Code Audit Report

Blockchain Commons LLC, with financial support by Bitmark Inc.

V 1.1
Amsterdam, September 14th, 2021
Public

Document Properties

Client	Blockchain Commons LLC, with financial support by Bitmark Inc.
Title	Code Audit Report
Targets	bc-shamir Shamir Secret Sharing (SSS) library bc-sskr Shared Secret Key Reconstruction library
Version	1.1
Pentesters	Stefan Marsiske, Christian Reitter, Jonathan Levin
Authors	Christian Reitter, Stefan Marsiske, Jonathan Levin, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	August 26th, 2021	Christian Reitter, Stefan Marsiske, Jonathan Levin	Initial draft
1.0	August 29th, 2021	Marcus Bointon	Review
1.1	September 14th, 2021	Christian Reitter, Jonathan Levin	Change title, BTM-019, conclusion

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between August 2, 2021 and August 27, 2021, Radically Open Security B.V. carried out a code audit for Blockchain Commons LLC, with financial support by Bitmark Inc.

This report contains our findings as well as detailed explanations of exactly how ROS performed the code audit.

1.2 Scope of work

The scope of the code audit was limited to the following targets:

- bc-shamir Shamir Secret Sharing (SSS) library
- bc-sskr Shared Secret Key Reconstruction library

The target repository sources and specific commit hashes for the initial review:

- Github bc-shamir - `8e5a555665c91af6fb11bc54b26c2ffd9506b63e`
- Github bc-sskr - `e5131c02b102e2d4f48c278f5b025680c63d1672`

Note that there have been several patches to the target library during the audit timeframe which were also reviewed upon request.

The scoped services are broken down as follows:

- Timeboxed audit of bc-shamir and bc-sskr libraries including reporting, communication and all other relevant steps: 5 days
- Additional in-depth audit: 0-3 days
- **Total effort: 5 - 8 days**

1.3 Project objectives

ROS, in conjunction with Bitmark, will perform a code audit of the target libraries in order to assess their security under practical conditions. To do so ROS will analyze the repositories and guide Bitmark in attempting to find vulnerabilities, exploiting any such found to evaluate their impact, and recommend suitable solutions or mitigations.

1.4 Timeline

The Security Audit took place between August 2, 2021 and August 27, 2021.

1.5 Results In A Nutshell

During this crystal-box code audit we found 2 Elevated, 5 Moderate, 2 Low and 1 High-severity issues.

Six issues relate to observed or potential memory corruption through boundary violations. Two findings concern the risk of information disclosure through discarded or swapped memory, and one finding is about missing mitigation features that can limit the impact of memory corruption. Additionally, there is one cryptanalytic finding about weakened security guarantees in this implementation of SSS compared to the classical scheme.

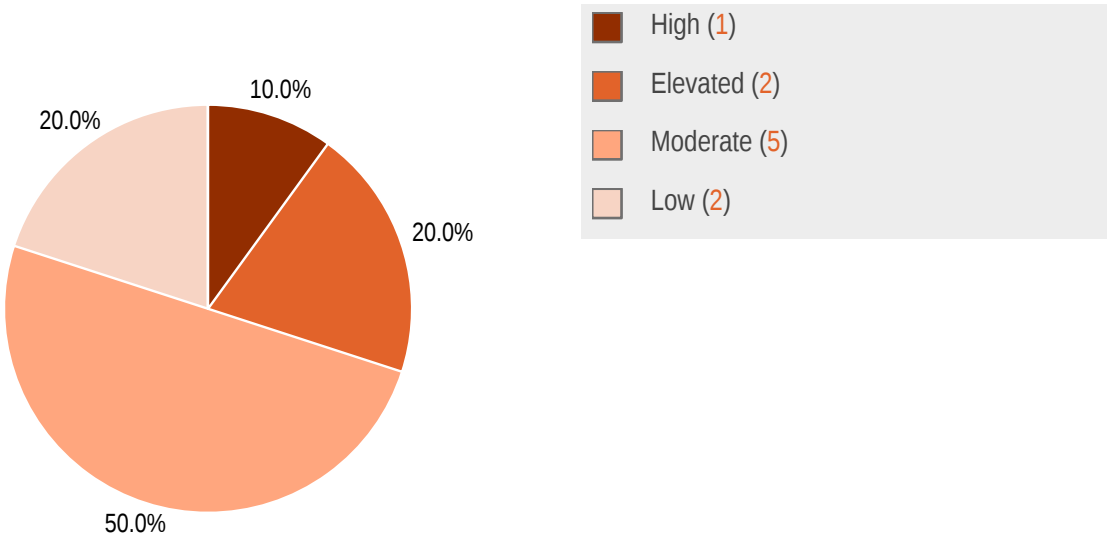
By exploiting these issues, an attacker might be able to crash the program, read memory, trigger unexpected runtime behavior or execute arbitrary code. Through information disclosure, it may also be possible for an attacker to obtain cryptographic secret keys that are handled by the libraries.

1.6 Summary of Findings

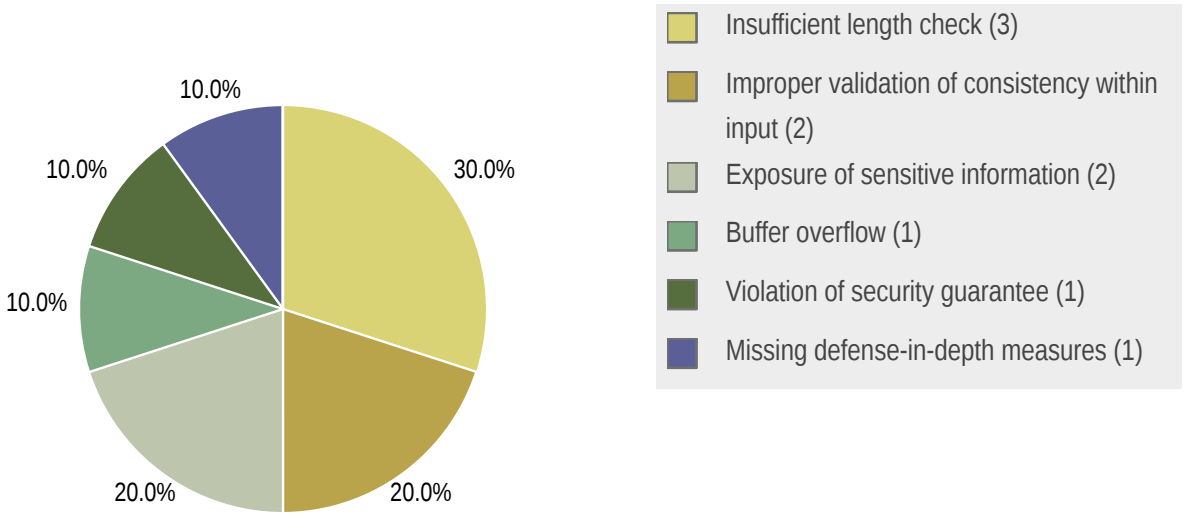
ID	Type	Description	Threat level
BTM-018	Insufficient Length Check	Insufficient length checks in sskr_generate() and generate_shards() allow dangerous out-of-bounds writes.	High
BTM-022	Improper Validation of Consistency within Input	Calls to the recover_secret() function with large, small, or inconsistent values for the threshold or shard_count parameters can lead to security issues.	Elevated
BTM-021	Buffer Overflow	There is a possible buffer overflow in hazmat_lagrange_basis().	Elevated
BTM-020	Improper Validation of Consistency within Input	Function calls to split_secret() with inconsistent threshold or shard_count parameters can lead to security issues.	Moderate
BTM-010	Missing Defense-In-Depth Measures	We were unable to find common hardening flags in the bc-shamir and bc-sskr library build system definitions.	Moderate
BTM-007	Exposure of Sensitive Information	Sensitive values in bc-shamir can be forced to leak to disk by a local attacker inducing the process to be swapped out.	Moderate
BTM-006	Exposure of Sensitive Information	The bc-shamir and bc-sskr libraries use memset() to sanitize sensitive data, but that may be optimized away by the compiler, depending on build settings.	Moderate
BTM-005	Insufficient Length Check	Insufficient length checks in split_secret() allow out of bounds writes and segmentation fault crashes.	Moderate
BTM-019	Violation of Security Guarantee	The inclusion of a hash digest of the secret in the bc-shamir design removes the information theoretical	Low

