checking values at runtime for production builds, and is hence not recommended in places where actual checks need to be performed.

The possible consequences in this case will likely lead to DoS issues, such as `NULL` pointer dereferences.

It has to be noted that this specific case serves as an example to demonstrate potential complications arising from using `assert` instead of runtime checks. This issue should be interpreted as a reminder to be particularly careful when using this approach.

Recommended Action

It is recommended to use actual runtime checks (for instance using `if`) instead of `assert` in cases where functions can actually return error values.

Furthermore, Secfault Security would like to note that while some functions (such as for instance `RAND_bytes`) might **currently** always succeed, this is not guaranteed behavior and can be changed without further notice (the documentation of `RAND_bytes` for instance already mentions that the function might fail). It might therefore be worthwhile to introduce actual runtime checks for such functions.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

At the end of the engagement, the particular issue mentioned in the details section does still use `assert`, but `tag_len` is limited to not exceed the expected authentication tag length. Hence, this issue is considered to be fixed.

Nonetheless, it has to be noted that a thorough evaluation of all `assert` -based return value checks should be performed and applications of this pattern in the future should be verified carefully.

4.2 Missing HMAC_CTX_cleanup Causes Memory Leak / Missing Zeroization

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Medium	Medium	Fixed

Technical Description

While reviewing the HMAC functionality in the `bare-crypto` module, a missing cleanup call has been identified that causes a memory leak on every completed HMAC operation.

In `binding.c`, the HMAC lifecycle begins in `bare_crypto_hmac_init` (line 359), which allocates a `bare_crypto_hmac_t` struct via `js_create_arraybuffer` (line 401), initializes the embedded `HMAC_CTX` via `HMAC_CTX_init` (line 404), and sets up the key and algorithm via `HMAC_Init_ex` (line 406). The HMAC operation completes in `bare_crypto_hmac_final` (line 455), which calls `HMAC_Final` (line 478) to produce the digest.

However, `HMAC_Final` does not implicitly release resources owned by the `HMAC_CTX`. According to the BoringSSL documentation[1], `HMAC_CTX_cleanse` should be called separately to free data owned by the context.

Additionally, beyond the memory leak, the missing cleanup call means that key material stored internally by the `HMAC_CTX` is not zeroized when the operation completes, leaving sensitive data in memory longer than necessary.

Recommended Action

In order to address this issue, it is recommended to add a call to `HMAC_CTX_cleanse` in `bare_crypto_hmac_final` after the `HMAC_Final` call.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

`HMAC_CTX_cleanse(&hmac->context)` was added in `binding.c` at lines 524 and 651 after the `HMAC_Final` calls.

This addresses both aspects of the original finding: `HMAC_CTX_cleanse` frees data owned by the context (fixing the memory leak) and zeros the digest state beforehand.

1 https://commondatastorage.googleapis.com/chromium-boringssl-docs/hmac.h.html

4.3 PBKDF2 key_len not Validated

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Medium	Medium	Fixed

Technical Description

While reviewing the PBKDF2 key derivation functionality in the `bare-crypto` module, a missing input validation issue has been identified that allows negative or excessively large key length values to reach the native binding.

In `lib/pbkdf2.js`, the `pbkdf2` function (line 5) accepts a `keylen` parameter from the caller. The only validation performed is an `assert` on line 11 checking that the value is a number and not `NaN`. No range check could be identified. The value is passed directly to `binding.pbkdf2` on line 29.

In the C binding (`binding.c`), `bare_crypto_pbkdf2` (line 1587) reads this value as an `int64_t` via `js_get_value_int64` (line 1647). It is then passed to `js_create_arraybuffer` on line 1653, which expects a `size_t`. If `key_len` is negative, this implicit conversion to `size_t` results in a very large unsigned value.

This might be exploited as a denial of service vector, if the key length can be influenced by untrusted users.

Recommended Action

In order to address this issue, it is recommended to add a range check on `keylen` in `lib/pbkdf2.js` before calling the native binding. It should be ensured that the value is a positive integer within a reasonable upper bound. As a defense-in-depth measure, the C binding could additionally validate `key_len` before passing it to `js_create_arraybuffer` and `PKCS5_PBKDF2_HMAC`, rejecting negative values and values exceeding a sensible maximum.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

A range check was added in `lib/pbkdf2.js` that rejects `keylen` values less than `1` or greater than `0x7fffffff` with a `RangeError`.

This prevents negative and excessively large values from reaching the C binding, where the `int64_tkey_len` was previously passed unchecked to `js_create_arraybuffer` and `PKCS5_PBKDF2_HMAC`. The upper bound of `0x7fffffff` (2^31 - 1) ensures the value fits within a

signed 32-bit integer, matching the expected range in the C layer.

4.4 PBKDF2 Iterations Silently Truncated

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Low	High	Fixed

Technical Description

While reviewing the PBKDF2 key derivation functionality in the `bare-crypto` module, an integer truncation issue has been identified that can silently reduce the iteration count to a much smaller value than intended.

