Audit-Report Monocypher Crypto Library 06.2020

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

Index

Introduction

> *"I wanted the simplicity of TweetNaCl, only with the latest and greatest at the time (Chacha20, Blake2b, Argon2i). That's what I started with: an unassuming clone of TweetNaCl with a couple tweaks. Then I began to improve performance, and Monocypher quickly became its own thing: a compact, portable, opinionated, fast crypto library."*

From https://monocypher.org/why

This report describes the results of a cryptography audit carried out by Cure53 against the Monocypher library in version 3.1.1. The work was requested by the Monocypher maintainer and funded by the Open Technology Fund program. Cure53 completed the project in late June 2020.

To give more details on the context, Cure53 examined Monocypher version 3.1.1, available as a downloadable tarball for auditing purposes. Precisely in Calendar Week 26 of 2020, two members of the Cure53 team spent six days on preparations, examining the scope, as well as documentation. The methodology in use was white-box, which stems from the nature of the project and all relevant code being available as Open Source.

The project started on time and progressed efficiently. The communications during the test were done in a dedicated private Slack channel into which the Cure53 invited the maintainer of the Monocypher library and software. Communications were fluent and no major roadblocks were experienced during the test, the audit was assisted by the well-readable code that did not offer an attack surface or delays related to lack of clarity or confusion.

The testing and auditing managed to unveil a total of six findings, four of which were classified to be security vulnerabilities of varying severity and two represent general weaknesses marked by lower exploitation potential. The most serious risks were set at *Medium* severity-levels, which contributes to a positive impression. Note that one of the findings was live-reported to the maintainer while the test was still ongoing and first fixes were created by the maintainer and inspected by Cure53.

In the following sections, the report will first shed light on the scope and key audit parameters, inclusive also of a list of files sorted by priority. Next, all findings will be discussed in a chronological order alongside technical descriptions, as well as PoC and mitigation advice when applicable. Finally, the report will close with broader conclusions about this 2020 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team’s observations and collected evidence. Tailored hardening recommendations for Monocypher are also incorporated into the final section.

Scope

- **Monocypher Crypto Audit**
  - Monocypher 3.1.1
    - https://monocypher.org/download/monocypher-3.1.1.tar.gz
    - https://github.com/LoupVaillant/Monocypher/releases/tag/3.1.1
  - Files receiving primary focus were:
    - *src/monocypher.h*
    - *src/monocypher.c*
    - *src/optional/monocypher-ed25519.h*
    - *src/optional/monocypher-ed25519.c*
  - Files receiving secondary focus were:
    - *src/deprecated/aead-incr.c*
    - *src/deprecated/aead-incr.h*
    - *src/deprecated/chacha20.c*
    - *src/deprecated/chacha20.h*
  - **Sources were made available to Cure53**
  - **Test-supporting material and documentation were shared with Cure53**

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *MON-01-001*) for the purpose of facilitating any future follow-up correspondence.

MON-01-001 Monocypher: *CSPRNG* handling unspecified on Darwin *(Low)*

It was found that Monocypher's documentation specified a set of recommended random number generators for many popular operating systems, did not have a specified set of handlers or system calls for seeding or obtaining randomness on Darwin-based systems, including MacOS. While the documentation specifies *linux/random.h* for Linux platforms, *arc4random_buf()*[1] for BSD and *BCryptGenRandom()* for Microsoft Windows, no similar target is specified for MacOS.

It is recommended for the */dev/urandom* provider to be explicitly specified on Darwin-type systems in order to lessen the ambiguity that may be present when Monocypher is deployed on these types of systems.

MON-01-004 Monocypher: Elligator2 test vectors not replicated *(Low)*

It was observed that Elligator2 was the only cryptographic primitive in Monocypher to have custom test vectors which were not independently verified for correctness or interoperability by comparing output with any other implementation.

Not verifying an exotic primitive for interoperability or correctness can lead to instances in which incorrect behavior may not be detected, resulting in security and privacy degradations. After this issue was reported and a third-party implementation was recommended for comparison[2], the Monocypher team replicated a sufficient number of test vectors across both implementations. It was verified that interoperability was achieved and that both implementations exhibited the same output to the chosen test vectors.

***Note:** The issue was reported to the maintainer while the test was still ongoing. The fix inspected by Cure53 was found valid and working.*

[1] The security of *arc4random_buf()* on various BSD-type systems is unclear, due to major flaws in the underlying RC4 cryptographic primitive: https://security.stackexchange.com/a/172905

[2] https://github.com/Kleshni/Elligator-2

MON-01-005 Monocypher: Safe/unsafe functions not differentiated (*Medium*)

It was found that the naming conventions of Monocypher does not organically indicate that some raw primitives, such as for example *crypto_chacha20*, are largely intended as underlying constructions for *crypto_lock* or *crypto_lock_aead*. They should not be accessed directly. This is despite large differences in safety and being fool-proof between high-level and low-level API functions.

As a result, Monocypher exposes its cryptographic library's API with all functions having the same flat hierarchy.

This manner of designing a cryptographic library can result in users not understanding the significantly different security guarantees provided by raw construction, such as *crypto_poly1305,* when compared with *crypto_lock*. Therefore, they may be used interchangeably.