		security property of the original Shamir secret sharing scheme (SSS). This represents a cryptographic design tradeoff that is also present in related designs, but should be explicitly documented and evaluated by the bc-shamir authors.	
BTM-015	Insufficient Length Check	The static size_t serialize_shard() function has a size_t destination_len parameter that is not used for memory access control of the provided uint8_t *destination buffer.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
BTM-022	Improper Validation of Consistency within Input	• Include additional defensive checks on `threshold` and `share_length` to reject invalid parameter values.
BTM-021	Buffer Overflow	• Do not expose any of the functions in `interpolate.c` to the outside world. • Enforce limits on function parameter `n`.
BTM-020	Improper Validation of Consistency within Input	• Include additional defensive value checks on `threshold` and `shard_count` to reject invalid parameter values.
BTM-019	Violation of Security Guarantee	• We recommend documenting this design tradeoff in public documents on `bc-shamir` so that users are aware of the implications. • We have listed a number of design changes that can be included in future revisions of this scheme to remove this issue.
BTM-018	Insufficient Length Check	• Add an unified safety check function that validates `master_secret_len` properties. • Add new error types in `sskr-errors.h` for `ERROR_SECRET_TOO_LONG` and `ERROR_SECRET_LENGTH_UNEVEN`. • Alternatively, redefine the existing `ERROR_SECRET_TOO_LONG` as `ERROR_SECRET_INVALID_LENGTH` to cover all three cases. • Investigate problematic use of the affected function in other projects.
BTM-015	Insufficient Length Check	• Include defensive length checks on `destination_len` before performing any memory accesses.

		• Evaluate how to flag error conditions to the caller.
BTM-010	Missing Defense-In-Depth Measures	• Add compiler flags to the default build settings that increase the defense-in-depth of the resulting library artifacts. • Apply the hardening flags to testing code as well, as this may help detect security issues at an earlier stage.
BTM-007	Exposure of Sensitive Information	• Use `mlock(2)` or `VirtualLock()` on Windows, or equivalent functions on other OS if available.
BTM-006	Exposure of Sensitive Information	• Use `memzero()` from `bc-crypto-base` instead of `memset(3)`.
BTM-005	Insufficient Length Check	• Add an error handling flag to `split_secret()`. • Reject invalid parameters to the `split_secret()` function. • Investigate the impact of functionally incorrect results.

2 Methodology

2.1 Planning

During the code audit we verify if the proper security controls are present, work as intended and are implemented correctly. If vulnerabilities are found, we determine the threat level by assessing the likelihood of exploitation of this vulnerability and the impact on the Confidentiality, Integrity and Availability (CIA) of the system. We will describe how an attacker would exploit the vulnerability and suggest ways of fixing it.

This requires an extensive knowledge of the platform the application is running on, as well as the extensive knowledge of the language the application is written in and patterns that have been used. Therefore a code audit is done by highly-trained specialists with a strong background in programming.

During this code audit, we take the following approach:

1. **Thorough comprehension of functionality**
   We try to get a thorough comprehension of how the application works and how it interacts with the user and other systems. Having detailed documentation at this stage is very helpful, as it aids the understanding of the application.
2. **Comprehensive code reading**
   Goals of the comprehensive code reading are:
   - to get an understanding of the whole code
   - identify adversary controlled inputs and trace their paths
   - identify issues
3. **Fuzzing**
   Fuzz testing or Fuzzing is a software testing technique which in essence consists of finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.
   As arranged with the client during the scoping, this is an explicit goal of the code audit and will be one of the main focus areas besides the manual code audit.
4. **Dynamic analysis**
   For dynamic analysis the program is run and actively analyzed or exploited by the specialist. During this code audit, dynamic analysis is used to confirm and evaluate the vulnerabilities that are found during manual analysis and fuzz testing.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.
- **High**
  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**
  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**
  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**
  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Automated Testing

A portion of this code audit was based on semi-automated and automated testing tools and frameworks. This section will give a brief overview of the technologies used. Any relevant scan output will be referred to in the findings.

- libFuzzer – coverage-guided fuzz testing framework
- AddressSanitizer – fast memory error detector
- UndefinedBehaviorSanitizer – fast undefined behavior detector
- MemorySanitizer – detector of uninitialized reads

In addition to the software listed above, we used verbose build warning configurations of the `clang-11` compiler in order to list potentially problematic code positions.

4 Findings

We have identified the following issues:

4.1 BTM-022 — Missing Parameter Checks at recover_secret() in bc-shamir

Vulnerability ID: BTM-022	**Status:** Resolved
Vulnerability type: Improper Validation of Consistency within Input	
Threat level: Elevated	

Description:

Calls to the `recover_secret()` function with large, small, or inconsistent values for the `threshold` or `shard_count` parameters can lead to security issues.

Technical description:

The function definition is as follows:

```
// returns the number of bytes written to the secret array, or -1 if there was an error
int32_t recover_secret(
    uint8_t threshold,
    const uint8_t *x,
    const uint8_t **shares,
    uint32_t share_length,
    uint8_t *secret
) {
```

The code does not enforce any limits on `uint8_t threshold` and `uint32_t share_length` or the logical relationships between them, such as `threshold <= share_length`.