In `lib/pbkdf2.js`, the `pbkdf2` function (line 5) accepts an `iterations` parameter. The only range check is on line 6, which rejects values less than or equal to `0`. No upper bound check could be identified. The value is passed to `binding.pbkdf2` on line 27.

In the C binding (`binding.c`), `bare_crypto_pbkdf2` reads the iterations value as an `int64_t` via `js_get_value_int64` (line 1639). This value is then passed directly to `PKCS5_PBKDF2_HMAC` on line 1660. According to the OpenSSL documentation, `PKCS5_PBKDF2_HMAC` expects the iteration count as an `int` parameter (see https://docs.openssl.org/3.5/man3/PKCS5_PBKDF2_HMAC/; BoringSSL maintains API compatibility for this function). On platforms where `int` is 32 bits, the `int64_t` value is implicitly truncated to 32 bits at the call site. This means that values exceeding `2^31 - 1` wrap around. For example, passing `4294967297` (`2^32 + 1`) as the iteration count results in an effective value of `1`, reducing the key derivation to a single PBKDF2 iteration.

This is not expected to be directly exploitable in practice today, but `2^32` PBKDF2 iterations can be assumed to be computable in the order of minutes to hours depending on the hardware.

Recommended Action

In order to address this issue, it is recommended to add a range check on the `iterations` value in `bare_crypto_pbkdf2` before passing it to `PKCS5_PBKDF2_HMAC`.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

The iterations range check in `lib/pbkdf2.js` was changed from `iterations <= 0` to `iterations < 1 || iterations > 0xffffffff`, which rejects values exceeding `2^32 - 1`.

This prevents the silent truncation that occurred when large `int64_t` values were passed to

`PKCS5_PBKDF2_HMAC`'s `int` parameter in the C binding. Values like `4294967297` (`2^32 + 1`) that would previously wrap to `1` are now rejected at the JS layer.

4.5 Multiple Issues in Web Crypto HMAC generateKey Produce Wrong Key Material

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Medium	Medium	Fixed

Technical Description

While reviewing the Web Crypto API layer in the `bare-crypto` module, multiple related issues have been identified in the `generateKey` function for HMAC keys (`lib/web/algorithm/hmac.js`). Together, these cause the function to generate keys of incorrect length, with misleading metadata, and to throw on valid inputs.

Incorrect destructuring target

The line `const { length = exports.getKeyLength(algorithm) } = algorithm.length` destructures the value of `algorithm.length` (a Number primitive) rather than the `algorithm` object itself. Since a Number primitive has no own `length` property, the destructure always yields `undefined`, causing the default (`exports.getKeyLength(algorithm)`) to fire regardless of whether the caller provided an explicit length. Furthermore, when `algorithm.length` is `undefined` (which is valid per the WebCrypto specification, meaning "use the default"), destructuring `undefined` throws a `TypeError` before the default can run.

Bits-to-bytes confusion

The expression `exports.getKeyLength` returns the key length in bits (e.g., `512` for SHA-256), consistent with the WebCrypto specification. However, `crypto.randomBytes(length)` interprets its argument as bytes. As a result, `randomBytes(512)` generates 512 bytes (4096 bits) of random data instead of the intended 64 bytes (512 bits).

HMAC-of-empty construction instead of raw key

`crypto.createHmac(hash.name, crypto.randomBytes(length)).digest()` computes an HMAC with empty input using the random bytes as a key, then stores the resulting digest as the actual key material. The digest length is determined by the hash algorithm (e.g., 20 bytes for SHA-1, 32 bytes for SHA-256), not by the requested key length. This means the stored key is always hash-output-sized, regardless of what was requested. The `key.algorithm.length` metadata still reports the originally requested value, creating a mismatch between the reported and actual key lengths.

As a concrete example: requesting a 256-bit HMAC key with SHA-1 produces a stored key of 20

bytes (160 bits), while `key.algorithm.length` reports `256`. Code that trusts this metadata for key derivation parameters, compliance checks, or interoperability decisions operates on incorrect information.

The entropy of the stored key is not critically reduced; however, the consequences are notable:

- The function violates the WebCrypto specification, which requires `generateKey` for HMAC to produce a key of exactly the requested number of random bits.
- Keys exported via `exportKey('raw')` have the wrong size, breaking interoperability with other WebCrypto implementations and protocols that expect specific key lengths.
- The `key.algorithm.length` metadata is incorrect, which can mislead application logic.

Additionally, `getKeyLength` is missing the SHA-384 case (which should return `1024` per the spec) and does not handle `hash` being passed as an object (`{name: 'SHA-256'}`) rather than a string, although the WebCrypto API surface allows both forms.

Recommended Action

In order to address this issue, it is recommended to:

- Change the destructuring to operate on the `algorithm` object instead of `algorithm.length`
- Fix the bits vs. bytes confusion, for instance by using `crypto.randomBytes(length / 8)`
- Add the SHA-384 case to `getKeyLength` and handle `hash` being either a string or an object with a `name` property

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

The following changes were made in `lib/web/algorithm/hmac.js`:

- The incorrect destructuring `const { length = ... } = algorithm.length` was replaced with `const length = exports.getKeyLength(algorithm)`.
- The `crypto.createHmac(hash.name, crypto.randomBytes(length)).digest()` construction was replaced with `crypto.randomBytes(length / 8)`.
- In `getKeyLength`, SHA-384 was added to the default length lookup (returning `1024`), and `hash` is now unwrapped from an object to a string if passed as `{name: '...'}`.

