Snow

Security Assessment

Jan 23, 2024

Prepared for:
Christian Rask
AgileBits

Prepared by: **Joe Doyle and Tjaden Hess**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Joe Doyle, Consultant
joe.doyle@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 05, 2024	Pre-project kickoff call
January 16, 2024	Status update meeting #1
January 23, 2024	Delivery of report draft
January 23, 2024	Report readout meeting

Executive Summary

Engagement Overview

AgileBits engaged Trail of Bits to review the security of the open-source Snow library. Snow is a Rust language implementation of the Noise Protocol Framework, a framework for crypto protocols based on Diffie-Hellman key agreement.

A team of two consultants conducted the review from January 8 to January 22, 2024, for a total of four engineer-weeks of effort. Our testing efforts focused on manually reviewing the Snow implementation for correct use of cryptographic primitives, API design safety and compliance with the Noise protocol specification. With full access to source code and documentation, we performed static and dynamic testing of the Snow library, using automated and manual processes.

Observations and Impact

The Snow library is generally clean, well-organized and follows the protocol specification. However, we discovered several issues with the handling of improperly formatted data (findings TOB-SNOW-3, TOB-SNOW-4, TOB-SNOW-5, TOB-SNOW-7, TOB-SNOW-9) that could have been uncovered by more extensive testing, especially focused on edge cases and adversarial behavior. Although most of those are strictly informational, finding TOB-SNOW-5 can cause a panic and finding TOB-SNOW-7 allows an adversary present on the network to permanently disrupt an encrypted channel without knowing any cryptographic secret.

We also found several portions of the library which could cause problems if more features are added to Snow or if they are used in non-standard ways, including improper handling of not-currently-available handshake patterns (finding TOB-SNOW-2), methods which silently overwrite previous values of configuration variables (finding TOB-SNOW-6), and inconsistent handling of incorrectly-sized keys.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the Snow developers take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Add tests which include adversarial interference.** Finding TOB-SNOW-7, which allows a malicious network participant to disrupt an ongoing connection indefinitely, would be quickly found by testing malicious behavior such as a message replay attack. These tests should exercise error cases, and also test the library's functionality in post-error states.

- **Use code-coverage tooling to guide further testing.** There are some aspects of the Noise handshake which are currently not tested by the unit test suite. These areas can be identified and remediated through regular use of a coverage tool such as cargo-llvm-cov.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	1
Informational	8
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	3
Data Validation	6
Patching	1

Project Goals

The engagement was scoped to provide a security assessment of the Snow Noise Protocol Framework implementation. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the codebase rely on any known insecure or outdated dependencies?
- Are there any exposures to known cryptographic attacks?
- Do any logic flaws or weak cryptography exist in the system?
- Does the implementation properly follow Trevor Perrin's Noise Protocol specifications, in terms of cryptographic principles and primitives?
- Are there any additional off by one errors which could allow invalid nonce usage?
- Does the implementation follow and enforce proper state transitions, such as requiring a constructed `HandshakeState` prior to conversion to a `TransportState` or `StatelessTransportState`?

- Are there any language or platform specific vulnerabilities?
- Do the functions properly implement their respective standards?
- Are error states or failure modes properly handled?
- Are there aspects of the implementation that seem error-prone or unintuitive?
- Could the system experience a denial of service?
- Are all inputs and system parameters properly validated?
- Does the codebase conform to industry best practices?
- Are there any areas of improvement for the CICD or SDLC?

Project Targets

The engagement involved a review and testing of the following target.

Snow

Repository	https://github.com/mcginty/snow
Version	`009411d2035a77ece57b0a6873169de7693a0aa0`
Type	Rust
Platform	Native

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual review of the Noise Protocol Framework specification
- Manual review of the Snow source code and documentation
- Static linting of the Rust codebase

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Cryptographic dependencies such as `Ring`, `SodiumOxide` and `RustCrypto`

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D.1
`cargo-llvm-cov`	A tool for generating test coverage reports in Rust	Appendix D.2
Cargo Audit	A tool for checking dependencies against the RustSec advisory database	Appendix D.3

Areas of Focus

Our automated testing and verification work focused on the following system properties:

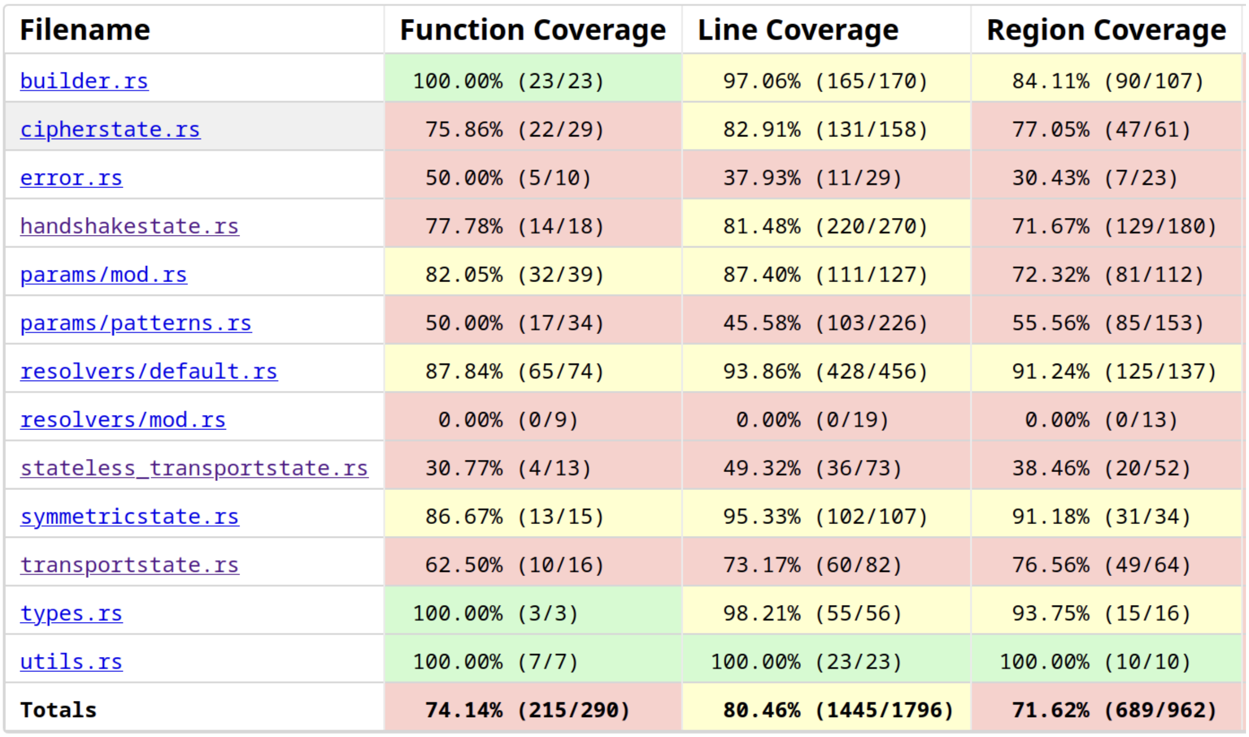
- General code quality issues and unidiomatic code patterns
- Untested code regions with coverage reports

Test Results

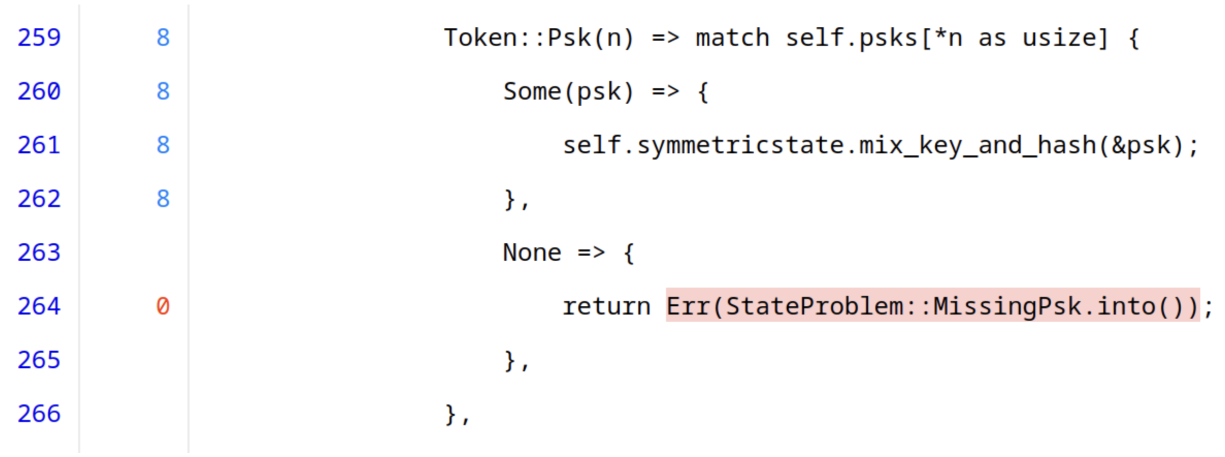
The results of this focused testing are detailed below.

cargo-llvm-cov

The coverage report illustrates several areas which are not covered by unit tests – for example, no unit tests exercise the code path in `src/handshakestate.rs` which handles using a PSK slot which has not been set.