For example, a call with `threshold == 0` causes runtime errors:

```
interpolate.c:144:16: runtime error: variable length array bound evaluates to non-positive value 0
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior interpolate.c:144:16 in
```

Some calls to `recover_secret()` may lead to out-of-bounds accesses. Here is a modified test case in `test.c` with a threshold of `0` instead of `3`:

```
_test_shamir("0ff784df000c4380a5ed683f7e6e3dcf", 0, 5, (uint8_t[]){1, 2, 4});
```

With this test case, the following out-of-bounds read was observed:

```
shamir.c:140:25: runtime error: index 4 out of bounds for type 'uint8_t [share_length]'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior shamir.c:140:25 in
```

```
=================================================================
==12843==ERROR: AddressSanitizer: dynamic-stack-buffer-overflow on address 0x7ffcbfd76ea4 at pc
 0x0000004977b7 bp 0x7ffcbfd76aa0 sp 0x7ffcbfd76268
READ of size 64 at 0x7ffcbfd76ea4 thread T0
    $0 0x4977b6 in __asan_memcpy (/home/user/target/bc-shamir/test/test+0x4977b6)
    $1 0x4fb14b in sha256_Update (/home/user/target/bc-shamir/test/test+0x4fb14b)
    $2 0x4fc461 in sha256_Raw (/home/user/target/bc-shamir/test/test+0x4fc461)
    $3 0x4f86b6 in hmac_sha256_Init (/home/user/target/bc-shamir/test/test+0x4f86b6)
    $4 0x4f8adc in hmac_sha256 (/home/user/target/bc-shamir/test/test+0x4f8adc)
    $5 0x4ca95e in create_digest /home/user/target/bc-shamir/src/shamir.c:31:5
    $6 0x4cdbee in recover_secret /home/user/target/bc-shamir/src/shamir.c:140:5
    $7 0x4ca31d in _test_recover_secret /home/user/target/bc-shamir/test/test.c:42:20
    $8 0x4c8f10 in _test_shamir /home/user/target/bc-shamir/test/test.c:67:22
    $9 0x4c8441 in test_shamir /home/user/target/bc-shamir/test/test.c:97:3
```

BTM-020 (page 15) is a similar problem in the related `split_secret()` function.

Impact:

The `recover_secret()` function does not reject incorrect calls, which may result in memory corruption or inconsistent results.

Recommendation:

- Include additional defensive checks on `threshold` and `share_length` to reject invalid parameter values.

4.2 BTM-021 — Buffer-overflow in hazmat_lagrange_basis

Vulnerability ID: BTM-021	**Status:** Resolved
Vulnerability type: Buffer Overflow	
Threat level: Elevated	

Description:

There is a possible buffer overflow in `hazmat_lagrange_basis()`.

Technical description:

If an attacker passes a `n` > 24 to `hazmat_lagrange_basis()` then in line 54 of `interpolate.c` you have a buffer overflow. An attacker can also control the input to this buffer.

`hazmat_lagrange_basis()` is unnecessarily public (not static and exposed through the file `interpolate.h`) since it's not called by anything but `interpolate()` in the same file. The function being public means that anyone else using this library and calling this function might use it incorrectly. Furthermore `interpolate()` itself (also a public function, but that is necessary for `shamir.c`) has no checks on its `n` parameter which is passed on directly to `hazmat_lagrange_basis()`.

Both `interpolate()` and `hazmat_lagrange_basis()` are public functions; they are even exposed through `bc-shamir.h`.

This issue affects git revision `8e5a555665c91af6fb11bc54b26c2ffd9506b63e`.

Also notable is that the source code documentation for `hazmat_lagrange_basis` says

```
*         n: number of points - length of the xc array, 0 < n <= 32
```

While this code breaks with `n` <= 32.

Impact:

An adversary can run code, possibly leaking the secret being split/recovered.

Recommendation:

- Do not expose any of the functions in `interpolate.c` to the outside world.
- Enforce limits on function parameter `n`.

4.3 BTM-020 — Missing Parameter Checks at split_secret() in bc-shamir

Vulnerability ID: BTM-020	**Status:** Resolved
Vulnerability type: Improper Validation of Consistency within Input	
Threat level: Moderate	

Description:

Function calls to `split_secret()` with inconsistent `threshold` or `shard_count` parameters can lead to security issues.

Technical description:

The function definition is as follows:

```
// shamir sharing
int32_t split_secret(
    uint8_t threshold,
    uint8_t shard_count,
    const uint8_t *secret,
    uint32_t secret_length,
    uint8_t *result,
    void* ctx,
    void (*random_generator)(uint8_t *, size_t, void*)
) {
```

The code enforces the limitation `shard_count <= SHAMIR_MAX_SHARD_COUNT`, but has no checks on the `uint8_t threshold` variable, which needs to be within certain bounds for the function to operate correctly.

BTM-021 (page 14) and BTM-022 (page 13) are related issues that contain additional technical information.

Impact:

The `split_secret()` function does not reject incorrect calls, which may result in memory corruption or inconsistent results.

Recommendation:

- Include additional defensive value checks on `threshold` and `shard_count` to reject invalid parameter values.

4.4 BTM-019 — T-out-of-N Property Violated with Partial Knowledge of S

Vulnerability ID: BTM-019

Vulnerability type: Violation of Security Guarantee

Threat level: Low

Description:

The inclusion of a hash digest of the secret in the `bc-shamir` design removes the information theoretical security property of the original Shamir secret sharing scheme (SSS). This represents a cryptographic design tradeoff that is also present in related designs, but should be explicitly documented and evaluated by the `bc-shamir` authors.

Technical description:

Because bytes of digest `D = msb_32(HMAC_SHA256(R,S))||R` are included as `f_x(254)` for each byte of the secret, this scheme is introducing redundant information about the secret into each generated polynomial. This breaks the information-theoretic security of the secret if an attacker obtains `T-1` shares. Some guesses for the secret `S` will result in a curve with `S`'s digest also on the curve, and some will not. The attacker learns information about `S` this way. If such an attacker also obtains partial knowledge of `S`, for example `m` out of `n` bytes of `S`, they now also have a brute force oracle to guess the remaining `n-m` bytes.

Typically applications that use the secret have limits on the number of attempts someone can make with the wrong secret, but this attack can allow an offline brute force that bypasses these protections.

The attack is:

1. 1) Guess `S_guessed` = `(known bits of S | guessed bits of S)`
2. 2) Compute the interpolated polynomials generated by the `T-1` shares and `S_guessed`
3. 3) Return `True` if the interpolated values at position 254 are a valid digest of `S_guessed`.

This is equivalent to executing the function `recover_secret()` in `shamir.c` for each `S_guessed`. Because of the construction of `D`, the probability of a random guess for `S` yielding a valid digest is approximately 2^-32, so if the number of unknown bits is much larger than 32, this will result in many false positives.