These changes address all aspects of the original finding: the destructuring now correctly delegates to `getKeyLength` (which respects caller-supplied `length`), the bits-to-bytes conversion is added, raw random bytes are used as key material instead of an HMAC digest, SHA-384 is supported, and `hash` can be provided in either string or object form.

4.6 Cipher and AEAD Construction Accept Too-Long IV/Nonce Values

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Low	High	Fixed

Technical Description

While reviewing the cipher construction in the `bare-crypto` module, an insufficient length validation issue has been identified for initialization vectors and nonces.

In `lib/cipher.js`, the `CryptoCipher` constructor and the `CryptoAuthenticatedCipher` constructor validate the IV and nonce lengths using a less-than comparison:

```
if (iv.byteLength < binding.cipherIVLength(algorithm)) {
  throw new RangeError('Invalid iv length')
}

if (nonce.byteLength < binding.aeadNonceLength(algorithm)) {
  throw new RangeError('Invalid nonce length')
}
```

These checks reject IVs and nonces that are too short, but accept values that exceed the expected length. For unauthenticated ciphers (such as AES-CBC, AES-CTR, AES-OFB), `EVP_CipherInit` in `binding.c` receives only a pointer to the IV without an explicit length parameter. It reads exactly the number of bytes expected by the algorithm (e.g., 16 bytes for AES) and silently ignores any trailing bytes. This means two distinct IVs that share the same first 16 bytes but differ afterward (e.g., a 17-byte IV and an 18-byte IV with the same prefix) would produce identical ciphertexts. For modes like CBC and CTR, where IV/nonce uniqueness per key is essential, this silently breaks the security guarantees of the cipher.

Recommended Action

In order to address this issue, it is recommended to change the less-than comparisons to strict equality checks (`!==`) in both constructors, so that IVs and nonces that do not exactly match the expected algorithm length are rejected. This would prevent silent truncation and ensure the caller's intent matches the actual cryptographic operation.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

The < comparisons were changed to `!==` in both `CryptoCipher` and `CryptoAuthenticatedCipher` in `lib/cipher.js`.

This ensures that only IVs and nonces matching the exact expected length for the algorithm are accepted, preventing silent truncation.

4.7 AEAD Cipher Accepts Weak Authentication Tag Length

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Medium	Medium	Fixed

Technical Description

During the investigation of Authenticated Encryption with Associated Data (AEAD) handling, it was identified that `bare_crypto_aead_init()` accepts arbitrary authentication tag length.

As shown in the following excerpt of the file `binding.c`, no bounds checking is performed on the tag length:

```
static js_value_t *
bare_crypto_aead_init(js_env_t *env, js_callback_info_t *info) {
  ...

  int64_t tag_len;
  err = js_get_value_int64(env, argv[4], &tag_len);
  assert(err == 0);

  ...

  err = EVP_AEAD_CTX_init(&aead->context, algorithm, &key[key_offset], key_len,
tag_len, NULL);
  assert(err == 1);

  return handle;
}
```

The underlying initialization code in BoringSSL ensures that the authentication tag is between `1` and `16` bytes long, thus allowing a small tag length such as `4`.

A tag length of only 32 bit potentially allows an attacker to forge a GCM tag, resulting in an integrity weakness.

Recommended Action

To address this issue, it should be evaluated if the tag length should be bounds checked to only allow for sane GCM tag lengths in respect to the threat model. Otherwise, the risk of short authentication tag lengths should be documented for users of the module.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be

provided.

Retest Status

A validation check was added in `lib/cipher.js`, restricting `authTagLength` to `12`, `14`, or `16` bytes, throwing a `RangeError` for any other value. This is enforced in the JS layer before the value reaches `binding.aeadInit`.

This addresses the original problem by rejecting short tag lengths (such as 4 bytes) that would weaken GCM integrity.

4.8 Zeroization-Related Issues

Summary

Type	Location	Exploitability	Attack Impact	Status
Code/Design	bare-crypto	Medium	High	Fixed

Technical Description

While reviewing the key management functionality of `bare-crypto`, an issue has been identified in the way cryptographic key material is stored in memory and in the absence of a mechanism to securely dispose of it.

Key material is held in garbage-collected JavaScript memory throughout the library. In `lib/key.js`, line 20, `CryptoEd25519Key` stores the raw bytes as `this._key`, which are later consumed as-is (for example `lib/signature.js`, line 24 and line 58). The WebCrypto layer behaves analogously, holding the material as `this._handle` in `lib/web/crypto-key.js`, line 8, which the HMAC and PBKDF2 implementations read directly.