Filename	Function Coverage	Line Coverage	Region Coverage
builder.rs	100.00% (23/23)	97.06% (165/170)	84.11% (90/107)
cipherstate.rs	75.86% (22/29)	82.91% (131/158)	77.05% (47/61)
error.rs	50.00% (5/10)	37.93% (11/29)	30.43% (7/23)
handshakestate.rs	77.78% (14/18)	81.48% (220/270)	71.67% (129/180)
params/mod.rs	82.05% (32/39)	87.40% (111/127)	72.32% (81/112)
params/patterns.rs	50.00% (17/34)	45.58% (103/226)	55.56% (85/153)
resolvers/default.rs	87.84% (65/74)	93.86% (428/456)	91.24% (125/137)
resolvers/mod.rs	0.00% (0/9)	0.00% (0/19)	0.00% (0/13)
stateless_transportstate.rs	30.77% (4/13)	49.32% (36/73)	38.46% (20/52)
symmetricstate.rs	86.67% (13/15)	95.33% (102/107)	91.18% (31/34)
transportstate.rs	62.50% (10/16)	73.17% (60/82)	76.56% (49/64)
types.rs	100.00% (3/3)	98.21% (55/56)	93.75% (15/16)
utils.rs	100.00% (7/7)	100.00% (23/23)	100.00% (10/10)
Totals	**74.14% (215/290)**	**80.46% (1445/1796)**	**71.62% (689/962)**



```
259    8            Token::Psk(n) => match self.psks[*n as usize] {
260    8                Some(psk) => {
261    8                    self.symmetricstate.mix_key_and_hash(&psk);
262    8                },
263                     None => {
264    0                    return Err(StateProblem::MissingPsk.into());
265                     },
266                 },
```

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We did not find any instances of unchecked arithmetic overflow, after checking potential instances returned by the `clippy::arithmetic-side-effects` lint.	**Strong**
Auditing	There is no logging functionality required.	**Not Applicable**
Authentication / Access Controls	There are no permissioned roles in the system.	**Not Applicable**
Complexity Management	The codebase is clear and well-organized, with meaningful names and no overly–complex functions.	**Satisfactory**
Cryptography and Key Management	We found one instance of state modification based on unauthenticated data (TOB-SNOW-7). Cryptographic keys are not zeroized after use (TOB-SNOW-8).	**Moderate**
Documentation	There is sufficient documentation and example code to use the library. Some functions need documentation describing their behavior when used outside of typical patterns (TOB-SNOW-6, TOB-SNOW-9, TOB-SNOW-10).	**Satisfactory**
Memory Safety and Error Handling	The codebase does not use unsafe Rust. There was one instance of an out-of-bounds index leading to a panic (TOB-SNOW-5).	**Satisfactory**
Testing and Verification	Unit testing covers most of the library functionality but does not hit all logic, especially those cases triggered by uncommon handshake patterns. Some fuzzing is present but is stateless and does not cover all stages of the handshake and transport.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Sodiumoxide dependency is deprecated and unmaintained	Patching	**Informational**
2	Pre-message ephemeral keys are not mixed into the encryption key	Cryptography	**Informational**
3	Received messages can exceed maximum message length	Data Validation	**Informational**
4	Noise protocol names can have extra data appended with no effect	Data Validation	**Informational**
5	Protocol builder panics on invalid pre-shared-key indices	Data Validation	**Low**
6	Builder setter methods silently overwrite existing values	Data Validation	**Informational**
7	CipherState::decrypt_ad increments nonce even after authentication failure.	Cryptography	**Medium**
8	Ephemeral keys are not cleared from memory	Cryptography	**Informational**
9	PSK modifiers can be repeated in handshake pattern names	Data Validation	**Informational**
10	Resolvers behave differently with long and short keys	Data Validation	**Informational**

Detailed Findings

1. Sodiumoxide dependency is deprecated and unmaintained	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Patching	Finding ID: TOB-SNOW-1
Target: `Cargo.toml`, `src/resolvers/libsodium.rs`	

Description

The Snow library allows users to choose between several backends implementing the cryptographic primitives required by the Noise protocol. One provided backend uses `libsodium` via the `sodiumoxide` Rust binding library, which has been deprecated and is listed by the RustSec security advisory database as unmaintained.

While no security vulnerabilities are currently known in the `sodiumoxide` library, vulnerabilities discovered in the future may not be patched in a timely manner.

Recommendations

Short term, consider deprecating the `sodiumoxide` backend or adding a warning to users that the backend is currently unmaintained.

Long term, use `cargo audit` as part of the release process to check for known vulnerabilities and deprecations in dependency libraries.

2. Pre-message ephemeral keys are not mixed into the encryption key	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-SNOW-2
Target: `src/handshakestate.rs`	

Description

When a handshake pattern uses both a pre-shared key and an ephemeral public key pre-message, the Snow implementation does not mix the ephemeral public key into the initial encryption key state, potentially allowing catastrophic nonce re-use in valid, non-standard handshake patterns. The Snow library currently does not support any handshake patterns that exhibit this behavior, but may do so in the future when the `fallback` modifier is implemented.