In a traditional Shamir secret share scheme there is no dependence between points on the polynomial, so you would not have this extra oracle. An application that allowed some small number of wrong guesses of the secret before blocking further guesses would prevent an attacker with `T-1` shares from violating the threshold property.

Naturally, the number of unknown bits needs to be small enough that an attacker can reasonably brute force them. So a 256-bit secret with the 128 bits known still requires on the order of 2^128 guesses to obtain the secret, comfortably outside the realm of possibility for a real-world attack.

The inclusion of hash digests of the secret is also present in the SatoshiLabs SLIP-39 design which is based on Shamir Secret Sharing. SatoshiLabs has publicly documented this design decision, related tradeoffs and corresponding security requirements. In other words, the computational security of the design in place of information-theoretic security is a known property of SLIP-39.

For `bc-shamir`, we are not aware of similar documentation. We see this as relevant since the practical use of `bc-shamir` is less strictly defined compared to SLIP-39, which allows its inclusion in other designs where the information-theoretic security may be desirable. Because the design modification changes the security guarantees provided by the classical Shamir's Secret Sharing scheme while still using its name, this may be confusing to users who expect these guarantees.

Impact:

This issue is mainly of theoretical importance.

Unlike classical Shamir's Secret Sharing, there is now a way for an attacker with `T-1` shares to learn information about the secret. Since most values for the guessed bits of `S` can now be determined to definitely be wrong, the probability distribution of each possible share value being correct is no longer uniform. This introduces the potential of an offline brute force attack. As described in the previous section, the cryptographic properties of the scheme are chosen so that this brute force attack is completely impractical unless there are additional, serious flaws in the implementation.

If there are such serious flaws, for example a broken random number generator or side channels that disclose large portions of the secret, this brute force attack can help an attacker with the required amount of shares to combine partial information of the secret to correctly brute force the full secret.

For more information, see the review https://hal.archives-ouvertes.fr/hal-03045663/document by Gabrielle de Micheli and Nadia Heninger for other examples of attacks that recover cryptographic secret values using partial information.

Recommendation:

We recommend documenting this design tradeoff in public documents on `bc-shamir` so that users are aware of the implications.

To avoid the described implications in future versions of the scheme, do not include the bytes of digest `D` as `f(254)` in the same shares as the secret itself. Instead create more shares where the bytes of `D` are the secret. This way a group of `T-1` conspirators guessing a few missing bytes of `S` can successfully reconstruct any `D` they want, thus making any value of `S` a valid guess and eliminating the reconstruction oracle.

This recommendation restores information-theoretic security at the cost of 32 more secret shares for a 32-byte digest. Also, putting bytes of `D` and `S` into the same shares doesn't have much of a benefit in terms of verification efficiency, because the reconstructed secret cannot be verified byte-wise. That is, the entire secret `S` needs to be recovered anyway before it can be verified against `D`.

Adding more shares that reconstruct `D` does not impact the ability to verify correctness of the reconstructed secret, since a malicious party with a corrupted secret share won't be able to generate corrupted digest shares that pass verification without first knowing the recomputed (wrong) secret. Therefore, with this scheme it is still possible to detect incorrect shares.

At the same time, for every wrong secret share, there exists a wrong digest share that will correctly interpolate to the "correct" digest of the wrong secret, passing verification. Finding this share should be difficult, however. This is analogous to Pedersen commitments, which are computationally binding but perfectly hiding.

Having one fixed secret value per share also removes the need to store this fixed value at `f(255)`. Both the bytes of `S` and `D` can be placed at `f(0)` for each polynomial. This means polynomials can be generated randomly instead of requiring calls to `interpolate` in `split_secret`.

4.5 BTM-018 — Out-Of-Bounds Write via generate_shards() in bc-sskr

Vulnerability ID: BTM-018 **Status:** Resolved

Vulnerability type: Insufficient Length Check

Threat level: High

Description:

Insufficient length checks in `sskr_generate()` and `generate_shards()` allow dangerous out-of-bounds writes.

Technical description:

This issue is related to BTM-005 (page 25).

The issue can be reproduced with the existing test code and a different master secret definition in test.c#L27:

```
char* master_secret_str =
 "116984227698422740011400116984227460561777777770011698422740011684924762056147 44";
```

This corresponds to a validly encoded master secret with a 40-byte raw data length:

```
(gdb) x/40xb master_secret
0x604000000590: 0x11  0x69  0x84  0x22  0x76  0x98  0x42  0x27
0x604000000598: 0x40  0x01  0x14  0x00  0x11  0x69  0x84  0x22
0x6040000005a0: 0x74  0x60  0x56  0x17  0x77  0x77  0x77  0x70
0x6040000005a8: 0x01  0x16  0x98  0x42  0x27  0x40  0x01  0x16
0x6040000005b0: 0x84  0x92  0x47  0x62  0x05  0x61  0x47  0x44
```

`make check` error result with Address Sanitizer:

```
==6919==ERROR: AddressSanitizer: dynamic-stack-buffer-overflow on address [...]
WRITE of size 40 at 0x7fffb88f2ee0 thread T0
    #0 0x497859 in __asan_memcpy (/home/user/target/bc-sskr/test/test+0x497859)
    #1 0x4cd411 in generate_shards /home/user/target/bc-sskr/src/encoding.c:188:13
    #2 0x4ca66a in sskr_generate /home/user/target/bc-sskr/src/encoding.c:249:20
    #3 0x4c8540 in test1 /home/user/target/bc-sskr/test/test.c:35:23
    #4 0x4c8004 in main /home/user/target/bc-sskr/test/test.c:62:5
[...]

Address 0x7fffb88f2ee0 is located in stack of thread T0 at offset 2400 in frame
    #0 0x4caf4f in generate_shards /home/user/target/bc-sskr/src/encoding.c:135
```

The `sskr_generate()` function documentation describes the following limitation:

```
*         master_secret_length: length of the master secret in bytes.
*                               must be >= 16, <= 32, and even.
```

However, only the `master_secret_length < 16` condition is actually detected and rejected in the code, see encoding.c:

```
if(master_secret_len < MIN_STRENGTH_BYTES) {
    return ERROR_SECRET_TOO_SHORT;
}
```

For `master_secret_length` > 32, the code will silently perform memory corruption. Internally, `sskr_generate()` calls `generate_shards()`, which has a different set of requirements on `master_secret_length` that also lack tests against values greater than 32.

As a result, encoding.c#L188 performs out of bounds writes by overwriting a fixed target buffer with an oversized variable-length buffer:

```
memcpy(shard->value, value, master_secret_len);
```

For reference, this is the struct definition with the `value` member that is 32 bytes in size and the target of the previously described `memcpy()` call:

```
typedef struct sskr_shard_struct {
    uint16_t identifier;
    size_t group_index;
    size_t group_threshold;
    size_t group_count;
    size_t member_index;
    size_t member_threshold;
    size_t value_len;
    uint8_t value[32];
} sskr_shard;
```

Impact:

Out-of-bounds writes may cause unexpected behavior, crashes or the execution of arbitrary code.

