Fine penetration tests for fine websites

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

Security Review & Audit Report rustls 05.-06.2020

Cure53, Dr.-Ing. M. Heiderich, MSc. N. Krein, Dr. N. Kobeissi, Dipl.-Inf. G. Kopf

Index

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

Introduction

"Rustls is a modern TLS library written in Rust. It's pronounced 'rustles'. It uses ring for cryptography and libwebpki for certificate verification."

From https://github.com/ctz/rustls

This report describes the results of a security assessment targeting the rustls complex, which is a TLS library written in Rust. While the project was completed by Cure53, it should be noted that this audit was requested and sponsored by CNCF. This can be seen in connection to rustls being a frequent dependency for several CNCF projects, for instance Linkerd[1].

In terms of timeline and resources, the work was executed in late May and early June 2020. Four members of the Cure53 team selected on the basis of best-matching expertise were tasks with this examination of rustles and spent a total of thirty days on the project. In cooperation with CNCF and rustls, Cure53 worked against a two-pronged scope. The primary target was the mentioned rustls library, while the secondary items of relevance entailed peripheral libraries and key dependencies, such as *rustls-native-certs, sct.rs, ring* and *webpki*.

Two work packages were derived from that scope, with WP1 addressing a cryptography and performing a code audit of rustls in versions 0.1.6 or newer. Rounding up the scope, WP2 centered on audits aimed at the *ring, webpki, sct.rs,* and *rustls-native-certs* libraries. All in all, this review encompassed five different but very much related scope objects, with rustls standing as a key priority for the auditors.

Because software in scope is all available as open source on GitHub, the project's methods correspondingly highlight white-box approaches. Before and during the assignment, Cure53 was in frequent contact with the maintainers, receiving a briefing about their expectations as well. A private Slack channel was created by Cure53 to enable communications. Representatives of each project in scope were invited to contribute to the discussions with feedback, scope clarifications, answers to questions and so forth. All exchanges were smooth and helpful, supporting the audit team in terms of correct focal areas.

The audit and reviews progressed efficiently and without any hindrance. Cure53 periodically updated the maintainer teams about test coverage, verifying that the project was going in the right direction. The assessment was concluded as planned in early June 2020 without any delay. The testing team identified only four minor findings, none of them classified as vulnerabilities but rather as general weaknesses. They mostly

[1] https://github.com/linkerd/linkerd2-proxy/blob/5264573433ceea....0a0/linkerd/identity/src/lib.rs

represent security recommendations or noteworthy yet unexploitable issues. None of the findings appear to indicate the presence of bug patterns, pointing instead to minor oversight or manifesting further hardening options for the code and its reliability

In the following sections, the report will first shed light on the scope and key test parameters. After that, a dedicated chapter about test coverage and methodology will detail the areas the Cure53 looked at without spotting findings. Next, all flaws 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 rustls are also incorporated into the final section.

Scope

- **Cryptography Review & Code Audit against rustls and related libraries**
  - **WP1**: Cryptography Review & Code Audit against *"rustls v0.1.6"* or newer
    - https://github.com/ctz/rustls
  - **WP2**: Code Audits against libraries: *ring, webpki, sct.rs, rustls-native-certs*
    - https://github.com/ctz/rustls-native-certs
    - https://github.com/ctz/sct.rs
    - https://github.com/briansmith/ring
    - https://github.com/briansmith/webpki
- **Sources were shared with Cure53 (all available as OSS)**

Test Methodology and Coverage

This section describes the testing methodology and resulting coverage of the security audit against rustls and its different components. While the first of the following sections covers the broader and more typical aspects of the code quality, the next two sections report on deep dives into specific areas of rustls. Especially the latter two chapters try to give a more detailed overview of common security issues in Rust code, while also focusing on the context and purpose of rustls itself.

General Code Quality Checks

Cure53 here addresses broader "over-the-top" aspects of the audit. Since this is part of the initial tasks when covering a new project to audit, the process described here helps with kicking off the security analysis and gives an impression of the targeted source code. Additionally, some of the aspects listed here help to uncover potential low-hanging fruit (as in: easily identifiable security issues) with the aid of automatic processes.

Starting a security analysis of any application with given source code, it is usually helpful to run an automated scanner that might be able to instantly spot red flags inside the project's codebase. For applications written in Rust, *Clippy*[2] is an excellent lint collection to spot inconsistencies and warnings inside the targets. Running it across the entire scope items of this audit revealed a few recommendations about redundant imports and readability improvements. *Clippy* also warned about redundant *struct* fields and *pass-by-reference* optimizations. However, none of the generated warnings led to any visible security impact. Since they are also easily reproducible, they are omitted from this report.