It is recommended for the API to adopt an entirely new naming scheme. The documentation needs to be re-organized to reflect a hierarchy between preferred, safe-by-default and high-level functions versus the less safe and volatile, low-level primitives that should only be used in specific circumstances.

MON-01-006 Monokex: Authentication code verification unspecified (*Medium*)

The Monokex specification does not properly specify the message authentication code (MAC) logic for communicated messages. The derivation of MAC values (denoted by a number prefixed with *T*, e.g. *T1 = Blake2b(H9 || one)*) is indicated in the protocol flow but the purpose of *T* as a message authentication code and the point and method at which it is verified as such by either party is not specified anywhere.

It is recommended to properly specify the function of *T* values and the point in the protocol flow at which they are authenticated locally by the respective principals. Furthermore, it is recommended that the reasons for adopting this construction, which does not benefit from the security proofs accompanying constructions such as HMAC or Poly1305, is justified in the Monokex specification (see MON-01-003).

Miscellaneous Issues

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

MON-01-002 Monocypher: Potential for nonce misuse *(Medium)*

In order to preserve the discipline of having no dependencies in system-specific interfaces or utilized external code, Monocypher requires users to independently derive and specify nonces, despite the requirement that they be used only once. The latter is done in order for their cryptographic primitives to retain fundamental security guarantees.

It is unclear if this tradeoff is necessary or worthwhile. It may be argued that by not offering support for system calls, developers employing Monocypher will be forced to hack together solutions of varying quality and security to handle nonce generation and related operations.

Given that the mishandling of nonces can result in catastrophic outcomes on the security guarantees of primitives offered by Monocypher, it is recommended to consider generating nonces for the user internally, returning them upon successful encryption. Finally, and perhaps unrelatedly, it is unclear whether avoiding the allocation of nonces into memory means effective security improvements for Monocypher, given that:

- Nonces are fixed-length, and thus memory safety issues are unlikely to occur.
- Nonces are public and assumed to be known by the attacker in virtually all symmetric primitive security reductions.

MON-01-003 Monokex: Key derivation not justified by security reduction *(Info)*

Monokex patterns are clearly derived from handshake patterns originally specified within the Noise Protocol Framework. For example:

- The X Monokex pattern matches the protocol flow and claimed security properties of the X Noise handshake pattern.[3] [4]
- The XK1 Monokex pattern matches the protocol flow and claimed security properties of the XK1 Noise handshake pattern.[5]

[3] https://noiseexplorer.com/patterns/X/

[4] Also modeled using the Verifpal protocol verifier: https://verifhub.verifpal.com/c5603ff17dde570e107d39030844c40b

[5] https://noiseexplorer.com/patterns/XK1/

The main difference between Monokex patterns and Noise handshake patterns appears to be the replacement of Hash-Based Key Derivation (HKDF) with regular, raw hashing. While Monokex claims substantial performance benefits due to this change, especially in embedded microcontroller environments, it is unclear whether these performance improvements warrant the loss of the indifferentiability guarantees provided by the HKDF construction[6]. Namely, there is no equivalent security reduction allowing for a comparison between proofs of security obtained on Noise handshake patterns[7] and those potentially obtained from the Monokex constructions. Finally, while Monokex employs keyed *Blake2b* hashing, this is not sufficiently clear in the specification, which appears to suggest that *MAC* values are obtained purely by appending tuples together and using the result as a hash input.

This observation does not come with any particular recommendation. However, it would be advisable to expand the Monokex specification, adding more concrete arguments on the eschewing of HKDF for raw hashing.

[6] https://eprint.iacr.org/2010/264.pdf
[7] https://link.springer.com/chapter/10.1007/978-3-030-45374-9_12

Conclusions

After spending six days on the Monocypher scope during this June 2020 project, two members of the Cure53 team can confirm that the provided C code held well to their scrutiny. Few findings with limited severities evidence a good security premise of Monocypher. What is more, the code is exceptionally clean and demonstrates a clear focus on security features. It relates to typical targets around embedded environments, for instance by avoiding unnecessary memory allocations.

The findings highlight some exceptions linked to undocumented behavior (MON-01-001) and a minor lack of rigor in test vectors (MON-01-004). Beyond these, no serious issues were found in the Monocypher code itself. However, some issues were spotted in the cryptographic library design (see MON-01-005 and MON-01-002). Finally, the Monokex protocol suite's specification was found to be lacking critical details on the behavior of its Message Authentication Codes (MON-01-006). In the same realm, Cure53 also points out the necessity to justify its relatively bareboned key derivation mechanism (MON-01-003).

In conclusion, while the Monocypher code is well-written and supported by clean, documented code and a suitable amount of test vectors, the high-level design of the Monocypher's developer-exposed API could use more refinement (MON-01-005), as could the specification of the Monokex suite of protocols (MON-01-006, MON-01-003). Since no issues of *High*- or *Critical*-severity could be spotted in the timeframe available for this audit, Cure53 concludes this 2020 assessment on a positive note.

Cure53 would like to thank Loup Vaillant-David who maintains Monocypher for his excellent project coordination, support and assistance, both before and during this assignment. Special gratitude needs to be extended to Open Technology Fund Washington for sponsoring this project.