Schnorrkel Library Review

Web 3.0 Technologies Stiftung

July 24, 2019 – Version 1.0

Prepared for

Jeff Burdges

Peter Czaban

Prepared by

Ava Howell

Thomas Pornin

Executive Summary

Synopsis

During the summer of 2019, Web 3.0 Technologies engaged NCC Group to conduct a cryptographic security review of the Schnorrkel rust crate. Schnorrkel implements Schnorr signatures, verifiable random functions (VRFs), hierarchical deterministic key derivation, and multisignatures. It operates over the `ristretto255` group, a prime order elliptic curve group constructed on top of Curve25519. The entirety of the Schnorrkel crate and the behavior of its dependencies were in scope.

Scope

- **Schnorrkel Crate** The main crate was in primary scope for the assessment. This included checking the correctness and implementation soundness of all of the functionality implemented in the crate. The current state of the Schnorrkel repository was extracted at the start of the engagement and used for the review.[1]
- **Primary Dependencies** Schnorrkel's primary dependencies were reviewed for correctness to the extent that they were used by Schnorrkel. These included:
  - `clear_on_drop`, used to remove secret key material from memory once no longer needed.
  - `dalek-cryptography/merlin`, used to compute transcripts and do hashing operations.
  - `dalek-cryptography/subtle`, used to perform constant-time comparisons and choices.
  - `rand`, used as an entropy source.

Testing was performed through source code review over the course of 10 person-days.

Key Findings

The assessment revealed a number of information and low-severity issues related to cryptographic defense-in-depth, best practices, and misuse resistance. These included:

- **API Callers May Pass Improperly Constructed CSPRNG to Key Generation.** Due to the design of the Schnorrkel key generation API, callers must properly construct the CSPRNG to be used for key generation. This may lead to mis-use, since there are a broad variety of ways to improperly construct a CSPRNG. See finding NCC-1905_Web3Foundation_Schnorrkel-001 on page 5.
- **Possibly Unreliable Clearing of Secrets in Memory.** Automatic clearing of secret values in RAM before deallocation relies on an external crate that is not robust against future compiler versions or usage contexts, for which the clearing may silently fail to be applied. See finding NCC-1905_Web3Foundation_Schnorrkel-006 on page 8.
- **Signatures are not Resilient to Collisions on the Hash Function.** Due to the ordering of the elements hashed to compute `r`, the signature scheme is not robust to collision attacks against the hash function. This is a trade-off between security against future attacks on the hash function, and general usability of the library, especially for streamed contents. See finding NCC-1905_Web3Foundation_Schnorrkel-003 on page 11.

Strategic Recommendations

- **Consider fully specifying the cryptographic operations.** In particular, all hashing is performed through the `Transcript` abstraction, in which every element is injected with a label; in order to allow third-party interoperable implementations of the same operations, the exact order of injection of elements, and their exact labels, should be documented. A full specification would also allow analysis of the cryptographic protocol independently of the exact implementation.

[1] https://github.com/w3f/schnorrkel/tree/849686ada82063a1e3e629c3128eda6d9564c4bc

Dashboard

Target Metadata

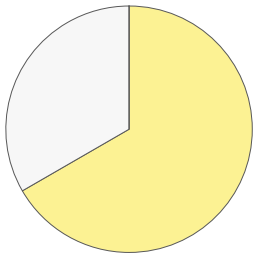
Name	Schnorrkel
Type	Library
Platforms	Rust

Engagement Data

Type	Source Code Review
Method	Code-assisted
Dates	2019-07-15 to 2019-07-23
Consultants	2
Level of effort	10 person-days

Finding Breakdown

Critical Risk issues	0
High Risk issues	0
Medium Risk issues	0
Low Risk issues	4
Informational issues	2
Total issues	**6**



Category Breakdown

Cryptography	6	

Key

Critical  High  Medium  Low  Informational 

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 13.