Having a good testing setup to easily confirm expected code correctness in the form of unit or fuzz testing is another important factor of modern software development, especially for codebases that are expected to work correctly in a security sensitive context. Rustls heavily falls into this category. Cure53 ran every unit-test for the given scope items and kept track of untested components for deep dives. Rustls makes use of *libfuzzer-sys* (a wrapper around LLVM's *libfuzzer*) with *corpora* around client and server messages as sample inputs. Since the given test-cases already covered a wide range of sample inputs for alert and handshake messages, Cure53 did not focus on fuzzing attempts any further.

Lastly, Cure53 made sure that all dependencies of the given code were correct. However, since *Cargo* with its *cargo update* functionality already makes it easy to keep track of any out-of-date components, it was not surprising to find that dependencies were generally up-to-date. A few exceptions include minor version changes for several

[2] https://github.com/rust-lang/rust-clippy

Cargo dependencies. Cure53 made sure that they did not introduce fixes for the already known security issues. Since no problems were encountered here either, Cure53 continued to focus on targeted code audits for selected functionalities across the scope.

Code Robustness Analysis

This section covers some details about the audit process of language specifics without going too much into details about the correctness of all provided functionality and protocols, which are instead addressed in WP2.

As Rust is a modern language with built-in memory safety and error handling features (e.g., via sum-types), an emphasis was placed on general best practices for resilient implementations. Such practices include - from a meta and advancement-oriented point of view - ensuring the totality of the used functions, limiting the computational complexity of the code at runtime, avoiding unsafe constructs (e.g., the C FFI or *unsafe* blocks) and ensuring correct failure modes. The code generally makes a positive impression in this regard. One recommendation for further strengthening the implementation is provided in TLS-01-002.

Furthermore, besides analyzing general coding best practices, Cure53 investigated the correctness of the code in terms of its application logic. This includes, for instance, aspects like the correctness of the TLS state machine implementation, proper handling of integer arithmetic and possible truncation issues, as well as the correctness of the protocol parsing and generating code, rounded up by the handling of sensitive memory contents. One observation regarding integer handling in the code is described in TLS-01-004.

Particular focus concerned reviewing code that might contain severe logic issues, such as the hostname verification code in *webpki.* Even subtle problems in such implementations can lead to critical consequences, as previous research results indicate[3]. This led to the observation described in TLS-01-003.

[3] https://ioactive.com/pdfs/PKILayerCake.pdf

Auditing Protocol Handlers and Cryptographic Primitives

From a cryptographic point of view, this part of the assessment focused on identifying implementation issues, such as side-channel problems (e.g., due to non-hardened comparison functions, branches depending on secret bits etc.), functional correctness of the cryptographic primitives, incorrect NONCE handling, etc. All analysis steps are described in detail next.

Correctness of Supported Protocols

Rustls offers a comprehensive implementation of TLS 1.2 and TLS 1.3, choosing not to support prior (and now deprecated) versions of TLS.

The goal of this audit component was to verify the correct implementation of TLS 1.2 and TLS 1.3, the resilience of the state machine and to the codebase against misbehaving or malicious connections from third-party clients and servers. In addition, the auditors sought to ensure that cryptographic primitives were employed correctly.

The following elements of TLS 1.2 and TLS 1.3 were checked for functional correctness and the absence of unexpected or exploitable behavior:

- **State Machines:** A lot of time was spent on verifying the functional correctness of the state transition and session state management logic in rustls. This includes the full protocol handshake and session between client and server in TLS 1.2 and TLS 1.3. Each state transition was checked in the code and special attention was given to detecting invalid state transitions or non-abortions in scenarios in which session negotiation is not supposed to be completed successfully.
- **Forward Secrecy and Session Resumption:** The auditors wanted to determine if forward-secure modes in TLS 1.3 were being activated at the appropriate moments in session handshake initialization and handling.
- **QUIC:** The QUIC implementation was checked for conforming to the IETF specification and achieving functional correctness. No other checks were made.

Core Cryptography of *ring*

The *ring* cryptographic library acts as a provider of various cryptographic primitives and constructions by exposing the most appropriate implementation type for each primitive via a Rust API. These low-level implementations are written in C, ASM or in Rust. depending on what is most appropriate for the protocol.