We briefly investigated two other public code bases by the client that use the affected library functionality:

Project	Preliminary Status	Comment
BlockchainCommons seedtool-cli	Not affected	Appears to have local mitigation checks against long inputs.
BlockchainCommons lethekit	Not affected	Appears to use a fixed 16-byte secret length.

Recommendation:

- Add an unified safety check function that validates `master_secret_len` properties.

- Add new error types in `sskr-errors.h` for `ERROR_SECRET_TOO_LONG` and `ERROR_SECRET_LENGTH_UNEVEN`.
- Alternatively, redefine the existing `ERROR_SECRET_TOO_LONG` as `ERROR_SECRET_INVALID_LENGTH` to cover all three cases.
- Investigate problematic use of the affected function in other projects.

4.6 BTM-015 — Internal serialize_shard() Function Ignores Length Parameter in bc-sskr

Vulnerability ID: BTM-015 **Status:** Resolved

Vulnerability type: Insufficient Length Check

Threat level: Low

Description:

The `static size_t serialize_shard()` function has a `size_t destination_len` parameter that is not used for memory access control of the provided `uint8_t *destination` buffer.

Technical description:

Depending on the future use of the `serialize_shard()` function, this may lead to security problems.

Compiler builds with verbose settings result in the following warning:

```
encoding.c:25:12: warning: unused parameter 'destination_len' [-Wunused-parameter]
    size_t destination_len) {
           ^
```

The corresponding the function definition: encoding.c#L22-L25:

```
static size_t serialize_shard(
    const sskr_shard *shard,
    uint8_t *destination,
    size_t destination_len) {
```

In our opinion, the combination of the `destination` data pointer and `destination_len` length value suggests to the caller that this function has some internal bounds checking and will respect the `destination_len` as the maximum amount of data that can be written to `destination`. In other words, the caller may rely on `destination_len` as a safety feature.

However, since the length value is unused, the following six write operations happen unconditionally:

```
destination[0] = id1;
    destination[1] = id2;
    destination[2] = (gt << 4) | gc;
    destination[3] = (gi << 4) | mt;
    destination[4] = mi;

    memcpy(destination + METADATA_LENGTH_BYTES, shard->value, shard->value_len);
```

This is a security concern due to potential out-of-bounds writes, for example on buffers that are too small to hold a shard of maximum length.

`serialize_shard()` is called from only one code position within the public `sskr_generate()` in encoding.c and is not otherwise accessible, so it is plausible that the current usage has no problematic edge cases.

Impact:

During this code audit, we have not found any practical code paths where out-of-bounds writes can occur. However, future changes may introduce edge cases with catastrophic out-of-bounds writes that may lead to unexpected program behavior, crashes, or the execution of arbitrary code.

Recommendation:

- Include defensive length checks on `destination_len` before performing any memory accesses.
- Evaluate how to flag error conditions to the caller.

4.7 BTM-010 — Compiler Hardening Flags are Missing

Vulnerability ID: BTM-010	**Status:** Resolved
Vulnerability type: Missing Defense-In-Depth Measures	
Threat level: Moderate	

Description:

We were unable to find common hardening flags in the `bc-shamir` and `bc-sskr` library build system definitions.

Technical description:

We particularly recommend adding stack canary protections to the default build.

1. Use of stack protection mechanisms via `-fstack-protector*` instrumentation options, for example `-fstack-protector-all`.
2. `-D_FORTIFY_SOURCE=2`, note that this also partially requires the `-O1` optimization level or above and may only have minor effects due to limited use of functions that can be fortified.

Overview using the checksec.sh tool on the standard build test binary, output shortened for print:

```
./checksec.sh --file=bc-shamir/test/test
RELRO           STACK CANARY      Symbols    FORTIFY Fortified Fortifiable FILE
Partial RELRO   No canary found   146) Symbols   No  0     2           bc-shamir/test/test
```

Additional references: see Debian Hardening.

Impact:

Compile-time hardening can partially or completely mitigate a number of attacks. Stack protection can make the difference between a (controlled) program crash and arbitrary code execution.

Recommendation:

- Add compiler flags to the default build settings that increase the defense-in-depth of the resulting library artifacts.
- Apply the hardening flags to testing code as well, as this may help detect security issues at an earlier stage.

4.8 BTM-007 — Sensitive Values Can Leak to Disk Due to Swapping

Vulnerability ID: BTM-007

Vulnerability type: Exposure of Sensitive Information

Threat level: Moderate

Description:

Sensitive values in `bc-shamir` can be forced to leak to disk by a local attacker inducing the process to be swapped out.

Technical description:

This attack is relevant on systems which have memory swapping support enabled.

Impact:

Sensitive information can leak to disk.

Recommendation:

- Use `mlock(2)` or `VirtualLock()` on Windows, or equivalent functions on other OS if available.

4.9 BTM-006 — Sanitizing of Sensitive Values in bc-shamir Possibly Removed by Compiler Optimization

Vulnerability ID: BTM-006	**Status:** Resolved
Vulnerability type: Exposure of Sensitive Information	
Threat level: Moderate	

Description:

The `bc-shamir` and `bc-sskr` libraries use `memset()` to sanitize sensitive data, but that may be optimized away by the compiler, depending on build settings.

Technical description:

In `shamir.c`, the `memset()` function is used to sanitize sensitive data, but that code may be optimized away. The `makefile.in` file specifies `CFLAGS += -O0` which mitigates this problem on common compilers since no aggressive optimization steps are requested. However this compiler flag does not solve the underlying issue and may be removed by consumers of the library, for example by a Linux distribution that uses a different set of default `CFLAGS`. In this case, the optimization may again cause leaks of sensitive data.

Note that `bc-crypto-base` already includes a specially hardened `memzero()` function in memzero.c, which is generally available to `bc-shamir` and can be used to work around the issue.

Impact:

Sensitive data may be leaked through unused memory.

Recommendation:

- Use `memzero()` from `bc-crypto-base` instead of `memset(3)`.

4.10 BTM-005 — Out of Bounds Accesses via split_secret() in bc-shamir

Vulnerability ID: BTM-005	**Status:** Resolved
Vulnerability type: Insufficient Length Check	
Threat level: Moderate	

Description:

Insufficient length checks in `split_secret()` allow out of bounds writes and segmentation fault crashes.

Technical description:

The error condition can be triggered with the following test case:

```
_test_shamir("", 3, 5, (uint8_t[]){1, 2, 4});
```

Note that there are other variants with non-zero `secret` string length that also cause the problematic edge case behavior.

The issues were observed through testing code, but we think it's plausible that external use of the library could trigger these issues as well.