Mixing ephemeral key pre-messages into the encryption key is required by Noise protocol section 9.2:

> In non-PSK handshakes, the "e" token in a pre-message pattern or message pattern always results in a call to MixHash(e.public_key). In a PSK handshake, all of these calls are followed by MixKey(e.public_key). In conjunction with the validity rule in the next section, this ensures that PSK-based encryption uses encryption keys that are randomized using ephemeral public keys as nonces.

Figure 2.1: Noise protocol section 9.2

The implementation does mix ephemeral keys from message patterns into the encryption key, but does not do so for pre-message patterns.

All recommended Noise patterns which include an ephemeral key pre-message (for example `XXfallback`) also include an ephemeral key as the first component of the first initiator message, preventing the catastrophic nonce-reuse that could occur due to this implementation flaw. However, if a user designs a custom handshake pattern, such as the one in figure 2.2, which includes a static key in the first initiator message, the static key will be encrypted with an AEAD key and nonce that is constant among different runs of the protocol.

```
Examplepsk0:
  <- s
  <- e
```

```
...
-> psk, s, ss, e, es
<- ee, se
```

Figure 2.2: Valid handshake pattern triggering catastrophic nonce re-use

Exploit Scenario

A user extends the library to support a non-standard handshake pattern such as the one listed in figure 2.2. Two users initiate the protocol using the same pre-shared key. Because the pre-message keys are not mixed into the encryption key, the initiators' static keys are each encrypted with an identical key derived from the PSK.

An adversary uses the catastrophic nonce-reuse attack on AES-GCM to recover the cipher authentication key. The adversary then modifies ciphertexts to substitute their own static key, bypassing the authentication provided by the PSK.

Recommendations

Short term, use `MixKey` to incorporate pre-message ephemeral public keys into the encryption key state.

Long term, consider modifying the Noise protocol specification to more clearly specify the behavior of PSK patterns by including the conditional `MixKey` operations in section 5.3.

3. Received messages can exceed maximum message length

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SNOW-3
Target: `src/transportstate.rs, src/stateless_transportstate.rs`	

Description

The Noise protocol specification states that “All Noise messages are less than or equal to 65535 bytes in length”. The snow implementation enforces this maximum length explicitly when sending messages, but it does not enforce it when receiving messages, as shown below in figures 3.1 and 3.2.

```
pub fn write_message(&mut self, payload: &[u8], message: &mut [u8]) -> Result<usize, Error>
{
    if !self.initiator && self.pattern.is_oneway() {
        return Err(StateProblem::OneWay.into());
    } else if payload.len() + TAGLEN > MAXMSGLEN || payload.len() + TAGLEN > message.len() {
        return Err(Error::Input);
    }

    let cipher =
        if self.initiator { &mut self.cipherstates.0 } else { &mut self.cipherstates.1 };
    cipher.encrypt(payload, message)
}
```

Figure 3.1: The maximum length of a message is enforced via the highlighted condition when sending messages (`snow/src/transportstate.rs#56–66`)

```
pub fn read_message(&mut self, message: &[u8], payload: &mut [u8]) -> Result<usize,
Error> {
    if self.initiator && self.pattern.is_oneway() {
        return Err(StateProblem::OneWay.into());
    }
    let cipher =
        if self.initiator { &mut self.cipherstates.1 } else { &mut self.cipherstates.0 };

    cipher.decrypt(message, payload)
}
```

Figure 3.2: Message length is not checked when receiving messages (`snow/src/transportstate.rs#78–86`)

While this does not cause any immediate security issues, allowing long messages to be received could create a potential denial-of-service attack or allow a partition-oracle attack, depending on the exact threat model and deployed configuration.

Recommendations

Short term, add validation to `read_message` to reject messages that are longer than 65535 bytes.

Long term, ensure that all parsing functions strictly accept only data formatted in accordance with the Noise protocol specification.

4. Noise protocol names can have extra data appended with no effect	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SNOW-4
Target: `src/params/mod.rs`	

Description

Noise protocol parameters are parsed from the protocol name by splitting the protocol name into underscore-delimited substrings, then parsing each substring as the corresponding parameter type, as shown below in figure 4.1.

```
#[cfg(not(feature = "hfs"))]
fn from_str(s: &str) -> Result<Self, Self::Err> {
    let mut split = s.split('_');
    Ok(NoiseParams::new(
        s.to_owned(),
        split.next().ok_or(PatternProblem::TooFewParameters)?.parse()?,
        split.next().ok_or(PatternProblem::TooFewParameters)?.parse()?,
        split.next().ok_or(PatternProblem::TooFewParameters)?.parse()?,
        split.next().ok_or(PatternProblem::TooFewParameters)?.parse()?,
        split.next().ok_or(PatternProblem::TooFewParameters)?.parse()?,
    ))
}
```