The following elements were examined during this audit:

- **AEAD Constructions:** The correct implementation of block-cipher-based AEADs, such as AES-GCM, was verified. Critical elements, such as nonce generation, were checked to adhere with the requisite uniform randomness and uniqueness standards. ChaCha20 bindings were similarly checked for functional correctness.
- **Elliptic Curve Cryptography:** The fundamental elliptic curve arithmetic for Curve25519 was checked for functional correctness. The primitives implemented on top of it, such as X25519 for Diffie-Hellman and Ed25519 for digital signatures, were also verified for functional correctness.
- **Constant Time Comparison:** Each of the constant-time comparison functions exposed by the *ring* library was checked for functional correctness.
- **Poly1305:** Poly1305's bindings were checked for correct usage. However, the underlying ASM implementation of Poly1305 was not checked in any way and is simply assumed to be correct.
- **HKDF:** The HKDF implementation was checked for functional correctness.
- **RSA, ECDSA:** Special attention was paid to the supported RSA PKCS standards and offered padding methods. For ECDSA, the auditors verified non-applicability of recent attacks that focus on biased nonce generation, such as LadderLeak.[4]
- **AES:** Special attention concerned making the determination about the offered AES implementations' presenting side0channel characteristics due to s-box or similar constructions.

[4] https://eprint.iacr.org/2020/615

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

Miscellaneous Issues

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

TLS-01-001 Rustls: Formally Verified Cryptography Recommendations *(Info)*

While rustls offers a variety of reliable implementations of cryptographic primitives in ASM, C and Rust, none of the provided primitives benefit from formally verified functional correctness or resistance to side-channel attacks. The *EverCrypt* project[5] is an open-source cryptographic provider structured very much in the same way as *ring* (in that it provides an interface for primitives implemented in different languages). The difference is that the *EverCrypt* library also provides formally verified guarantees of side-channel attack resistance and functional correctness.

In the future, it might be worthwhile to investigate the adoption of some or all *EverCrypt* primitives either into *ring* or into rustls directly, as there does not seem to be any performance cost to doing so, while the increased assurance on the reliability and correctness of the primitives could be a significant advantage.

TLS-01-002 Rustls: Unchecked usage of *unwrap* *(Info)*

During the review of rustls, it was found that the code does not always statically enforce a correct handling of *Option* values. This includes code constructs such as *if foo.is_some() { … foo.unwrap() }*, which are obviously safe but could potentially be handled more strictly by using the *if let* syntax. However, there were also instances that are harder to verify. To give an example, one such instance is described below.

Affected File:
rustls/src/client/tls13.rs

Affected Code:
```
fn handle_new_ticket_tls13(&mut self, sess: &mut ClientSessionImpl, m: Message)
-> Result<(), TLSError> {
let nst = extract_handshake!(m,
HandshakePayload::NewSessionTicketTLS13).unwrap();
```

It can be observed in the code that the *handle_new_ticket_tls13* function attempts to unwrap an *Option* value. Before unwrapping, no checks are performed on whether the *Option* actually represents a *Some*. This pattern can generally lead to a *panic!*, which in

[5] https://www.microsoft.com/en-us/research/publication/evercrypt-a-fast-ver...rm-cryptographic-provider/

turn might lead to a DoS condition. It should be noted that the above *unwrap* is in fact safe, which is however not directly evident from the *handle_new_ticket_tls13* function. As the code already operates in a *Result* context, one solution to address the issue could be to rely on a syntax like *extract_handshake!(m, Handshake-Payload::NewSessionTicketTLS13).ok_or(SomeError)?*.

TLS-01-003 Webpki: Support for Non-Contiguous Subnet Masks *(Low)*

While reviewing the *webpki* implementation, it was found that the name constraints code allows for non-contiguous subnet masks. This means that a subnet mask like *42.42.42.42* would be treated as valid by the verifier, which might have unintended consequences.

Affected File:
webpki/src/name.rs

Affected Code:
```
loop {
    let name_byte = name.read_byte().unwrap();
    let constraint_address_byte = constraint_address.read_byte().unwrap();
    let constraint_mask_byte = constraint_mask.read_byte().unwrap();
    if ((name_byte ^ constraint_address_byte) & constraint_mask_byte) != 0 {
        return Ok(false);
    }
    if name.at_end() {
        break;
    }
}
```

Typically, subnet masks should be contiguous and the presence of a non-contiguous mask might indicate a typo (such as *225.255.255.0* vs. *255.255.255.0*), or potentially an attempt to bypass an access control scheme. Therefore, it is recommended to treat certificates containing non-contiguous subnet masks in their name constraints as invalid.