Behavior without sanitizers:

```
# _test_shamir("", 3, 5, (uint8_t[]){1, 2, 4});
# was added manually to the `test_shamir()` unit tests in test/test.c
test/test
Segmentation fault
```

Behavior with sanitizers:

```
shamir.c:80:32: runtime error: index 4 out of bounds for type 'uint8_t [secret_length]'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior shamir.c:80:32 in
shamir.c:80:49: runtime error: unsigned integer overflow: 0 - 4 cannot be represented in type
 'unsigned int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior shamir.c:80:49 in
=================================================================
==1904==ERROR: AddressSanitizer: dynamic-stack-buffer-overflow on address 0x7ffc7befb544 at pc
 0x0000004e27f5 bp 0x7ffc7befb490 sp 0x7ffc7befb488
WRITE of size 1 at 0x7ffc7befb544 thread T0
    $0 0x4e27f4 in fake_random /home/user/target/bc-shamir/test/test-utils.c:88:12
```

```
    $1 0x4ca799 in split_secret /home/user/target/bc-shamir/src/shamir.c:80:9
    $2 0x4c8e52 in _test_split_secret /home/user/target/bc-shamir/test/test.c:20:20
    $3 0x4c849e in _test_shamir /home/user/target/bc-shamir/test/test.c:56:23
    $4 0x4c8309 in test_shamir /home/user/target/bc-shamir/test/test.c:84:3
```

High-level steps towards the issue:

- `_test_shamir()` is called with a short secret.
- Internally, `_test_split_shamir()` decides that the secret is of length 0.
- The production library function `split_secret()` is called with a short `secret_len` value.
- A subtraction in `split_secret()` results in an unsigned integer underflow due to the assumption that the `secret_len` value is larger than 4.
- `split_secret()` calls `create_digest()` with a large `uint32_t rdlen` length value.
- `fake_random()` performs the deadly out of bounds write into a high memory region, resulting in a segmentation fault.

Notably, the documented function contract at shamir.h#L49 requires `secret_length` to have a safe value with specific properties, but this is apparently not enforced in either the library nor in the unit test helper function within `bc-shamir`.

For the practical usage of `bc-shamir` as part of `bc-sskr`, the `generate_shards()` function in `bc-sskr` has guard clauses that appear to catch this issue for lengths smaller than intended and odd numbers, see encoding.c#L137-L143 and encoding.h#L16. However, it appears that a check for longer than expected secrets is missing. This topic is evaluated in BTM-005 (page 25).

Initial tests in `bc-shamir` with `master_secret_len > 32` point to logical issues with long secret inputs:

```
# test case
 _test_shamir("AAAAAAAAAAEAAAAAAAAAAAAAAAAAAAEFdfffffffffffffffffffffAAAAffffffffffffffAAAA", 3,
 5, (uint8_t[]){1, 2, 4});
make check
test: test.c:21: size_t _test_split_secret(const char *, uint8_t, uint8_t, char **): Assertion
 `result == shard_count' failed.
```

GDB view:

```
Thread 1 "fuzzer" hit Breakpoint 1, _test_split_secret (secret=0x7fffffffd7a0 "AAAAAAAAAAE",
 'A' <repeats 20 times>, "EFd", 'f' <repeats 23 times>, "AAAA", 'f' <repeats 15 times>, "AAAA",
 threshold=2 '\002',
    shard_count=7 '\a', output_shares=0x7fffffffd440) at fuzzer.c:128
128   assert(result == shard_count);
(gdb) info local
[...]
secret_len = 40
result_len = 280
__vla_expr0 = 280
result = -103
[...]
```

A shard count of `-103` is not meaningful and will likely lead to functional issues.

Impact:

Out of bounds writes may cause unexpected behavior, crashes or the execution of arbitrary code. Additionally, we observed edge cases with functionally incorrect results.

Recommendation:

- Add an error handling flag to `split_secret()`.
- Reject invalid parameters to the `split_secret()` function.
- Investigate the impact of functionally incorrect results.

5 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

5.1 NF-017 — Internal `combine_shards()` Function is Unused in `bc-sskr`

```
encoding.c:427:12: warning: unused function 'combine_shards' [-Wunused-function]
static int combine_shards(
           ^
```

It appears that this internal function is unused and that equivalent code exists within `sskr_combine()`. It is likely this code is no longer necessary and can be deleted.

5.2 NF-016 — Integer Comparisons with Different Signs

We briefly investigated `clang-11 -Wsign-compare` warnings related to integer comparisons.

Most warnings can be resolved by switching the iterator variable type from `int` to `size_t`.

One exception is in `sskr_generate()`, where `total_shards` is defined as `int` to accommodate negative error values. A possible solution to that is copy the `total_shards` value to a separate `size_t` variable or change the loop iteration variable to `int`.

bc-sskr

```
encoding.c:109:22: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
    for(int i = 0; i < groups_len; ++i) {
                   ~ ^ ~~~~~~~~~~
encoding.c:259:35: warning: comparison of integers of different signs: 'size_t' (aka 'unsigned
 long') and 'int' [-Wsign-compare]
    for(size_t i = 0; !error && i < total_shards; ++i) {
                                  ~ ^ ~~~~~~~~~~~~
encoding.c:334:26: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
       for(int j = 0; j < next_group; ++j) {
                      ~ ^ ~~~~~~~~~~
encoding.c:340:34: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
               for(int k = 0; k < groups[j].count; ++k) {
                              ~ ^ ~~~~~~~~~~~~~~~

test.c:47:22: warning: comparison of integers of different signs: 'int' and 'size_t' (aka 'unsigned
 long') [-Wsign-compare]
    for(int i = 0; i < input_indexes_len; i++) {
                   ~ ^ ~~~~~~~~~~~~~~~~~

test-utils.c:91:23: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
    for (int i = 0; i < count; i++) {
                    ~ ^ ~~~~~
```

```
test-utils.c:98:20: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
  for(int i = 0; i < len; i++) {
                 ~ ^ ~~~
test-utils.c:106:20: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
  for(int i = 0; i < len; i++) {
                 ~ ^ ~~~
test-utils.c:117:20: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
  for(int i = 0; i < len1; i++) {
                 ~ ^ ~~~~
test-utils.c:131:20: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
  for(int i = 0; i < len1; i++) {
                 ~ ^ ~~~~
```

bc-shamir

```
test-utils.c:87:20: warning: comparison of integers of different signs: 'int' and 'size_t' (aka
 'unsigned long') [-Wsign-compare]
  for(int i = 0; i < count; i++) {
                 ~ ^ ~~~~~
```

5.3 NF-014 — Memory Resource Leak in bc-sskr Unit Test

During testing of the `bc-sskr` library, the following Leak Sanitizer warning came up:

```
==23126==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 16 byte(s) in 1 object(s) allocated from:
    #0 0x49835d in malloc (/home/user/target/bc-sskr/test/test+0x49835d)
    #1 0x4e0c92 in hex_to_data /home/user/target/bc-sskr/test/test-utils.c:59:9
    #2 0x4c8593 in test1 /home/user/target/bc-sskr/test/test.c:29:32
    #3 0x4c8014 in main /home/user/target/bc-sskr/test/test.c:61:5
    #4 0x7fa66285ad09 in __libc_start_main csu/../csu/libc-start.c:308:16

SUMMARY: AddressSanitizer: 16 byte(s) leaked in 1 allocation(s).
```

The testing circumstances are very similar to the leak in `bc-crypto-base`.