Title	ID	Risk
API Callers May Pass Improperly Constructed CSPRNG to Key Generation	001	Low
Some Internal Functions Are Needlessly Public	004	Low
Incomplete Or Misleading Documentation	005	Low
Possibly Unreliable Clearing of Secrets in Memory	006	Low
Inconsistent Transcript Labels	002	Informational
Signatures are not Resilient to Collisions on the Hash Function	003	Informational

Finding Details

Finding	**API Callers May Pass Improperly Constructed CSPRNG to Key Generation**
Risk	Low Impact: Low, Exploitability: None
Identifier	NCC-1905_Web3Foundation_Schnorrkel-001
Category	Cryptography
Location	`keys.rs`
Impact	Users may incorrectly configure and pass objects which implement the `rand::Rng` trait and contain the `CryptoRng` marker, but are not actually cryptographically secure pseudo random number generators. This may undermine key generation by creating predictable keys.
Description	The implementations for generating cryptographic key material, `KeyPair.generate`, `SecretKey.generate`, `MiniSecretKey.generate`, take as an argument an object which implements the `rand::Rng` trait and carries the `CryptoRng` marker:

```
// ...snip... MiniSecretKey
pub fn generate<R>(mut csprng: R) -> MiniSecretKey
where R: CryptoRng + Rng,
{
    let mut sk: MiniSecretKey = MiniSecretKey([0u8; 32]);
    csprng.fill_bytes(&mut sk.0);
    sk
}
```

```
// ...snip... SecretKey
pub fn generate<R>(mut csprng: R) -> SecretKey
where R: CryptoRng + Rng,
{
    let mut key: [u8; 64] = [0u8; 64];
    csprng.fill_bytes(&mut key);
    let mut nonce: [u8; 32] = [0u8; 32];
    csprng.fill_bytes(&mut nonce);
    SecretKey { key: Scalar::from_bytes_mod_order_wide(&key), nonce }
}
```

This is flawed from a misuse resistance perspective: a caller may choose to implement the `rand::Rng` trait with their own behavior, and there are a broad variety of ways to improperly construct a CSPRNG.

Recommendation	Consider changing the API for key generation such that it makes the decision of which RNG to use for the caller, and defaults to the OS PRNG. A useful design might be to pass a `seed` parameter to the function. If `seed` is empty, use the default CSPRNG (`rand::thread_rng`). If a seed is provided, initialize a `rand_chacha::ChaChaRng` using the provided seed. If no seed is provided and no default CSPRNG is available, raise an error.

Finding **Some Internal Functions Are Needlessly Public**

Risk Low Impact: Low, Exploitability: None

Identifier NCC-1905_Web3Foundation_Schnorrkel-004

Category Cryptography

Location `scalars.rs`

Impact Two internal functions are public, not documented, and behave in a confusing way for potential callers.

Description `scalars.rs` defines two public functions:

```
pub fn divide_scalar_bytes_by_cofactor(scalar: &mut [u8; 32]) {
    /* ... */
}

pub fn multiply_scalar_bytes_by_cofactor(scalar: &mut [u8; 32]) {
    /* ... */
}
```

Since they are public, they are potentially callable by library users. They do not have explicit documentation. They, respectively, divide or multiply the provided scalar value (encoded in little-endian over 32 bytes) by the *cofactor*, which is the integer 8. The cofactor is the quotient of the complete curve order by the order of the sub-group in which cryptographic operations are conventionally made (the Ristretto group has the same order as that sub-group). Due to the use of the term "cofactor", library users might infer that the functions multiply and divide by 8 modulo the curve order, but this is not the case; in fact, the function performs the multiplication or division over plain integers. In practice, division and multiplication by the cofactor are left and right shifts by 3 bits, respectively. Extra bits are silently dropped:

- `divide_scalar_bytes_by_cofactor()` really computes $\lfloor x/8 \rfloor$ for an input scalar value x.
- `multiply_scalar_bytes_by_cofactor()` really computes $8x$ mod 2^{256} for an input scalar value x.