At the native layer, several functions in `binding.c` allocate JavaScript `ArrayBuffers` via `js_create_arraybuffer` and write sensitive key material into them before returning them:

- `bare_crypto_ed25519_generate_keypair`, line 1165: the freshly generated private key (`ED25519_keypair`, line 1171).
- `bare_crypto_ed25519_from_pkcs8`, line 1518: the private key extracted from DER (`EVP_PKEY_get_raw_private_key`, line 1522).
- `bare_crypto_ed25519_to_pkcs8`, line 1460: the DER-encoded private key. Notably, the code already treats this material as sensitive, wiping the intermediate DER buffer with `OPENSSL_cleanse` at line 1465, yet the identical copy written into the returned `ArrayBuffer` at line 1463 is not cleansed.
- `bare_crypto_pbkdf2`, line 1653: the derived key material (`PKCS5_PBKDF2_HMAC`, line 1660).

In general, two distinct problems arise: Firstly, the key bytes live in garbage-collected `ArrayBuffer` or `Buffer` instances, and no API to explicitly zero and release them could be identified on the `CryptoKey` classes. Secret bytes can therefore remain resident in process memory for an unbounded period after use, and copies may be left behind by a relocating collector. The inconsistency in `bare_crypto_ed25519_to_pkcs8` illustrates the gap directly. Additionally, in native code contexts are not deterministically cleaned up. The HMAC, cipher, and AEAD contexts are allocated inside JavaScript `ArrayBuffers` (`binding.c` lines 401, 624, 912) and initialized into that storage (lines 404, 631, 919). No finalizer invoking the corresponding cleanup routines could be identified, so allocations held internally by the OpenSSL context structures are not released when the backing

ArrayBuffer is reclaimed.

Neither problem is a directly remotely triggerable vulnerability on its own. Their relevance is realized in combination with a memory-disclosure capability: anyone with access to a memory image of the process (for example through a core dump, a swap file, or a debugging interface) could recover secret key material that lingers longer than necessary. The impact depends on the deployment context and on the sensitivity of the keys handled.

Recommended Action

To address the primary issue, it is recommended to move secret key material out of garbage-collected memory into native, manually managed memory and to provide an explicit disposal mechanism:

- Store key bytes in a native allocation owned by the binding, exposing lifecycle functions to initialize, export, and destroy the key. On destruction, the buffer should be wiped with a function that is not optimized away, such as `OPENSSL_cleanse`.
- Register a native finalizer that runs the same cleanse-and-free routine on garbage collection, as a backstop when explicit disposal is omitted.
- Add an explicit disposal API to the `CryptoKey` classes enabling deterministic wiping.
- Route the affected functions through this native key storage and zero any transient JavaScript buffers once their contents have been copied into native storage.

To address the secondary issue, it is recommended to allocate the HMAC, cipher, and AEAD contexts natively and register a finalizer that calls the corresponding OpenSSL cleanup routine.

Reproduction Steps

The issue has been identified during a static code review and hence no reproduction steps are provided.

Retest Status

Tether addressed this issue during project execution by moving cryptographic key material from the JavaScript heap to the C heap and adding deterministic and finalizer-based wiping. All key material on the C heap is zeroed on garbage collection of the owning JavaScript handle. Key material that is handed to a caller (e.g., `export()` or `to_pkcs8`) can explicitly be zeroed by the caller.

Hence, this issue is considered fixed.

4.9 Missing CRLF Validation Allows Header Injection and Response Splitting

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-http1	Medium	Medium	Fixed

Technical Description

The `_header()` methods in both `server-response.js` and `client-request.js` construct HTTP messages by string concatenation. Header values are interpolated as-is on `server-response.js` line 71 and `client-request.js` line 65:

```
h += httpCase(n) + ': ' + v + '\r\n'
```

The values originate from `setHeader` in `outgoing-message.js`, which stores them without validation:

```
setHeader(name, value) {
  this._headers[name.toLowerCase()] = value
}
```

Besides `setHeader`, this also applies to `statusMessage` (in the server) and to the `_message` and `_path` properties (client).

If untrusted user input is reflected in a header, this might enable response splitting attacks.

Recommended Action

It is recommended to generically handle unsafe characters when building header values. At least, newlines and carriage return characters should be checked for. Ideally, the code would throw an exception instead of attempting to salvage invalid values.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

A new validation module (`lib/validate.js`) was added with functions to check header names, header values, status messages, request paths, and methods for forbidden characters (CRLF, null bytes, control characters). These validations were integrated into `outgoing-message.js`, `server-response.js` and `client-request.js`.

This addresses all four injection surfaces from the original finding (header values, status message, request path, and method). Validation occurs both at the point of setting values and again in `_header()` before serialization, providing defense-in-depth.

4.10 Header Storage Accessing Prototype Properties

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-http-parser	Medium	Medium	Fixed

Technical Description

In `index.js`, the `_storeHeader` method stores parsed headers into `this._headers`. The method explicitly rejects the header names `__proto__`, `constructor`, and `prototype` to prevent prototype pollution. However, in the default branch, the code uses the `in` operator to check for duplicate headers:

```
if (name in this._headers) {
  this._headers[name] += delimiter + value
} else {
  this._headers[name] = value
}
```

The `in` operator traverses the prototype chain, meaning it returns `true` for any property name that exists on `Object.prototype`, even if it has not been set as an own property on `this._headers`. Properties such as `toString` and others inherited from `Object.prototype` would trigger the duplicate-detection branch on their first occurrence.

