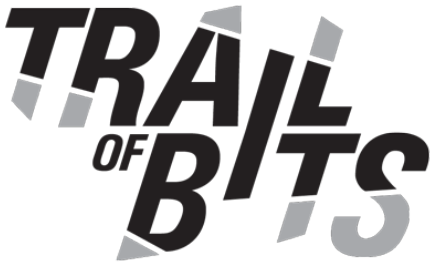

ChainSafe go-schnorrkel

Security Assessment

August 25, 2021

Prepared For:
Elizabeth Binks | *ChainSafe*
elizabeth@chainsafe.io

Prepared By:
Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Opal Wright | *Trail of Bits*
opal.wright@trailofbits.com

Changelog:

August 25, 2021:	Initial report draft
September 10, 2021:	Added Appendix D: Fix Log

Executive Summary

From August 9 to August 20, 2021, Trail of Bits assessed the security of ChainSafe's implementation of the Schnorr signature algorithm over the `ristretto25519` curve. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commit hash `76165a18` of the `go-schnorrkel` repository.

We spent the first week of the assessment obtaining and building the source code and ensuring that all unit tests pass. To better understand the codebase, we also examined the existing Rust implementation of the Schnorr signature algorithm and the implementation's related resources. From there, we ran a series of static analysis tools, such as `gosec` and `errcheck`, and triaged their results. We then began manually reviewing the codebase, verifying that secrets are handled safely and that the implementation conforms to the protocol. Lastly, we began using `go-fuzz` to integrate fuzz testing into the codebase.

In the second week, we completed our manual review of the codebase. We also ran additional static analysis tools, such as `Semgrep`. Finally, we implemented differential testing between `go-schnorrkel` and the Rust implementation of the protocol to ensure that they are compatible. This testing targeted the traditional Schnorr signature verification and batch verification schemes.

Our assessment resulted in five findings ranging from high to informational severity. The high-severity finding relates to a vulnerability in the verifiable random function (VRF) and signature schemes: the verification functions in these schemes do not check whether the public key is the point at infinity. The remaining findings pertain to issues like insufficient data validation and challenge generation. In addition to these findings, we developed code quality recommendations, which are detailed in Appendix C.

Overall, we found the codebase to be well organized and documented. The codebase contains unit tests for the most critical functions and uses test vectors for both the BIP39 and VRF implementations to give better assurance of their correctness. However, data validation throughout the codebase could be improved, as we identified instances in which unexpected inputs result in `panics`.

Going forward, in addition to addressing the findings in this report, we encourage ChainSafe to expand on our differential testing and integrate it into the test suite. This will help ensure that the repository remains compatible with the Rust implementation even as changes are made. The Rust implementation uses the `zeroize` crate to protect secrets. Go does not appear to offer an analogous library; however, we encourage ChainSafe to integrate such a solution if ever available.

Update: On September 10, 2021, Trail of Bits reviewed fixes implemented for the issues presented in this report. See a detailed review of the current status of each issue in Appendix D.

Project Dashboard

Application Summary

Name	`go-schnorrkel`
Version	Commit: `76165a18`
Type	Go
Platform	Various

Engagement Summary

Dates	August 9–August 20, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	1	■
Total Informational-Severity Issues	2	■■
Total Undetermined-Severity Issues	0	
Total	5	

Category Breakdown

Cryptography	3	■■■
Data Validation	1	■
Denial of Service	1	■
Total	5	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Weak.** Data validation throughout the codebase could be improved, as multiple functions have implicit assumptions about user-provided data. Better access controls and data validation would have prevented many of the issues uncovered in our audit.
Arithmetic	**Satisfactory.** Most of the library's arithmetic is handled by an external Ristretto library, which has strong side-channel resistance and constant-time guarantees.
Function Composition	**Strong.** The codebase is organized neatly, with each file implementing a particular primitive. The functions have clear, narrow purposes, and critical functions can be easily extracted for testing.
Key Management	**Satisfactory.** Secrets are handled securely throughout the system. The reference Rust implementation uses the `zeroize` crate to zero out secrets after they are no longer needed. An analogous solution does not appear to exist for Go, but ChainSafe can consider implementing such a solution if it becomes available in the future.
Specification	**Satisfactory.** The codebase contains a good amount of inline documentation and pointers to a reference implementation, which includes references to all of the relevant protocol descriptions.
Testing & Verification	**Moderate.** The codebase contains unit tests for most critical functions and includes available test vectors. However, the codebase lacks dynamic testing and negative testing, which could have prevented some of the issues uncovered in our audit.