These exact behaviors cannot be reliably inferred from the function names only, making the functions hardly usable by external users.

Recommendation To avoid usage confusion, these internal functions should not be made public. Declaring them `pub(crate)` would prevent usage from outside of the crate.

Finding	**Incomplete Or Misleading Documentation**
Risk	Low Impact: Low, Exploitability: None
Identifier	NCC-1905_Web3Foundation_Schnorrkel-005
Category	Cryptography

Location	`keys.rs:613`
	`points.rs:96`
	`context.rs:214`
	`cert.rs:83`
	`cert.rs:126`

Impact	The documentation of some public functions is incomplete or misleading, possibly inducing misuse by library users.

Description

In `keys.rs:613` and `points.rs:96`, the documentations of `PublicKey.from_bytes()` and `RistrettoBoth.from_bytes()`, respectively, contain an explicit warning:

```
    /// # Warning
    ///
    /// The caller is responsible for ensuring that the bytes passed into this
    /// method actually represent a `curve25519_dalek::ristretto::CompressedRist
➜   retto`
    /// and that said compressed point is actually a point on the curve.
```

This is not actually true: the implementation calls `RistrettoBoth.from_compressed()`, which performs decompression by calling the `CompressedRistretto.decompress()` function from `curve25519-dalek`, which dutifully validates that the incoming point is the canonical encoding of a Ristretto element.[2] The documentation above may induce library users to perform extra checks that are not needed, and possibly do so in insecure ways.

In `context.rs:214`, the `SigningContext.xof()` function is documented as follows:

```
    /// Initalize an owned signing transcript on a message provided as a hash fu
➜   nction with extensible output
    #[inline(always)]
    pub fn xof<D: ExtendableOutput>(&self, h: D) -> Transcript {
```

The documentation does not state how the cryptographic binding is done between the provided XOF (`h`) and the returned transcript. The implementation uses exactly 32 bytes from `h`; knowing this length is important to assess the cryptographic robustness of that binding.

In `cert.rs:83` and `cert.rs:126`, it is stated that the `Keypair.issue_ecqv_cert()` and `PublicKey.accept_ecqv_cert()` functions work over a digest called `h`:

```
    /// Aside from the issuer `PublicKey` supplied as `self`, you provide
    /// (1) a digest `h` that incorporates both the context and the
    ///     certificate requester's identity,
```

However, the actual parameter is called `t` and is a an object that implements the `Signing-Transcript` trait, not a digest value or a classic hash function.

[2] https://github.com/dalek-cryptography/curve25519-dalek/blob/1.0.3/src/ristretto.rs#L235

Finding	**Possibly Unreliable Clearing of Secrets in Memory**
Risk	Low Impact: Low, Exploitability: None
Identifier	NCC-1905_Web3Foundation_Schnorrkel-006
Category	Cryptography
Location	`clear_on_drop` crate
Impact	If used in a future Rust compiler version, the clearing of secret values from memory might silently fail, leading to secret values lingering in RAM for longer than expected.
Description	When Schnorrkel is done with performing computations with secret values held in temporary arrays, these arrays are explicitly cleared so that the corresponding bytes no longer contain any trace of these values. This is considered a best cryptographic practice, since nominally unallocated area may still contain copies of sensitive data which could be gathered by successful software or hardware exploits in other parts of the application that uses the Schnorrkel library. Automatic clearing of secrets is a defense-in-depth mechanism that reduces the temporal window for action of such exploits.

Automatic clearing is difficult to implement: from the point of view of the compiler, temporary arrays that are about to be released are no longer useful, and any operation that only impacts the contents of such arrays is prone to be omitted from the compiled output, since it has no observable consequence in the abstract machine model that the compiler is supposed to implement. Rust does not provide a guaranteed way to achieve clearing of temporary values, but it includes `std::ptr::write_volatile()`,[3] which is close:

> Rust does not currently have a rigorously and formally defined memory model, so the precise semantics of what "volatile" means here is subject to change over time. That being said, the semantics will almost always end up pretty similar to C11's definition of volatile.