Figure 4.1: Parsing a protocol name to initialize a `NoiseParams` object (`snow/src/params/mod.rs#199–210`)

However, no validation is done to the string after these parameters have been parsed, so extra text appended to a protocol name after an underscore yields the same protocol, but with a different name, and therefore a different initial chaining key value. For example, the protocol names "`Noise_XX_25519_AESGCM_SHA256`", "`Noise_XX_25519_AESGCM_SHA256_`", and "`Noise_XX_25519_AESGCM_SHA256_OtherThingsThatShouldNotBeHere`" all correspond to using the `XX` handshake pattern, the `X25519` DH function, the `AES-GCM` cipher, and the `SHA256` hash function; but each will have a different initial chaining key.

In addition to protocol names which lead to different initial chaining keys, there are also some protocol names which are non-equal but lead to identical initial chaining keys. The initial chaining key is initialized by loading the protocol name into a buffer that is pre-filled with zero bytes, as shown in figure 4.2. By appending an underscore followed by null characters, one can create protocols with non-equal names but are otherwise identical, including in the initial chaining key value. For example, building a noise protocol instance with the strings "`Noise_XX_25519_AESGCM_SHA256_\0\0\0`" and

“Noise_XX_25519_AESGCM_SHA256_” will yield exactly identical protocols. Note that the underscore is necessary, since otherwise the builder will parse the trailing null characters as part of the hash function name.

```
pub fn new(cipherstate: CipherState, hasher: Box<dyn Hash>) -> SymmetricState {
    SymmetricState { cipherstate, hasher, inner: SymmetricStateData::default() }
}

pub fn initialize(&mut self, handshake_name: &str) {
    if handshake_name.len() <= self.hasher.hash_len() {
        copy_slices!(handshake_name.as_bytes(), self.inner.h);
    } else {
        self.hasher.reset();
        self.hasher.input(handshake_name.as_bytes());
        self.hasher.result(&mut self.inner.h);
    }
    copy_slices!(self.inner.h, &mut self.inner.ck);
    self.inner.has_key = false;
}
```

Figure 4.2: Protocol names shorter than the length of the hash (typically 32 bytes) are copied into a default-initialized buffer which becomes the initial chaining key. (`src/symmetricstate.rs#32–46`)

Any protocol name with an underscore after the hash function name is invalid according to the Noise protocol specification (section 8.1), but accepting them does not appear to allow an attack. However, this issue could cause errors if a protocol built on top of snow assumes that all noise protocols with different names will be incompatible.

Recommendations

Short term, add validation to reject protocol names with any extraneous data after the hash function name.

Long term, ensure that all parsing functions strictly accept only data formatted in accordance with the Noise protocol specification.

5. Protocol builder panics on invalid pre-shared-key indices

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SNOW-5
Target: `src/params/patterns.rs`	

Description

Handshake patterns are selected by a pattern name followed by a potentially-empty list of modifiers delimited by the "+" character. The modifier can be "fallback" (which is not currently supported by snow) or `pskN`, where `N` is an 8-bit integer identifying the position in the handshake pattern where a pre-shared key should be mixed into the state. The snow library implements this by separately parsing the pattern name and the modifiers, then calling the `apply_psk_modifier` function, shown below in figure 5.1, on the `Pattern` object constructed based on the chosen pattern name.

```
fn apply_psk_modifier(patterns: &mut Patterns, n: u8) {
    match n {
        0 => {
            patterns.2[0].insert(0, Token::Psk(n));
        },
        _ => {
            let i = (n as usize) - 1;
            patterns.2[i].push(Token::Psk(n));
        },
    }
}
```

Figure 5.1: The unchecked array access when applying the pre-shared-key modifier (`snow/src/params/patterns.rs#523–533`)

However, the indexing in this function is unchecked, and can trigger a panic if the index is greater than the number of patterns. For example, the protocol "`Noise_NNpsk200_25519_ChaChaPoly_SHA256`" triggers a panic in `Builder::build_initiator` because the index 199 is out of bounds.

Exploit Scenario

Alice and Bob communicate in a protocol that involves negotiating a particular choice of Noise protocol. Alice tricks Bob into agreeing to a malicious protocol name with an invalid pre-shared-key index, and Bob's client crashes when building the protocol, causing a denial of service.

Recommendations

Short term, validate PSK indices when parsing protocol names and return an error value instead of panicking.

Long term, ensure that data validation functions return actionable errors whenever possible, and avoid panicking unless an error is unrecoverable.

6. Builder setter methods silently overwrite existing values

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SNOW-6
Target: `src/builder.rs`	

Description

Calling `Builder::prologue` or `Builder::psk` replaces any existing value without warning. This behavior is not documented and could lead to callers accidentally failing to incorporate pre-shared keys or portions of prologue data into handshake keys.