Engagement Goals and Coverage

The engagement was scoped to provide a security assessment of ChainSafe's implementation of the Schnorr signature scheme over the `ristretto25519` curve.

Specifically, we sought to answer the following questions:

- Is the library susceptible to any known cryptographic attacks?
- Is the API designed to be safe and resistant to misuse?
- Are cryptographic secrets handled securely?
- Does the codebase contain sufficient data validation?
- Are there any vulnerabilities that can be discovered with static analysis or fuzzing?
- Is the library compatible with the Rust implementation of the Schnorr signature scheme?

To answer these questions, we performed a manual review of the `go-schnorrkel` repository. To test the library's compatibility with the Rust implementation, we implemented differential testing, which targeted the Schnorr signature verification and batch verification schemes. To conduct the differential testing, we generated signatures using the Rust implementation, wrote them to standard output, and then wrote a series of unit tests in Go that parsed the signatures from standard output and passed them to the verification function in `go-schnorrkel`. Lastly, we ran the following static analysis tools and triaged their results: `gosec`, `errcheck`, `ineffassign`, `staticcheck`, and `Semgrep`.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Add a check to the `MakeBytes` function so that it returns an error if the `size` value is less than or equal to zero or if it is larger than a predetermined value.** TOB-CNSF-001

❑ **Add a check to ensure that each generated random scalar value is non-zero.** TOB-CNSF-002

❑ **Consider including the base point in the calculation of the challenge scalar.** TOB-CNSF-003

❑ **Add `nil` pointer checks to all affected functions.** TOB-CNSF-004

❑ **Add checks in the signature and VRF verification functions to reject all inputs when the public key is the point at infinity.** TOB-CNSF-005

Long term

❑ **Consult the maintainers of the existing Rust implementation to determine whether the base point should be used in the challenge scalar.** TOB-CNSF-003

❑ **Ensure that all pointers are checked before dereferencing them.** TOB-CNSF-004

❑ **Add tests to ensure that the point at infinity is always invalidated in the signature and VRF verification functions.** TOB-CNSF-005

Findings Summary

#	Title	Type	Severity
1	Insufficient input validation in the VRF MakeBytes function	Data Validation	Low
2	Random scalars used in batch verification can be zero	Cryptography	Informational
3	Base point of the Ristretto curve is not used in the VRF challenge scalar value calculation	Cryptography	Informational
4	Functions missing nil pointer checks	Denial of Service	Medium
5	VRF and signature schemes do not check for the point at infinity	Cryptography	High

1. Insufficient input validation in the VRF MakeBytes function

Severity: Low
Type: Data Validation
Target: `vrf.go`

Difficulty: High
Finding ID: TOB-CNSF-001

Description
The `go-schnorrkel` codebase implements a VRF that outputs random bytes. As shown in figure 1.1, the `MakeBytes` function takes `size` and `context` input and returns a raw-byte output from the VRF.

```
// MakeBytes returns raw bytes output from the VRF
// It returns a byte slice of the given size
// see https://github.com/w3f/schnorrkel/blob/master/src/vrf.rs#L334
func (io *VrfInOut) MakeBytes(size int, context []byte) []byte {
        t := merlin.NewTranscript("VRFResult")
        t.AppendMessage([]byte(""), context)
        io.commit(t)
        return t.ExtractBytes([]byte(""), size)
}
```

Figure 1.1: The `MakeBytes` *function does not validate* `size` *input.*

However, there are currently no restrictions on the values that the `size` input can be. This could lead to unexpected or potentially dangerous behavior. For instance, if `size` is the value zero, the program will `panic`. In addition, a large `size` value could cause a denial of service.

Exploit Scenario
A user, Alice, includes `go-schnorrkel` in her application. An attacker, Eve, causes Alice to call the `MakeBytes` function with a `size` value of either zero or a very large number. This causes a denial of service or an unexpected `panic` that disrupts Alice's application.