Since `write_volatile()` is part of the Rust standard library, chances are that any change of the compiler that would make `write_volatile()` semantics substantially diverge from that of C's `volatile` specifier, would be noticed, and a special case added for `write_volatile()`. Despite the explicit warning about a lack of a formally defined memory model, this is the most future-robust way to enforce clearing of soon-to-be-released temporary arrays.

Schnorrkel, however, does not use `write_volatile()`. Instead, it relies on the external crate `clear_on_drop`.[4] `clear_on_drop` does not use `write_volatile()`, mostly because of perceived performance issues.[5] We may note that no actual benchmark was performed; in the case of Schnorrkel, the cost of writing 32 bytes as 32 individual write operations will likely be negligible with regards to the cost of any operation on elliptic curve points.

`clear_on_drop` will use three distinct mechanisms for automatic clearing, depending on the abilities of the compilation target:

- If the Rust compiler has the "nightly" features, then a basic clearing is done, followed by an empty inline assembly block with the target array as explicit dependency. This relies on the compiler not parsing the assembly code itself, and thus using the stated dependencies; since the compiler cannot know what the array is used for, it cannot assume that the cleared

[3] https://doc.rust-lang.org/std/ptr/fn.write_volatile.html
[4] https://docs.rs/clear_on_drop/0.2.3/clear_on_drop/
[5] https://github.com/cesarb/clear_on_drop/issues/2#issuecomment-272690027

contents are not accessed, and thus cannot optimize the clearing away.
- Otherwise, on the "stable" Rust, if a C compiler is available, an explicit call to a dummy function implemented in C is applied, for an effect similar to the inline assembly block.
- Otherwise, if using the "stable" Rust and no C compiler is available, then `clear_on_drop` will "attempts to confuse the optimizer through the use of atomic instructions": the pointer to the array is converted to a `usize` value which is then written with an `AtomicUsize.store()` call on a statically allocated slot.

All three mechanisms may plausibly fail in future compiler versions:

- Using inline assembly relies on the compiler not understanding the contents of that assembly code, in this case not noticing that the inline assembly block is actually empty. While this is the current state of the LLVM backend used by Rust, this is not a guaranteed feature for all future times; in C compilers, compilers that parse inline assembly have been known (e.g. Microsoft's Visual C, for 32-bit x86 targets). Moreover, the assembler itself might include an optimization pass; this was the case, for instance, of the SPARC assembler shipping with SunOS 5.x: the assembler included a peephole optimizer that was applied by default (but it could be explicitly disabled for some code chunks[6]).
- A similar case can be made about using a dummy C function, since the C compiler is likely to use the same backend as the Rust compiler, and, in particular, the same assembler. Interprocedural optimizations, e.g. as implemented by Intel's C compiler,[7] may also notice that the dummy C function does nothing, and *a posteriori* remove the call and the automatic clearing.
- The confusion induced by the use of an atomic instruction relies on the current state of the optimizer and is very fragile with regards to future developments. Moreover, on small embedded systems that are inherently uniprocessor, atomic stores are likely to be optimized into simple stores (since the hardware would guarantee atomicity for all such writes) and thus not so confusing for the optimizer.

The main risk here is that `clear_on_drop` relies on assumptions that may become wrong in some contexts and/or with future compiler versions, and since this is an external crate, there is no organizational mechanism that may ensure that `clear_on_drop` is promptly updated to account for such new situations. That crate is a one-developer project with low activity and questionable maintenance reactivity, e.g. the last commit was in 2017, and there are issues that have been opened more than a year ago and did not meet any action, comment or even acknowledgement.