Figure 6.1 excerpts the code for `Builder::psk`

```
/// Specify a PSK (only used with `NoisePSK` base parameter)
pub fn psk(mut self, location: u8, key: &'builder [u8]) -> Self {
    self.psks[location as usize] = Some(key);
    self
}
```

Figure 6.1: `Builder::psk` silently replaces previous PSK values (`snow/src/builder.rs#99–103`)

Setting the PSK at a single location twice is likely to be an implementation error by the caller. Users may also assume that calling `Builder::prologue` multiple times will append the new data to the current prologue. The precise behavior is currently undocumented.

`Handshake::set_psk` also allows silently overwriting pre-set values.

Exploit Scenario

A user wants to incorporate several kinds of data into the handshake prologue. They serialize each and call `Builder::prologue` on each in turn. The calls succeed and the handshake proceeds but the resulting shared channel binds only the last call to `prologue`.

Recommendations

Short term, document the expected behavior when calling setter methods multiple times.

Long term, consider returning an error if a PSK is set on an already-filled location.

7. `CipherState::decrypt_ad` increments nonce even after authentication failure.

Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-SNOW-7
Target: `src/cipherstate.rs`	

Description

The Noise protocol framework specification describes the `DecryptWithAd` procedure as shown in figure 7.1.

- **`DecryptWithAd(ad, ciphertext)`**: If `k` is non-empty returns `DECRYPT(k, n++, ad, ciphertext)`. Otherwise returns `ciphertext`. If an authentication failure occurs in `DECRYPT()` then `n` is not incremented and an error is signaled to the caller.

Figure 7.1: Noise protocol section 5.1

However, the implementation in `CipherState::decrypt_ad` increments the nonce value n even when decryption fails, as shown in figure 7.2.

```
pub fn decrypt_ad(
    &mut self,
    authtext: &[u8],
    ciphertext: &[u8],
    out: &mut [u8],
) -> Result<usize, Error> {
    if (ciphertext.len() < TAGLEN) || out.len() < (ciphertext.len() - TAGLEN) {
        return Err(Error::Decrypt);
    }

    if !self.has_key {
        return Err(StateProblem::MissingKeyMaterial.into());
    }

    validate_nonce(self.n)?;
    let len = self.cipher.decrypt(self.n, authtext, ciphertext, out);

    // We have validated this will not wrap around.
    self.n += 1;

    len
}
```

Figure 7.2: Even if the `decrypt` call returns an error, n will be incremented (`snow/src/cipherstate.rs#47–68`)

This allows a malicious party that can send messages to a client to cause that client to increment its nonce state, potentially breaking protocol synchronization and causing all later messages to fail authentication.

Exploit Scenario

Alice and Bob establish an encrypted channel between themselves. Eve sends a junk message to Bob, causing him to increment his receiving nonce. He then uses an incorrect nonce when decrypting further messages from Alice, causing him to reject all messages and creating a denial of service.

Recommendations

Short term, return an error immediately when decryption fails – for example, by adding a question mark operator after the call to `decrypt`, e.g.:

```
pub fn decrypt_ad(
    &mut self,
    authtext: &[u8],
    ciphertext: &[u8],
    out: &mut [u8],
) -> Result<usize, Error> {
    if (ciphertext.len() < TAGLEN) || out.len() < (ciphertext.len() - TAGLEN) {
        return Err(Error::Decrypt);
    }

    if !self.has_key {
        return Err(StateProblem::MissingKeyMaterial.into());
    }

    validate_nonce(self.n)?;
    let len = self.cipher.decrypt(self.n, authtext, ciphertext, out)?;

    // We have validated this will not wrap around.
    self.n += 1;

    Ok(len)
}
```

Figure 7.3: A tweaked version of `decrypt_ad` which returns immediately if decryption fails

Long term, ensure that minimal processing happens when a ciphertext fails authentication, and especially ensure that state variables cannot be modified without successful authentication.

8. Ephemeral keys are not cleared from memory	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-SNOW-8
Target: `src/handshakestate.rs`	

Description

In order to ensure forward secrecy of communications encrypted by a Noise handshake, ephemeral Diffie-Hellman private keys and derived symmetric keys must be deleted after use. The Snow library currently does not implement any form of zeroization for secret data, so secret keys may remain in RAM after they go out of scope.

Noise handshakes aim to provide certain forms of post-compromise security. Forward secrecy attempts to ensure that an adversary who can observe network messages and later compromises a device should not be able to decrypt the previously observed communication. Forward secrecy is achieved by encrypting all messages under ephemeral keys which are discarded after use. However, if developers do not take care to securely delete the key material from RAM, an adversary with root or physical access to a device may be able to retrieve old key material and decrypt past communications.