Recommendations
Short term, add a check to the `MakeBytes` function so that it returns an error if the `size` value is less than or equal to zero or if it is larger than a predetermined value.

2. Random scalars used in batch verification can be zero

Severity: Informational
Type: Cryptography
Target: `batch.go`

Difficulty: High
Finding ID: TOB-CNSF-002

Description
The `go-schnorrkel` codebase implements batch verification of Schnorr signatures. To perform batch verification, random scalar values are generated for each signature. The supplied signatures and public keys are then multiplied by the generated scalar values. These values should be non-zero and unpredictable for the batch verification process to be secure. If any of these values were zero, the batch verification process would ignore the corresponding signatures. In this situation, forged signatures could be verified as valid.

As shown in figure 2.1, random scalar values are generated by the `NewRandomScalar` function. However, it is possible for this function to return zero for the random scalar.

```
// VerifyBatch batch verifies the given signatures
func VerifyBatch(transcripts []*merlin.Transcript, signatures []*Signature, pubkeys
[]*PublicKey) (bool, error) {
	if len(transcripts) != len(signatures) || len(signatures) != len(pubkeys) ||
len(pubkeys) != len(transcripts) {
		return false, errors.New("the number of transcripts, signatures, and public
keys must be equal")
	}

	var err error
	zero := r255.NewElement().Zero()
	zs := make([]*r255.Scalar, len(transcripts))
	for i := range zs {
		zs[i], err = NewRandomScalar()
		if err != nil {
			return false, err
		}
	}
```

Figure 2.1: The `VerifyBatch` *function generates random scalars by calling* `NewRandomScalar`.

Exploit Scenario
Another flaw in the random number generator causes `NewRandomScalar` to generate zero more often than expected. An attacker notices this flaw and submits invalid proofs, knowing that they will be verified as valid.

Recommendations
Short term, add a check to ensure that each generated random scalar value is non-zero.

3. Base point of the Ristretto curve is not used in the VRF challenge scalar value calculation

Severity: Informational
Type: Cryptography
Target: `vrf.go`

Difficulty: High
Finding ID: TOB-CNSF-003

Description
The `go-schnorrkel` codebase implements a VRF, which generates random values and a proof that it generated these values correctly. To generate this proof, the VRF computes a challenge scalar value. According to the specification, this challenge value should be calculated by hashing six different values. However, the implementation of the VRF in `go-schnorrkel` uses only five of these values; it is missing the base point of the Ristretto curve.

In general, it is best practice to follow a specification as closely as possible. For proof systems in particular, challenge scalar values need to be calculated very carefully. Omitting certain values from this calculation could result in insecure proofs.

The reference Rust implementation does not use this base point in the challenge-generation process. The ChainSafe team consulted the maintainers of the Rust implementation, and the maintainers indicated that omitting the base point of the Ristretto curve will not be an issue because the protocol and the base point will not change.

Exploit Scenario
Alice uses the VRF implemented in `go-schnorrkel`, but she uses a different base point. Because the base point is not included in the challenge scalar, an attacker is able to forge proofs.

Recommendations
Short term, consider including the base point in the calculation of the challenge scalar.

Long term, consult the maintainers of the existing Rust implementation to determine whether the base point should be used in the challenge scalar.

References
How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios

4. Functions missing nil pointer checks

Severity: Medium
Type: Denial of Service
Target: `batch.go`

Difficulty: Low
Finding ID: TOB-CNSF-004

Description
Several functions fail to check for `nil` pointers before dereferencing them. If a `nil` pointer is provided to these functions, a runtime `panic` will occur. The functions should check for `nil` pointers and return an error in the event that a nil pointer is provided.

The following functions contain this vulnerability and are publicly exposed:
- `VerifyBatch`
- `VrfSign`
- `DeriveScalarAndChaincode`
- `DeriveKey`
- `AttachInput`
- `Verify`

The following functions contain this vulnerability but are not publicly exposed:
- `challengeScalar`
- `commit`

Exploit Scenario
An attacker is able to manipulate a program so that it passes a `nil` pointer to the `go-schnorrkel` library, leading to a denial of service.