Recommendation

Replace the invocations of `clear_on_drop` with explicit `std::ptr::write_volatile()` calls, for better robustness to future compiler versions and library usage contexts.

[6] https://docs.oracle.com/cd/E19963-01/pdf/821-1607.pdf
[7] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-interprocedural-optimization-ipo-options

Finding **Inconsistent Transcript Labels**

Risk Informational Impact: None, Exploitability: None

Identifier NCC-1905_Web3Foundation_Schnorrkel-002

Category Cryptography

Location `musig.rs:182`
`musig.rs:497`

Impact There is no practical impact on the security of the Schnorrkel crate.

Description Throughout the Schnorrkel crate, labels are used to associate metadata with commitments and challenges. The MuSig protocol specification calls for three domain separated cryptographic hashing functions, H_{agg} for computing aggregate keys H_{com} for commitments, and H_{sig} for computing Schnorr signatures (that is, computing $c = H(X, R, m)$).

When computing commitments ($H_{com}(g^r)$), where r is a random scalar in $\mathbb{Z}_p$, Schnorrkel uses the following implementation:

```
    fn for_R(R: &CompressedRistretto) -> Commitment {
        let mut t = Transcript::new(b"MuSig-commitment");
        t.commit_point(b"no\x00",R);
        let mut commit = [0u8; COMMITMENT_SIZE];
        t.challenge_bytes(b"sign\x00",&mut commit[..]);
        Commitment(commit)
    }
```

The label used here is `sign`, which is the same label as used when computing $c = H_{sig}(X, R, m)$ in the cosignature stage:

```
    pub fn cosign_stage(mut self) -> MuSig<T,CosignStage> {
        // ... snip ...
        let a_me = compute_weighting(t0, &self.stage.keypair.borrow().public);
        let c = self.t.challenge_scalar(b"sign\x00");  // context, message, A/pu
➜   blic_key, R=rG // NOTE
        let s_me = &(&c * &a_me * &self.stage.keypair.borrow().secret.key) + &se
➜   lf.stage.r_me;
        // ... snip ...
    }
```

This does not violate the protocol since each transcript will be properly domain separated by their overall label. However, it represents an inconsistency in the choice of labels used for the challenges.

One other inconsistency noted was the use of the null terminator `\x00` in some labels. Since merlin prepends proper separators (the label, followed by the data length), this null terminator is not strictly necessary to ensure proper framing.

Recommendation To ensure that H_{com} and H_{sig} always remain domain separated and to provide correct transcript metadata, each should have distinct labels. Consider updating all labels to remove the unnecessary null terminator.

Finding	**Signatures are not Resilient to Collisions on the Hash Function**
Risk	Informational Impact: Low, Exploitability: None
Identifier	NCC-1905_Web3Foundation_Schnorrkel-003
Category	Cryptography
Location	`sign.rs`, function `SecretKey.sign()`
Impact	The current Schnorrkel signature implementation allows for easier processing of streamed data, but is not robust against potential collision attacks on the underlying hash function.

Description

The EdDSA signature generation algorithm[8] can be described as follows:

- The curve subgroup has prime order q, and a conventional generator is point B.
- The private key is (x, h) where x is a scalar (integer modulo q) and h is a secret seed value. Public key is $A = xB$.
- From the message m, compute: $r = H(h \parallel m) \bmod q$
- Compute: $R = rB$
- Compute: $k = H(R \parallel A \parallel m) \bmod q$
- Compute: $s = kx + r \bmod q$
- The signature is (R, s)

The Schnorr signature implemented by Schnorrkel follows the same design, with three main changes:

- The curve subgroup is replaced with the Ristretto group.
- All hashing is done through the `SigningTranscript` abstraction, which adds explicit frames and labels, to avoid any ambiguousness when processing variable-length elements, and to tie the process to the signature generation.
- The ordering of elements injected into the hash function is different; in Schnorrkel, the following is performed:
  - Per-signature secret is computed as: $r = H(m \parallel A \parallel h) \bmod q$ (the value h is called "nonce" in Schnorrkel)
  - The hashed message itself is: $k = H(m \parallel A \parallel R) \bmod q$