Exploit Scenario

Alice sends a sensitive message to Bob, using the KK handshake pattern. An attacker, Eve, records the ciphertext and handshake messages. Because of the sensitive nature of the message, Alice deletes the message from her device after sending. Later, Eve steals Alice's laptop and gains root access to the device. She inspects the process memory and finds uncleared keys. Eve uses the keys to decrypt the previously captured message and break the protocol's confidentiality.

Recommendations

Short term, use the zeroize crate to implement zeroize-on-drop wherever possible.

Long term, consider using the secrecy crate to prevent copies of sensitive data from being left in RAM.

9. PSK modifiers can be repeated in handshake pattern names

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SNOW-9
Target: `src/params/patterns.rs`	

Description

Handshake patterns are selected by a pattern name followed by a potentially-empty list of modifiers delimited by the "+" character. The `pskN` modifier specifies that a pre-shared key should be mixed into the handshake at a position selected by the integer `N`. However, the `snow` parser does not prevent repeated use of the same `pskN` modifier, leading to repeated mixing of the same pre-shared key into the state at that point in the handshake.

We do not believe this leads to any attacks, and it is neither explicitly allowed nor explicitly forbidden by the Noise Protocol Framework specification. However, a user of the library who misses a typo may unwittingly use a handshake which ignores one of their preshared keys. For example, a user who uses "`Noise_XXpsk0+psk0_25519_AESGCM_SHA256`" instead of "`Noise_XXpsk0+psk1_25519_AESGCM_SHA256`" will only use one pre-shared key during the handshake. Since setting a pre-shared key without using it does not cause the library to report an error, as shown in figure 9.1, this user will receive no feedback and may not have a protocol which is as secure as they expect.

```
/// Set the preshared key at the specified location. It is up to the caller
/// to correctly set the location based on the specified handshake - Snow
/// won't stop you from placing a PSK in an unused slot.
///
/// # Errors
///
/// Will result in `Error::Input` if the PSK is not the right length or the location
is out of bounds.
pub fn set_psk(&mut self, location: usize, key: &[u8]) -> Result<(), Error> {
    if key.len() != PSKLEN || self.psks.len() <= location {
        return Err(Error::Input);
    }

    let mut new_psk = [0u8; PSKLEN];
    new_psk.copy_from_slice(key);
    self.psks[location] = Some(new_psk);

    Ok(())
}
```

Figure 9.1: Setting pre-shared keys which will not be used may not report an error (`src/handshakestate.rs#454–471`)

Recommendations

Short term, either explicitly specify the behavior of repeated use of the same `pskN` modifier, or add validation to reject it. Consider adding additional validation to the `psks` array in `HandShakeState` so that setting a pre-shared key which will not get used causes an error to be reported.

Long term, document all implementation decisions made in areas where the Noise Protocol Framework is ambiguous.

10. Resolvers behave differently with long and short keys

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SNOW-10
Target: `src/resolvers/{default,libsodium,ring}.rs`	

Description

The `snow` library uses cryptographic primitives through an interface to various "resolvers" which provide those primitives. The three implemented resolvers are `DefaultResolver`, which wraps some commonly used pure-rust implementations of these primitives; `SodiumResolver`, which wraps the `sodiumoxide` bindings for `libsodium`; and `RingResolver`, which wraps the `ring` library. Diffie-Hellman key exchange and authenticated encryption are handled through the traits `Dh` and `Cipher`, shown below in figures 10.1 and 10.2.

```
pub trait Dh: Send + Sync {
    /// The string that the Noise spec defines for the primitive
    fn name(&self) -> &'static str;

    /// The length in bytes of a public key for this primitive
    fn pub_len(&self) -> usize;

    /// The length in bytes of a private key for this primitive
    fn priv_len(&self) -> usize;

    /// Set the private key
    fn set(&mut self, privkey: &[u8]);

    /// Generate a new private key
    fn generate(&mut self, rng: &mut dyn Random);

    /// Get the public key
    fn pubkey(&self) -> &[u8];

    /// Get the private key
    fn privkey(&self) -> &[u8];

    /// Calculate a Diffie-Hellman exchange.
    fn dh(&self, pubkey: &[u8], out: &mut [u8]) -> Result<(), Error>;
}
```

Figure 10.1: The Diffie-Hellman trait (`src/types.rs#13–37`)

```
pub trait Cipher: Send + Sync {
    /// The string that the Noise spec defines for the primitive
    fn name(&self) -> &'static str;

    /// Set the key
    fn set(&mut self, key: &[u8]);

    /// Encrypt (with associated data) a given plaintext.
    fn encrypt(&self, nonce: u64, authtext: &[u8], plaintext: &[u8], out: &mut [u8])
-> usize;

    /// Decrypt (with associated data) a given ciphertext.
    fn decrypt(
        &self,
        nonce: u64,
        authtext: &[u8],
        ciphertext: &[u8],
        out: &mut [u8],
    ) -> Result<usize, Error>;

    /// Rekey according to Section 4.2 of the Noise Specification, with a default
    /// implementation guaranteed to be secure for all ciphers.
    fn rekey(&mut self) {
        let mut ciphertext = [0; CIPHERKEYLEN + TAGLEN];
        let ciphertext_len = self.encrypt(u64::MAX, &[], &[0; CIPHERKEYLEN], &mut
ciphertext);
        assert_eq!(ciphertext_len, ciphertext.len());
        self.set(&ciphertext[..CIPHERKEYLEN]);
    }
}
```

Figure 10.2: The AEAD cipher trait (`src/types.rs#40–67`)

Note that both of these traits have a `set()` method which sets the affiliated key based on a potentially-variable-length byte slice. The implementations of these traits each have somewhat different behavior when the `key` argument to `set()` differs in length from a proper key.

For the two implementations of the Dh trait:

- The default resolver will silently accept secret keys which are shorter than 32 bytes, but will panic if given a longer key.
- The `libsodium` resolver will panic when given a key which is not 32 bytes in length

For the implementations of the `Cipher` trait:

- Due to the use of the `copy_slices!` macro, the default resolver will silently accept keys shorter than 32 bytes and will panic if given a key longer than 32 bytes.

- The `libsodium` resolver will silently allow keys longer than 32 bytes, but will panic if given a key shorter than 32 bytes.
- The `ring` resolver will panic if given a key which is not 32 bytes long.

It appears that most uses of these `set()` methods guarantee that the `key` parameter will be the correct length, except in the `Builder` type, where the local private key field `s` and the testing-only fixed ephemeral key field `e_fixed` can each be set with an arbitrary-length byte sequence, as shown in figures 10.3 and 10.4.

```rust
/// Your static private key (can be generated with [`generate_keypair()`]).
///
/// [`generate_keypair()`]: #method.generate_keypair
pub fn local_private_key(mut self, key: &'builder [u8]) -> Self {
    self.s = Some(key);
    self
}

#[doc(hidden)]
pub fn fixed_ephemeral_key_for_testing_only(mut self, key: &'builder [u8]) -> Self {
    self.e_fixed = Some(key);
    self
}
```

Figure 10.3: `src/builder.rs#105–117`

```rust
let s = match self.s {
    Some(k) => {
        (*s_dh).set(k);
        Toggle::on(s_dh)
    },
    None => Toggle::off(s_dh),
};

if let Some(fixed_k) = self.e_fixed {
    (*e_dh).set(fixed_k);
}
let e = Toggle::off(e_dh);
```

Figure 10.4: Usage of s and `e_fixed` *(*`src/builder.rs#178–189`*)*

Although most uses of these traits in the `snow` library guarantee that the `key` arguments are the correct length, these behavioral differences are a potential source of confusion and errors for users of the library.

Recommendations

Short term, make the handling of the `key` parameters consistent across all resolvers. Consider replacing the slice type `&[u8]` with a fixed-length array type, e.g. `[u8; CIPHERKEYLEN]` in the case of AEAD ciphers.

Long term, specify and document the expected input validation for interface traits such as `Dh` and `Cipher`.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

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

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix lists findings that are not directly security-relevant but may impact readability, maintainability or efficiency.

- Functions used only for testing should be prefixed with `#[cfg(test)]`.

```
#[doc(hidden)]
pub fn fixed_ephemeral_key_for_testing_only(mut self, key: &'builder [u8]) ->
Self {
    self.e_fixed = Some(key);
    self
}
```

Figure C.1: Testing-only function publicly exposed in non-test builds

- Use of `OsRng` in test leads to compilation error when combining `--no-default-features` with `--features libsodium-resolver`.

```
#[test]
fn test_curve25519_shared_secret() {
    let mut rng = OsRng::default();

    // Create two keypairs.
    let mut keypair_a = SodiumDh25519::default();
    keypair_a.generate(&mut rng);
```

Figure C.2: When testing libsodium, code should use `SodiumRng::default`

- The Kyber1024 KEM implementation uses an internal RNG for key generation and encapsulation, rather than the resolver-provided RNG. This may present issues when compiling for embedded devices, HSMs, and other no-std environments.
- There are a few instances where panics are possible inside functions that otherwise return `Error` types. Detect these with the `p/trailofbits` Semgrep ruleset.

D. Automated Analysis Tool Configuration

As part of this assessment, we used the tools described below to perform automated testing of the codebase.

D.1. Semgrep

The semgrep linter checks code for known problematic patterns based on community-published rules. We used the command

```
semgrep --config "p/rust" --config "p/trailofbits"
```

to run the lints.

D.2. cargo llvm-cov

`cargo-llvm-cov` generates Rust code coverage reports. We used the `cargo llvm-cov --open` command in the codebase to generate the coverage report presented in the Automated Testing section.

D.3. cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.