It appears that the `uint8_t* master_secret` allocated via `hex_to_data()` in test.c#29 is never released via `free()`.

Given that this is located in unit test code, there is no practical security impact. Since the memory will be cleaned up after the program has exited, we also see no impact on the reliability of the current testing configuration.

5.4 NF-013 — Potentially Incorrect Documentation on shamir.c in bc-shamir

The Readme.md contains a section on file origins, which attributes the `shamir.c` and `shamir.h` files to `Copyright 2017 Daan Sprenkels`, dsprenkels/sss and notes "We have commented out sss_create_keyshares & sss_combine_keyshares".

We have been unable to find direct sources for `shamir.c` content in the upstream `sss` library. Additionally, the comments about `sss_create_keyshares` and `sss_combine_keyshares` functions do not appear to be relevant for this file. We therefore suspect that this documentation is incorrect.

5.5 NF-012 — Consider Use of Automated Code Formatting

At the moment, it appears that there is no official code style for the `bc-shamir` and `bc-sskr` libraries. `make lint` runs `cppcheck`, which focuses on bugs and not on code style.

We recommend using a tool such as `clang-format` and a `.clang-format` format specification file to define a consistent code style and automatically report or correct issues.

Example lines in locally introduced code:

```
for(uint8_t i=0; i< threshold-2; ++i, share+=secret_length) {
for(uint8_t i=threshold -2; i<shard_count; ++i, share += secret_length) {
random_generator(digest+4, secret_length-4, ctx);
```

Example line in upstream `sss` library code:

```
for (size_t bit_idx = 0; bit_idx < 8; bit_idx++) {
```

This is primarily for readability and code quality, and is only indirectly associated with security benefits.

5.6 NF-011 — Use Verbose Compiler Warnings

We highly recommend enabling verbose compiler warnings, either in the default build or as an optional build mode, in order to help developers to find unexpected or unintentional program behavior.

Primary compiler flags:

- `-Wall`, note this includes `-Wformat`
- `-Wextra`
- `-Wformat=2` to enable more format checks. The production code of the target libraries does not appear to have any essential `printf()` use, so in practice this may not be relevant for this project

Some projects decide to treat those warnings as errors via `-Werror`. This has some benefits as it enforces regular observation and cleanup of warnings, but may not be suitable for the desired development process. It may also cause problems for users with other compiler versions that have false positive warnings or new warning types which interrupt the build.

5.7 NF-009 — Incorrect Assertion in _test_shamir() in bc-shamir

Consider the following check in test.c#L74:

```
assert(equal_strings(secret, out_secret));
```

The check attempts to find cases where the originally given `const char* secret` and the newly reconstructed `char* out_secret` that is derived from the minimum number of shares result in **different logical values**.

The current design does a direct string comparison between the two values. However, the secrets are actually hex-encoded which allows different character capitalization. Therefore multiple distinct strings can map to the same logical decoded binary value. This means that a case-sensitive string comparison incorrectly flags some computations as problematic despite them being valid, resulting in false positive error detections via the assert.

This can be demonstrated with the following test case:

```
_test_shamir("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 3, 5, (uint8_t[]){1, 2, 4});
```

GDB view directly before the assert, captured from an equivalent fuzzer function call to the above:

```
Thread 1 "fuzzer" hit Breakpoint 1, _test_shamir (secret=0x7fffffffd7c0 'A' <repeats 48 times>,
 threshold=2 '\002', shard_count=7 '\a', recovery_share_indexes=0x7fffffffd830 "\003\004") at
 fuzzer.c:190
190   assert(equal_strings(secret, out_secret));
(gdb) info local
__vla_expr0 = 7
secret_len = 24
__vla_expr1 = 2
out_secret = 0x6060000002c0 'a' <repeats 48 times>
output_shares = {0x606000000020 "\001", 0x606000000080 "\001", 0x6060000000e0 "\001", 0x606000000140
 "\001", 0x6060000001a0 "\001", 0x606000000200 "\001", 0x606000000260 "\001"}
recovery_shares = {0x606000000140 "\001", 0x6060000001a0 "\001"}
```

As you can see, the input secret is `'A' <repeats 48 times>` in uppercase, but the recovered secret is `'a' <repeats 48 times>` in lowercase. This causes the string comparison between the two secrets to fail.

From our perspective, the assertion issue does not currently represent a security issue in production code, but it reduces the reliability and correctness of the testing suite.

Technical recommendation:

- Switch to a string comparison that ignores the character case.

- Decode the hex representation and compare the secret data bytes directly via `memcmp()`.

5.8 NF-008 — Insufficient Test Coverage in bc-shamir and bc-sskr

We recommend adding test cases to both libraries with test error conditions, different formatting, invalid inputs, and other edge cases in order to gain more consequential test coverage, and spot both existing problems and potential future regressions.

bc-shamir

The test suite currently consists of just two test cases that test valid cases with secrets of 16 and 32 byte length.

bc-sskr

The test suite currently consists of just one test case that tests a valid case with a 16 byte secret.

5.9 NF-003 — Use Strong Compiler Sanitizer Settings With Existing Unit Tests

At the time of the audit, the library tests are built without sanitizers. By adding one or more well-supported sanitizers, dangerous program behavior that would otherwise be missed can be spotted during testing.

Recommended

- Address Sanitizer and Undefined Behavior Sanitizer
- `CFLAGS="-fsanitize=address,undefined" LDFLAGS="-fsanitize=address,undefined"`
- Benefits: well-tuned default behavior, a low false-positive rate, and both are available under gcc and clang

Not Strongly Recommended, For Future Reference

- `CC=clang CFLAGS="-fsanitize=memory" LDFLAGS="-fsanitize=memory"`
- Memory Sanitizer is not as straightforward to use since non-instrumented code leads to false positives.
- Memory Sanitizer is incompatible with Address Sanitizer and is only available with the clang compiler.