When this happens, the code executes `this._headers[name] += delimiter + value`, which reads the inherited property (possibly a function), coerces it to a string via the `+=` operator, and concatenates the delimiter and header value.

This might lead to various types of problems, depending on the particular usage of the headers in the application code.

Recommended Action

It is recommended to either use another data structure, such as a `Map`, for storing header values. If this is not an option, the code should make use of `hasOwnProperty` instead of the `in` operator.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

The `this._headers` initialization was changed from `{}` to `Object.create(null)`.

`Object.create(null)` creates an object with no prototype, so the `in` operator can no longer match inherited properties like `toString` or `hasOwnProperty`. This fixes the original issue.

4.11 HMAC Verification Uses Non-constant-time Comparison

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-crypto	Medium	Medium	Fixed

Technical Description

While inspecting the HMAC implementation in `lib/web/algorithm/hmac.js` it was identified that the verification of the signature does use a non-constant-time comparison.

The following excerpt shows the respective implementation:

```
// https://w3c.github.io/webcrypto/#hmac-operations-verify
exports.verify = function verify(algorithm, key, signature, data) {
  const digest = crypto.createHmac(key.algorithm.hash.name,
key._handle).update(data).digest()

  if (ArrayBuffer.isView(signature)) {
    signature = Buffer.coerce(signature)
  } else {
    signature = Buffer.from(signature)
  }

  return signature.equals(digest)
}
```

As stated in the referenced specification, the comparison must be performed in constant-time. However, the implementation uses `Buffer.equals()` that is a `memcmp()` based comparison which is non-constant time.

This potentially allows an attacker to use this timing side-channel as a timing oracle to infer HMAC values. However, it has to be noted that this requires a scenario where high-resolution timing measurements are possible.

Recommended Action

In order to address this issue a constant-time comparison should be used to compare the provided signature with the inferred digest.

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

`signature.equals(digest)` was replaced with `crypto.timingSafeEqual(signature, digest)`, backed by a new C binding function that uses BoringSSL's `CRYPTO_memcmp()`.

This eliminates the timing side-channel, as `CRYPTO_memcmp()` compares all bytes regardless of differences, unlike the original `memcmp` -based `Buffer.equals()`.

4.12 Integer Overflow in Port Parsing

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-url	Medium	High	Fixed

Technical Description

While reviewing the `liburl` library, the following issue when parsing port numbers was identified:

In `parse.h`, the port state handler accumulates the port value from individual ASCII digits into a `uint32_t` variable, using the usual string to integer parsing. There is a check in place, guarding that `port <= UINT16_MAX`, which is sensible. However, the intermediate `uint32_t` value might wrap around during parsing, which would make the system parse a "port number" like `12345678912345678912345678912345678 9` - a string that should likely rather be rejected.

Recommended Action

It is recommended to directly reject "port numbers" that do not fall within the range of an unsigned 16-bit integer, instead of continuing parsing (and overflowing the intermediate value).

Reproduction Steps

This issue has been identified by static code analysis, and hence no reproduction steps can be provided.

Retest Status

The `if (port > UINT16_MAX) goto err;` check was moved from after the accumulation loop to inside it, so it runs on every iteration.

This prevents the overflow because `port` is checked against `UINT16_MAX` after each digit is accumulated. Once the intermediate value exceeds `65535`, parsing fails immediately, before the `uint32_t` can wrap around to a small value.

4.13 No Connection Limit on TCPServer Allows Remote Resource Exhaustion

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-tcp	Medium	Medium	Fixed

Technical Description

While reviewing the server-side connection handling in the `bare-tcp` module, a missing connection limit has been identified that allows remote attackers to exhaust server resources by opening a large number of idle connections.

In `index.js`, the `TCPServer` class tracks active connections in `this._connections`, a `Set`. When a new connection arrives, the `_onconnection` callback is invoked. The only check before accepting the connection is whether the server is in the `CLOSING`. No check on `this._connections.size` could be identified. Every incoming connection results in:

- A new `TCPSocket` being constructed, which allocates a `bare_tcp_t` struct in the C binding via `js_create_arraybuffer` and a read buffer of `readBufferSize` bytes (default 65536) via `Buffer.alloc`
- A call to `binding.accept`, which accepts the connection at the libuv level, consuming a file descriptor
- The socket being added to `this._connections`

Each accepted connection therefore consumes at minimum one file descriptor, approximately 65 KB for the read buffer, the `bare_tcp_t` struct, and JS object overhead for the `TCPSocket` instance and its bound callbacks. Since accepted sockets have no default idle timeout (the `setTimeout` method exists but is opt-in), connections persist until the remote end disconnects or the application explicitly destroys them.

A remote attacker might be able to exploit this by opening a large number of TCP connections to the server and keeping them idle. No payload data needs to be sent.

For reference, Node.js provides `server.maxConnections` on its `net.Server` class, which rejects incoming connections when the limit is reached (see https://nodejs.org/api/net.html#servermaxconnections). No equivalent mechanism could be identified in `bare-tcp`.