This last change is the subject of this finding. In EdDSA the seed or curve points are injected in the hash function *before* the message as part of an explicit defense-in-depth against possible collision attacks on the hash function such as that presented in following hypothetical situation:

- The hash function works over its input with a single pass, maintaining an internal finite state. This is the case of all Merkle-Damgård functions such as MD5, SHA-1, and the SHA-2 family; this also applies to sponge constructions like Keccak/SHA-3.
- An efficient attack is found, that creates collisions on the internal state.
- The attacker can find colliding pairs, in which one value is an innocuous-looking message for which the attacker can obtain a signature from the signer; the signature value could then be transported onto the other message of the colliding pair.

That kind of attack has been demonstrated[9] in the context of an X.509 certification authority using the RSA signature algorithm with MD5 as underlying hash function. EdDSA defends against such attacks by hashing the secret seed, or the per-signature point R, before the

[8] https://tools.ietf.org/html/rfc8032
[9] https://www.win.tue.nl/hashclash/rogue-ca/

message: these values are not predictable by the attacker, preventing use of precomputed collisions. This is called *collision resilience* in the EdDSA specification[10]; EdDSA provides this property as long as the "PureEdDSA" variant is used. In essence, if PureEdDSA were to be used with MD5, it would still be cryptographically unbroken, despite the extreme weakness of MD5 with regards to collisions.

By injecting the message first in the hash function calls, Schnorrkel forfeits collision resilience. The following points should be noted:

- Collision resilience is a defense-in-depth mechanism, meant to ensure survival of existing signatures in some cases of catastrophic breaks on the underlying hash function. The SHA-3 competition was enacted because of the known attacks on MD5 and SHA-1, and the main result of the decade of accumulated research is that the kind of differential paths that lead to these attacks are unlikely to apply to Keccak, the competition winner (and, arguably, the SHA-2 functions appear to be also adequately robust in that respect). Moreover, Keccak, being a sponge function, has a larger internal state than Merkle-Damgård functions (200 bytes for Keccak, vs 32 bytes for SHA-256 or 64 bytes for SHA-512); this should heuristically make internal state collisions comparatively harder to achieve. Keccak (which powers STROBE) is at the core of the default `Transcript` implementation provided by the `merlin` crate.
- Injecting the message first makes support of streamed processing easier. If a large message must be signed or verified, and the data is obtained by chunks, then the current Schnorrkel design allows for starting to process the data chunks before having obtained the private key (for signature generation) or the public key and signature value (for signature verification). Compatibility with streamed processing is an important feature when using RAM-constrained embedded systems. In that sense, there is a trade-off between collision resilience and ease of streamed processing.

Recommendation

Given the internal use of Keccak for hashing, potential collisions are only a very remote threat, and allowing easier streamed data processing can be argued to be a more important property than collision resilience. However, since Schnorrkel departs from the EdDSA design, this choice should be explicitly documented.

Alternatively, if collision resilience must be recaptured, then the current API will have to change: the injection of the seed, or public key and signature point, will need to occur before injecting the message data itself.

[10] https://tools.ietf.org/html/rfc8032#section-4

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Project Contacts

The team from NCC Group has the following primary members:

- Ava Howell — Consultant
  ava.howell@nccgroup.com
- Thomas Pornin — Consultant
  thomas.pornin@nccgroup.com
- Javed Samuel — Account manager
  javed.samuel@nccgroup.com

The team from Web 3.0 Technologies Stiftung has the following primary members:

- Jeff Burdges — Web 3.0 Technologies Stiftung
  jeff@web3.foundation
- Peter Czaban — Web 3.0 Technologies Stiftung
  peter@web3.foundation