TLS-01-004 Rustls: Data Truncation in DER Encoding Implementation *(Low)*

While reviewing the DER parsing and generating code, it was found that the *wrap_in_asn1_len* function in the *rustls/src/x509.rs* file does not operate properly on input sequences longer than *0xffff* bytes. The code excerpt below provides an example.

Affected File:
rustls/src/x509.rs

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

Affected Code:

```
fn wrap_in_asn1_len(bytes: &mut Vec<u8>) {
        let len = bytes.len();

        if len <= 0x7f {
        bytes.insert(0, len as u8);
        } else if len <= 0xff {
        bytes.insert(0, 0x81u8);
        bytes.insert(1, len as u8);
        } else if len <= 0xffff {
        bytes.insert(0, 0x82u8);
        bytes.insert(1, ((len >> 8) & 0xff) as u8);
        bytes.insert(2, (len & 0xff) as u8);
        }
}
```

It can be observed that the code only handles input smaller than *0xffff* bytes. This code is used for creating DER sequences later in the code flow. As the function *wrap_in_asn1_len* fails silently, this behavior could result in creating DER output that does not match the intended semantics.

It is recommended to address the issue by introducing a more explicit failure mode - for instance by making the function *wrap_in_asn1_len* return an *Error* type.

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

Conclusions

During this 2020 project targeting rustls and its surroundings, Cure53 was unable to uncover any application-breaking security flaws. After spending thirty days on the scope in late May and early June of 2020, the team of auditors considered the general code quality really good and can attest to a solid impression left consistently by all scope items. Naturally, this is partially thanks to the usage of Rust as the preferred language for the entire implementation of the rustls project.

The examined code was consistently well-documented and readable, demonstrating that security processes are ingrained in the development and documentation processes at the rustls complex. Both from a design point of view as from an implementation perspective the entire scope can be considered of exceptionally high standard. Using the type system to statically encode properties such as the TLS state transition function is one just one example of great defense-in-depth design decisions. Furthermore, the code is typically explicit about the expected input and the possible failure modes.

The parsing code, for example deployed during certificate handling, relies on a strict approach. This is evident from it often demanding that all available input has to be consumed by a parser. No overly long messages are accepted and the general approach of using a combinator-like scheme for parsing message contents furthermore makes the parser implementation easily readable. While a number of recommendations have been provided (see TLS-01-002 to TLS-01-004) in order to further strengthen the implementation, no directly exploitable weaknesses could be identified.

From a cryptographic point of view, the code left a positive impression as well. It appears to have been developed with all previously known issue-types in mind; furthermore, its missing support for insecure or outdated protocols and primitives indicates a security-conscious development approach. Rustls' implementation of TLS takes security and cryptographic engineering very seriously. A very high standard of care is observable in engineering a reliable, complete, well-implemented TLS stack that follows the standard specification. Cryptographic operations are implemented and managed with great care. It can be said that in terms of cryptographic engineering, the level of care and quality exhibited by this codebase is exceptional both across the protocol layer and the primitives' layer.

No issues were found with regards to the cryptographic engineering of rustls or its underlying *ring* library. A recommendation is provided in TLS-01-001 to optionally supplement the already solid cryptographic library with another cryptographic provider (*EverCrypt)* with an added benefit of formally verified cryptographic primitives. Overall, it is very clear that the developers of rustls have an extensive knowledge on how to

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

correctly implement the TLS stack whilst avoiding the common pitfalls that surround the TLS ecosystem. This knowledge has translated reliably into an implementation of exceptional quality.

The technical observations additionally shined through in the multiple discussions within the shared Slack channel where both Cure53 and the rustls development team were actively engaged. Misconceptions and questions about the provided source code were quickly resolved. The developer's intent to provide a high-quality TLS implementation is very clear and this goal can be considered as achieved successfully. With that said, Cure53 has no negative feedback about security at rustls. Minor recommendations here and there are always possible for any project, but this does not change the fact that there is really not much to improve at rustls. Cure53 had the rare pleasure of being incredibly impressed with the presented software.

Cure53 would like to thank Dirkjan Ochtman, Joe Birr-Pixton, Oliver Gould and Brian Smith as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.