Recommended Action

In order to address this issue, it is recommended to introduce a configurable connection limit on

`TCPServer`, similar to Node.js's `maxConnections` property. When the limit is reached, incoming connections should be rejected by immediately closing them.

As a complementary measure, it is recommended to consider providing a default idle timeout for server-accepted sockets, or at minimum documenting that applications using `bare-tcp` are responsible for setting timeouts on accepted connections.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

A `maxConnections` option was added to `TCPServer` (default `Infinity`), with a getter/setter for runtime modification. In `_onconnection`, when the connection count exceeds the limit, a `'drop'` event is emitted with connection metadata and the socket is destroyed. Excess connections are accepted with a zero-byte read buffer to obtain address information for the drop event, then immediately closed.

This addresses the original finding by providing a mechanism to cap the number of active connections, preventing unbounded resource consumption from idle connection flooding.

4.14 Missing Port Range Validation Allows Silent Truncation to Unintended Port

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-tcp	Low	Medium	Fixed

Technical Description

While reviewing the connection and bind logic in the `bare-tcp` module, a missing input validation issue has been identified that causes out-of-range port numbers to be silently truncated.

In `index.js`, the `connect` method (line 99) and the `listen` method (line 469) accept a user-provided `port` parameter and pass it directly to the native binding without any range validation. In the C binding, `bare_tcp_connect` (`binding.c`, line 470) and `bare_tcp_bind` (`binding.c`, line 552) read this value as a `uint32_t` via `js_get_value_uint32`. The value is then forwarded to `uv_ip4_addr` or `uv_ip6_addr`, which internally store it via `htons()` into the `sin_port` field of the socket address structure. Since `sin_port` is a `uint16_t`, the value is silently truncated to 16 bits. For example, port 65537 becomes port 1, and port 131072 becomes port 0. Additionally, negative values from JavaScript are first converted to large unsigned integers by `js_get_value_uint32` (e.g., -1 becomes 4294967295), which then also truncate silently.

While this is not directly exploitable by remote attackers, it might open an interesting opportunity for obfuscating used port numbers, or for bypassing certain restrictions in higher-level application code. For instance, if an application were to naively check that `port >= 1024`, such a check could potentially be bypassed by leveraging the above behavior.

Recommended Action

In order to address this issue, it is recommended to introduce a port range validation in the JavaScript layer for both the `connect` method in `TCPSocket` and the `listen` method in `TCPServer`. It should be ensured that the provided port value is an integer in the range 0-65535 (inclusive).

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

A `validatePort` function was added in `index.js` that checks the port is a number, an integer, and within the range `0` to `0xffff`. It throws `INVALID_PORT` for invalid values. This validation is called in both `Socket.connect` and `Server.listen` before the port reaches the C binding.

This prevents out-of-range values from being silently truncated via `htons()` in the C layer.

4.15 Server ALPN Negotiation Uses Client Preference Order

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-tls	High	Medium	Fixed

Technical Description

While reviewing the TLS handshake functionality, an ALPN (Application-Layer Protocol Negotiation) preference order issue has been identified in the `bare-tls` module.

In `binding.c`, line 172, the function `bare_tls__on_alpn_select` is registered as the server-side ALPN selection callback via `SSL_CTX_set_alpn_select_cb`. This callback is invoked during the TLS handshake when the client advertises supported application protocols. It receives the client's protocol list in the `in` / `inlen` parameters and is expected to select one of them based on the server's own preferences stored in `socket->alpn` / `socket->alpn_len`.

At line 179, the callback delegates the selection to `SSL_select_next_proto()`. This function accepts two protocol lists: a server list (preferred) and a client list (fallback). It selects the first protocol from the server list that also appears in the client list. However, the arguments are passed in the wrong order: the client-provided `in` / `inlen` is passed as the preferred server argument, while the server's own `socket->alpn` / `socket->alpn_len` is passed as the fallback client argument.

As a result, the first protocol offered by the client that appears anywhere in the server's list will be selected, regardless of the server's intended priority. This effectively gives remote clients control over application protocol selection.

Recommended Action

In order to address this issue, please swap the protocol list arguments in the `SSL_select_next_proto()` call in `bare_tls__on_alpn_select` so that `socket->alpn` / `socket->alpn_len` (the server's list) is passed as the preferred server parameter, and `in` / `inlen` (the client's list) is passed as the client parameter.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

The arguments to `SSL_select_next_proto()` in `binding.c` were swapped: `socket->alpn` /

`socket->alpn_len` (the server's list) is now passed as the preferred parameter, and `in` / `inlen` (the client's list) as the fallback.

This argument swap addresses the original finding, ensuring the server's ALPN preference order is respected during protocol selection.

4.16 Inconsistent Integer Width Handling When Passing Sizes to OpenSSL APIs

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-tls	Medium	High	Fixed

Technical Description

Several places in binding.c pass `size_t` values to BoringSSL functions that expect `int` or `unsigned int`, without checking for truncation. For example, `BIO_write()` and `BIO_new_mem_buf()` receive `(int)` casts of `size_t` lengths, `SSL_set_alpn_protos()` receives a `size_t` as `unsigned int`, and the hostname length undergoes a `size_t + 1` addition that could theoretically wrap to zero.