Recommendations
Short term, add `nil` pointer checks to all affected functions.

Long term, ensure that all pointers are checked before dereferencing them.

References
- Go Language Specification, the “Address operators” section

5. VRF and signature schemes do not check for the point at infinity

Severity: High
Type: Cryptography
Target: `keys.go`, `sign.go`, `vrf.go`

Difficulty: Low
Finding ID: TOB-CNSF-005

Description
The `go-schnorrkel` codebase uses the Ristretto curve for both the VRF and signature schemes. Therefore, for both schemes, the secret keys are scalar values, and the public keys are the Ristretto base point scalar-multiplied by the secret keys. If the secret key is the zero scalar, then the public key will be the point at infinity.

In general, allowing the validity of signatures associated with public keys that are the point at infinity is not best practice. However, the use of these keys for the VRF scheme makes this issue even more problematic. In the implemented scheme, the output for the VRF is computed by scalar-multiplying the input value by the secret key. However, if the secret key is the zero scalar, the VRF will produce the same output every time regardless of the input value. If the VRF produces the same output every time, it is clearly not operating as a random function.

Trail of Bits wrote a test and confirmed that the VRF implementation verifies outputs generated by the point at infinity instead of throwing an error.

Exploit Scenario
A user builds an application that relies on the VRF to generate secure random values. An attacker uses the point at infinity to cause the VRF to generate the same values repeatedly, compromising the application's security.

Recommendation
Short term, add checks in the signature and VRF verification functions to reject all inputs when the public key is the point at infinity.

Long term, add tests to ensure that the point at infinity is always invalidated in the signature and VRF verification functions.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
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 protecting the privacy or integrity of 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 a 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
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.

High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

This appendix contains findings that do not have immediate or obvious security implications.

- Various magic numbers are used in the `bip39.go`, `derive.go`, and `helpers.go` files. These numbers appear to be constant string and buffer sizes and some cryptographic parameters. It is best practice to replace these numbers with global or constant variables. This makes the code more readable and easier to maintain should these values ever need to be adjusted.
- There are three "TODOs" in the `derive.go`, `keys.go`, and `vrf.go` files. These should be resolved.
- Several functions throughout the codebase follow a pattern of creating `Scalar` objects and populating them via the `FromUniformBytes` method. To reduce code duplication, this pattern could be integrated into a helper function.
- Run the `golangci-lint` tool and consider implementing GolanCI's style recommendations.

D. Fix Log

After the initial assessment, ChainSafe addressed the reported issues through pull request #139 to `go-schnorrkel`. Trail of Bits reviewed each fix to ensure that it correctly addresses the corresponding issue. The results of this review and additional details on each fix are provided below.

#	Title	Severity	Status
1	Insufficient input validation in the VRF MakeBytes function	Low	Fixed
2	Random scalars used in batch verification can be zero	Informational	Fixed
3	Base point of the Ristretto curve is not used in the VRF challenge scalar value calculation	Informational	Not fixed
4	Functions missing nil pointer checks	Medium	Partially fixed
5	VRF and signature schemes do not check for the point at infinity	High	Fixed

Detailed Fix Log

TOB-CNSF-001: Insufficient input validation in the VRF MakeBytes function
Fixed. Checks have been added to ensure that `size` is greater than zero and less than 64 bytes.

TOB-CNSF-002: Random scalars used in batch verification can be zero
Fixed. A check has been added to ensure that `NewRandomScalar` returns non-zero values.

TOB-CNSF-003: Base point of the Ristretto curve is not used in the VRF challenge scalar value calculation
Not fixed. The reference Rust implementation does not include the base point, and the developers maintaining the Rust implementation indicated that they will not adjust their codebase. Therefore, in order to ensure that `go-schnorrkel` remains compatible with this library, this issue cannot be fixed.

TOB-CNSF-004: Functions missing nil pointer checks
Partially fixed. Each of the publicly exposed functions now have `nil` pointer checks; however, the two aforementioned functions that are not publicly exposed still do not have `nil` pointer checks.

TOB-CNSF-005: VRF and signature schemes do not check for the point at infinity
Fixed. The verification functions in `sign.go` and `vrf.go` now return an error if the supplied public key is the point at infinity.