5.10 NF-002 — Memory Resource Leak in bc-crypto-base Unit Test

During initial setup and testing of the `bc-crypto-base` dependency library, we observed the following Leak Sanitizer warning:

```
make clean && CC=clang CFLAGS="-fsanitize=address,undefined" LDFLAGS="-fsanitize=address,undefined"
 make check
[...]

./test
```

```
=================================================================
==32310==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 1 object(s) allocated from:
    $0 0x49834d in malloc (/home/user/target/bc-crypto-base/test/test+0x49834d)
    $1 0x4c8dc0 in _test_pbkdf2_data /home/user/target/bc-crypto-base/test/test.c:135:23
    $2 0x4c8d81 in _test_pbkdf2 /home/user/target/bc-crypto-base/test/test.c:147:3
    $3 0x4c8c8b in test_pbkdf2 /home/user/target/bc-crypto-base/test/test.c:157:3
    $4 0x4c7f89 in main /home/user/target/bc-crypto-base/test/test.c:174:3
    $5 0x75ef85cd8d09 in __libc_start_main csu/../csu/libc-start.c:308:16

Direct leak of 32 byte(s) in 1 object(s) allocated from:
    $0 0x49834d in malloc (/home/user/target/bc-crypto-base/test/test+0x49834d)
    $1 0x4c8dc0 in _test_pbkdf2_data /home/user/target/bc-crypto-base/test/test.c:135:23
    $2 0x4c8d81 in _test_pbkdf2 /home/user/target/bc-crypto-base/test/test.c:147:3
    $3 0x4c8c31 in test_pbkdf2 /home/user/target/bc-crypto-base/test/test.c:153:3
    $4 0x4c7f89 in main /home/user/target/bc-crypto-base/test/test.c:174:3
    $5 0x75ef85cd8d09 in __libc_start_main csu/../csu/libc-start.c:308:16

[...]

Direct leak of 16 byte(s) in 1 object(s) allocated from:
    $0 0x49834d in malloc (/home/user/target/bc-crypto-base/test/test+0x49834d)
    $1 0x4c8dc0 in _test_pbkdf2_data /home/user/target/bc-crypto-base/test/test.c:135:23
    $2 0x4c8cc8 in test_pbkdf2 /home/user/target/bc-crypto-base/test/test.c:158:3
    $3 0x4c7f89 in main /home/user/target/bc-crypto-base/test/test.c:174:3
    $4 0x75ef85cd8d09 in __libc_start_main csu/../csu/libc-start.c:308:16

SUMMARY: AddressSanitizer: 152 byte(s) leaked in 5 allocation(s).
```

It appears that the following memory allocation

```
uint8_t* key_data = malloc(key_len);
```

on test.c#L135 is not correctly deallocated, leading to resource leaks in the `test_pbkdf2()` call. Note that `data_to_hex()` uses `malloc()` internally to create a separate memory region, which is correctly deallocated via `free(key);`.

Given that this issue is located in unit test code, there is no practical security impact. Since the memory will be cleaned up after the program has exited, we also see no impact on the reliability of the current testing configuration.

6 Future Work

- **Extend Code Testing**

  We recommend extending the current unit tests and consistency checks that are present in both target libraries, as described in the report. The tests should cover common use cases of the exposed library functions as well as verifying correct error handling of invalid inputs. Additionally, we recommend extending the test suite with regression tests that detect unintentional changes of computation results and formats.
- **Add Fuzz Testing**

  We recommend introducing fuzz testing harnesses which manipulate various input fields to common library function calls. The goal is to gain broad test coverage over the exposed library functions and discover potential edge cases, for example through the use of sanitizer tools.

  The fuzzing code and filtered fuzzing corpus of this audit have been shared with the client.
- **Retest of findings**

  When mitigations for the vulnerabilities described in this report have been deployed, a repeat test should be performed to ensure that they are effective and have not introduced other security problems.

  Note that the majority of findings have already been fixed and retested during this code audit.
- **Regular security assessments**

  Security is an ongoing process and not a product, so we advise undertaking regular security assessments and penetration tests, ideally prior to every major release or every quarter.

7 Conclusion

We discovered 2 Elevated, 5 Moderate, 2 Low and 1 High-severity issues during this penetration test.

Overall, we found the libraries to be more fragile and less hardened than expected given their purpose and small size. Due to the design characteristics and limitations of the C language, it is generally necessary to implement library interfaces defensively and with strict checks to avoid dangerous edge cases, which was not done uniformly in the `bc-shamir` and `bc-sskr` libraries. We discovered a number of issues in different functions which lead to memory corruption, undefined behavior and other highly problematic results for specific edge cases, for example with oversized secrets. Whether these edge cases are reachable accidentally through bugs or intentionally by an attacker depends on the functionality and additional input sanitization of the caller programs which make use of the audited libraries. To our knowledge, replacing the libraries with reimplementations in a memory-safe language is not an option for the client, therefore this class of issues will likely continue to represent a security risk and warrants extra development time and testing.

We identified two cases of incomplete sanitization and incomplete protection of sensitive values in memory, as well as missing hardening flags during compilation that result in a lack of buffer overflow protection. Improvements in this area can reduce the likelihood of information leaks and mitigate some attacks.

Our analysis of the cryptographic design of the `bc-shamir` library revealed a notable difference in security guarantees from the classical Shamir's Secret Sharing scheme. This is due to an inherited modification of the scheme that includes a byte of a digest of the secret as an extra share in each polynomial. Including this redundant information removes the information-theoretic security provided by SSS, meaning that `bc-shamir` only provides computational security: an attacker with `T-1` shares can reject guesses of the secret whose digest bytes are not points on the interpolated polynomials. In practice, this is not a problem as long as the secret is sufficiently random and the cost of brute forcing is sufficiently high, but we recommend documenting this design aspect better.

We generally recommend fixing all findings and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced. Note that for the majority of findings, this has already been completed during the code audit period after code updates by the client, which we see as positive.

Finally, we want to emphasize that security is a process – this code audit is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your organization's information security. We hope that this report and the detailed explanations of our findings will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing team

Stefan Marsiske	Stef runs workshops on radare2, embedded hardware, lockpicking, soldering, gnuradio/SDR, reverse-engineering, crypto topics. He also conducts training on OPSEC for journalists and NGOs. Last year he scored in the top 10 of the Conference on Cryptographic Hardware and Embedded Systems Challenge. He operates one of the most comprehensive databases on EU policy-making. He played important roles in the founding of some central European hackerspaces, tech events and various NGOs. Long before all this he worked for Siemens doing reverse-engineering, UNIX development, telco security and innovation management.
Christian Reitter	Christian is an IT Security Consultant with experience in the area of software security and security relevant embedded devices. After his M. Sc. in Computer Science, he has worked as a developer and freelance security consultant with a focus on fuzzing research. Notable published research includes several firmware vulnerabilities in popular cryptocurrency hardware wallets, including remote code execution, remote theft of secret keys and circumvention of 2FA protection. He has also discovered multiple memory issues in well-known smartcard driver stacks.
Jonathan Levin	Jonathan is a cryptographic researcher with a specialization in network protocols and post-quantum cryptography. He has performed cryptographic software audits on various projects including attribute-based credential systems, virtual private network protocols, and PKI implementations.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.