In a number of places, for instance when calling `SSL_read()` and `SSL_write()`, the code correctly caps to `INT_MAX` before casting, but the pattern is not applied consistently. In particular, `BIO_new_mem_buf()` interprets a length of `-1` as "use strlen", which would silently change the semantics of the call rather than failing.

The practical exploitability of these issues is likely not high, as modifying the required length fields might require unrealistically large data structures such as certificates or host names. Regardless, at least as a defense-in-depth measure and to prevent accidental misuse of the affected code in other contexts, the issues should still be addressed.

Recommended Action

It is recommended to apply the same `INT_MAX` capping pattern already used in `bare_tls_read` and `bare_tls_write` to such conversions. For `unsigned int` parameters, `UINT_MAX` should be used accordingly. Furthermore, it is recommended to add an overflow check before the hostname's `len += 1`.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

The following changes were made:

- `if (len > INT_MAX) len = INT_MAX;` guards were added before the `BIO_write` calls (lines 366, 425) and the `BIO_new_mem_buf` call (line 517).

- An `if (len == SIZE_MAX)` check was added for the hostname `len += 1` operation (line 465), preventing the wrap-around to zero.
- `if (len > UINT_MAX) len = UINT_MAX;` was added before the `SSL_set_alpn_protos` call (line 578), with an explicit `(unsigned int)` cast.

These changes prevent the integer width truncation at all identified locations. The hostname wrap-around is properly handled by erroring out, and the remaining locations are clamped before the casts.

4.17 Null Handle Dereference via alpnProtocol Getter After Destroy

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-tls	Medium	Medium	Fixed

Technical Description

While reviewing the TLS socket lifecycle in the `bare-tls` module, a null handle dereference issue has been identified in the `alpnProtocol` property getter.

In `index.js`, line 100, the `alpnProtocol` getter calls `binding.alpnProtocol(this._handle)` without checking whether `this._handle` is `null`. When a `TLSSocket` is destroyed, `this._handle` is set to `null` after the native resources are freed.

When `binding.alpnProtocol(null)` is called, the C function `bare_tls_alpn_protocol` invokes `js_get_arraybuffer_info` on the `null` value. This call is expected to fail, but its return value is only checked via assert.

Please note that this issue requires the caller to access the `alpnProtocol` property after the socket has been destroyed; whether such a situation will happen is not trivial to judge. It might for example be the case that in cleanup or debugging code the `aplnProtocol` property is accessed even after the socket is closed.

Recommended Action

In order to address this issue, it is recommended to add a `null` check on `this._handle` in the `alpnProtocol` getter in `index.js`, returning `null` when the handle has already been destroyed. Additionally, as a defense-in-depth measure, it might be worth adding an explicit `null` check on the socket pointer in `bare_tls_alpn_protocol` in `binding.c` after the `js_get_arraybuffer_info` call, rather than relying solely on assert.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

A null guard was added to the `alpnProtocol` getter in `index.js`: `return this._handle ? binding.alpnProtocol(this._handle) : null`.

This prevents the null handle from reaching the C binding after destroy, returning `null` instead.

4.18 ASCII Case Conversion Corrupts Non-Alphabetic Characters

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-url	Medium	Low	Fixed

Technical Description

The auditors briefly inspected the `liburl` library, in order to better understand the implementation. While reviewing the code in `infra.h`, the function `url__to_ascii_lowercase` was inspected:

```
static inline utf8_t
url__to_ascii_lowercase (utf8_t c) {
  return c | 0x20;
}
```

Similarly, `url__to_ascii_uppercase` is implemented as `c & ~0x20`. These bit-manipulation shortcuts rely on the fact that ASCII uppercase and lowercase letters differ only in bit 5 (`0x20`). However, the functions do not check whether the input is actually an alphabetic character. When applied to non-alphabetic bytes, the OR/AND operation produces incorrect results.

Recommended Action

In order to provide defense-in-depth, it is recommended to first verify that the characters that are being processed actually fall within the ASCII range. Otherwise, the function should not perform any modifications.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

`url__to_ascii_lowercase` and `url__to_ascii_uppercase` in `infra.h` now check whether the input is an alphabetic character before applying the bitwise operation. Non-alphabetic characters are returned unchanged.

4.19 Null Byte Injection Allows Path Truncation at C Boundaries

Summary

Type	Location	Exploitability	Attack Impact	Status
Code	bare-url	Medium	High	Fixed

Technical Description

The `fileURLToPath` function converts a `file:` URI into a local file system path. After some initial checks, it performs the following operation:

```
const pathname = path.normalize(decodeURIComponent(url.pathname))
```

No check for `%00` (encoded null bytes) could be identified in this function.

If the resulting path is subsequently passed to a native file system operation implemented in C, the null byte acts as a string terminator. For example, a path like `/etc/passwd\x00.png` would be seen as `/etc/passwd` by any C API that operates on null-terminated strings. An application that relies on checking the file extension in the JavaScript string (e.g., verifying that only `.png` files are served) could be bypassed this way.

Recommended Action

In order to address this issue, it is recommended to reject inputs that contain embedded `null` bytes.

Reproduction Steps

This issue has been identified during a static code review and hence no reproduction steps can be provided.

Retest Status

A `/%00/i` regex check was added in `fileURLToPath`, which now throws `INVALID_FILE_URL_PATH` when `%00` is present in the path name. This check runs before `decodeURIComponent`.

This prevents encoded null bytes from reaching the decode step and appearing as literal null bytes in the output path.

5 Additional Observations

Secfault Security would like to point out a number of general observations and recommendations regarding the analyzed system in the following subsections.

5.1 Uninitialized Variables

While reviewing the native C bindings across the codebase, a recurring pattern has been identified where variables are declared without initialization and set via a function call that might fail.

The consequences are most severe when the uninitialized variable is a pointer, because subsequent writes through it might target an arbitrary stack-derived address rather than a predictable location. The following two examples illustrate this:

- In `bare-crypto/binding.c`, `bare_crypto_pbkdf2` (line 1652): `uint8_t *out;` is declared without initialization and passed to `js_create_arraybuffer`. If the allocation fails and the `assert` is stripped, `out` remains uninitialized. It is then passed to `PKCS5_PBKDF2_HMAC` on line 1660, which writes `key_len` bytes to whatever address `out` happens to contain.
- In `bare-tls/binding.c`, `bare_tls_init` (line 305): `bare_tls_t *socket;` is declared without initialization and allocated via `js_create_arraybuffer`. It is then dereferenced on line 308 to set `socket->certificate = NULL` and used throughout the rest of the function to configure the TLS connection.

For non-pointer types the consequences are different but still problematic. For instance, `int err;` is declared without initialization throughout the bindings and used to hold return values from assert-checked calls. If an `assert` is stripped and the call that was supposed to set `err` fails, subsequent control flow decisions that depend on `err` operate on a stale stack value, potentially taking the wrong branch and skipping error handling or executing unintended code paths.

Please note that this issue is a defense-in-depth concern. It is likely not directly exploitable. However, as the reviewed codebase is intended to be used as a fundamental set of libraries, applying stricter hardening measures might be advisable.

Initializing variables, in particular pointer variables, is therefore recommended.

6 Vulnerability Rating

This section provides a description of the vulnerability rating scheme used in this document. Each finding is rated by its type and its exploitability/impact of a successful exploitation. The meaning of the individual ratings are provided in the following sub-sections.

6.1 Vulnerability Types

Vulnerabilities are rated by the types described in the following table.

Type	Description
Configuration	The finding is a configuration issue
Design	The finding is the result of a design decision
Code	The finding is caused by a coding mistake
Observation	The finding is an observation, which does not necessarily have a direct impact

6.2 Exploitability and Impact

The exploitability of a vulnerability describes the required skill level of an attacker as well as the required resources. Therefore, it provides an indication of the likelihood of exploitation.

Exploitability Rating	Description
Not Exploitable	This finding can most likely not be exploited.
Minimal	Although an attack is theoretically possible, it is extremely unlikely that an attacker will exploit the identified vulnerability.
Low	Exploiting the vulnerability requires the skill-level of an expert. An attack is possible, but difficult pre-conditions (e.g., prior identification and exploitation of other vulnerabilities) exist or the attack requires resources not available to the general public (e.g., expensive equipment). Successful exploitation indicates a dedicated, targeted attack.
Medium	The vulnerability can be exploited under certain pre-conditions (e.g., user interaction or prior authentication). Non-targeted, random attacks are possible for attackers with a medium skill level who perform such attacks on a regular basis.
High	The vulnerability can be exploited immediately without special pre-conditions, by random attackers or in an automated fashion. Only general knowledge about vulnerability exploitation is required.

The following table describes the impact rating used in this document.

Impact Rating	Description
Critical	The vulnerability is a systematic error or it permits compromising the system completely and beyond the scope of the assessment.

Impact Rating	Description
High	The vulnerability permits compromising the systems within the scope completely.
Medium	The vulnerability exceeds certain security rules, but does not lead to a full compromise (e.g., Denial of Service attacks)
Low	The vulnerability has no direct security consequences but provides information which can be used for subsequent attacks.
Informational	The observed finding does not have any direct security consequence; however, addressing the finding can lead to an increase in security or quality of the system in scope.

When rating the impact of a vulnerability, the rating is always performed based on the scope of the analysis. For example, a vulnerability with high impact typically allows an attacker to fully compromise one or all of the core security guarantees of the components in scope. Identical vulnerabilities can therefore be rated differently in different projects.

7 Glossary

Term	Definition
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CBC	Cipher Block Chaining
CTR	Counter Mode (Used to operate a block cipher)
DER	Distinguished Encoding Rules
DoS	Denial Of Service
GCM	Galois Counter Mode
HMAC	Keyed-Hash Message Authentication Code
HTTP	Hyper Text Transfer Protocol
IV	Initialization Vector
SHA	Secure Hash Algorithm
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator