AgileBits Snow

Security Assessment

March 29, 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 5, 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
March 29, 2024	Delivery of comprehensive report with fix review appendix

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 cryptographic handshake 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 Framework 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, is well-organized, and follows the protocol specification. However, we discovered multiple issues related to the handling of improperly formatted data (TOB-SNOW-3, TOB-SNOW-4, TOB-SNOW-5, TOB-SNOW-7, and TOB-SNOW-9) that could have been uncovered by more extensive testing, particularly focused on edge cases and adversarial behavior. Although most of those findings are informational, TOB-SNOW-5 can cause a panic, and TOB-SNOW-7 allows an adversary on the network to permanently disrupt an encrypted channel without knowing any cryptographic secrets.

We also found several portions of the library that could cause problems if more features are added to Snow or if they are used in nonstandard ways, including improper handling of handshake patterns that are not currently available (TOB-SNOW-2), methods that silently overwrite previous values of configuration variables (TOB-SNOW-6), and inconsistent handling of incorrectly sized keys (TOB-SNOW-10).

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 that include adversarial interference.** 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 the library's functionality in post-error states.

- **Use code coverage tooling to guide further testing.** There are some aspects of the Noise handshake that 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 Framework specification, in terms of cryptographic principles and primitives?
- Are there any additional off-by-one errors that could allow invalid nonce use?
- Does the implementation follow and enforce proper state transitions, such as requiring a completed `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 (DoS)?
- Are all inputs and system parameters properly validated?
- Does the codebase conform to industry best practices?
- Are there any areas for improvement in the CI/CD or software development life cycle?

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:

- 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 that are not covered by unit tests. For example, no unit tests exercise the code path in the `src/handshakestate.rs` file, which handles using a preshared key (PSK) slot that 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)**

Figure 1: Per-file code coverage results

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

Figure 2: Example of missing coverage on an error condition

cargo-audit. The tool identified one unmaintained core dependency (TOB-SNOW-1).

Semgrep. The tool identified multiple instances of `expect` and `unwrap` that might lead to undesired panics. However, we did not identify any way to trigger these cases.

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	After checking potential instances returned by the `clippy::arithmetic-side-effects` lint, we did not find any instances of unchecked arithmetic overflow.	**Strong**
Auditing	No logging functionality is 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). Additionally, cryptographic keys are not zeroized after use (TOB-SNOW-8).	**Moderate**
Documentation	There are sufficient code examples and documentation to help users interact with the library. However, some functions need documentation to describe their behavior when they are used outside of typical patterns (TOB-SNOW-6, TOB-SNOW-9, and TOB-SNOW-10).	**Satisfactory**
Memory Safety and Error Handling	The codebase does not use unsafe Rust. There is 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 cover all logic, especially those cases triggered by uncommon handshake patterns. Some fuzzing is present, but it 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 PSK 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: **High**
Type: Patching	Finding ID: TOB-SNOW-1
Target: `Cargo.toml`, `src/resolvers/libsodium.rs`	

Description

The Snow library allows users to choose between several back ends that implement the cryptographic primitives required by the Noise protocol. One provided back end uses `libsodium` via the `sodiumoxide` Rust binding library, which has been deprecated and is listed by the Rust 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.

Exploit Scenario

A security vulnerability is found in the `sodiumoxide` bindings. Because the library is unmaintained, no fix or alert is put out, leaving the Snow library vulnerable to exploitation.

Recommendations

Short term, consider deprecating the `sodiumoxide` back end or adding a warning to users that the back end 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 PSK and a pre-message ephemeral public key, the Snow implementation does not mix the ephemeral public key into the initial encryption key state, potentially allowing catastrophic nonce reuse in valid, nonstandard 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 pre-message ephemeral keys into the encryption key is required by the Noise protocol specification 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.*

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 that include a pre-message ephemeral key (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 2.1, which includes a static key in the first initiator message, the static key will be encrypted with an AEAD key and a nonce that are constant among different runs of the protocol.

```
Examplepsk0:
  <- s
  <- e
  ...
  -> psk, s, ss, e, es
  <- ee, se
```

Figure 2.1: Valid handshake pattern triggering catastrophic nonce reuse

Exploit Scenario

A user extends the library to support a nonstandard handshake pattern such as the one shown in figure 2.1. Two users initiate the protocol using the same PSK. 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, modify the code to use `MixKey` operations 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 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: The message length is not checked when receiving messages. (`snow/src/transportstate.rs#78–86`)

While this issue does not cause any immediate security issues, allowing long messages to be received could allow a potential DoS attack or partitioning oracle attack, depending on the exact threat model and deployed configuration.

Recommendations

Short term, add validation to the `read_message` function to reject messages that are longer than 65,535 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 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 performed on 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 `Noise_XX_25519_AESGCM_SHA256`, `Noise_XX_25519_AESGCM_SHA256_`, and `Noise_XX_25519_AESGCM_SHA256_OtherThingsThatShouldNotBeHere` protocol names all correspond to using the XX handshake pattern, the X25519 DH function, the AES-GCM cipher, and the SHA-256 hash function, but each will have a different initial chaining key.

In addition to protocol names that lead to different initial chaining keys, there are also some protocol names that are different but lead to identical initial chaining keys. The initial chaining key is initialized by loading the protocol name into a buffer that is prefilled with zero bytes, as shown in figure 4.2. By appending an underscore followed by null characters, it is possible to create protocols that have different 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. However, the underscore is necessary because 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 that 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 PSK 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 plus sign [+]. 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 PSK should be mixed into the state. The Snow library implements the PSK modifier by separately parsing the pattern name and the modifiers, then calling the `apply_psk_modifier` function, shown in figure 5.1, on the `Pattern` object that is 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 PSK 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 `Noise_NNpsk200_25519_ChaChaPoly_SHA256` protocol triggers a panic in the `Builder::build_initiator` function 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 PSK index, and Bob's client crashes when building the protocol, which causes a DoS.

Recommendations

Short term, modify the code to validate PSK indices when parsing protocol names and to 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: **High**
Type: Data Validation	Finding ID: TOB-SNOW-6
Target: `src/builder.rs`	

Description

Calling the `Builder::prologue` or `Builder::psk` functions replaces any existing value without warning. This behavior is not documented and could lead to callers accidentally failing to incorporate PSKs 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 made 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.

The `Handshake::set_psk` function also allows silently overwriting preset 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 one 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 modifying the code to return 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 specification section 5.1 describes the `DecryptWithAd` procedure as follows:

> ***`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.*

However, the implementation in the `CipherState::decrypt_ad` function increments the nonce value n even when decryption fails, as shown in figure 7.1.

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

This would allow 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 DoS.

Recommendations

Short term, modify the code to immediately return an error when decryption fails—for example, by adding a question mark operator after the call to `decrypt`:

```
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 7.2: A tweaked version of the `decrypt_ad` function that returns immediately if decryption fails

Long term, ensure that minimal processing happens when a ciphertext fails authentication, and, in particular, 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

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 using ephemeral keys that are discarded after use. However, if developers do not securely delete the key material from RAM, an adversary with root or physical access to a device may 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 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 plus sign [+]. The `pskN` modifier specifies that a PSK should be mixed into the handshake at a position indicated by the integer `N`. However, the Snow parser does not prevent repeated use of the same `pskN` modifier, which allows repeated mixing of the same PSK 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 specification. However, a library user who misses a typo may unwittingly use a handshake that ignores one of their PSKs.

For example, a user who uses `Noise_XXpsk0+psk0_25519_AESGCM_SHA256` instead of `Noise_XXpsk0+psk1_25519_AESGCM_SHA256` will use only one PSK during the handshake. Since setting a PSK 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 that 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 PSKs that will not be used may not return an error.
(`src/handshakestate.rs#454–471`)

Recommendations

Short term, either explicitly specify the behavior that occurs with repeated use of the same `pskN` modifier, or add validation to reject it. Consider adding validation to the `psks` array in the `HandShakeState` struct so that setting a PSK that will not be used causes an error to be returned.

Long term, document all implementation decisions made in areas where the Noise protocol specification 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 that 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. The Diffie-Hellman key exchange (DHKE) and authenticated encryption are handled through the `Dh` and `Cipher` traits, shown 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 Dh 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`)

Both of these traits have a `set()` method that 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 following events will occur:

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

For the implementations of the `Cipher` trait, the following events will occur:

- 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 that is not 32 bytes long.

It appears that most uses of these `set()` methods guarantee that the `key` argument 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.

```
/// 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: The `key` argument lengths are unchecked. (`src/builder.rs#105–117`)

```
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: Use 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 library users. For example, a developer who improperly uses the `local_private_key()` method with a short private key may not notice the problem, reducing authentication security and allowing a person-in-the-middle attack that would otherwise not be possible.

Recommendations

Short term, make the handling of the `key` arguments consistent across all resolvers. Consider replacing the slice type `&[u8]` with fixed-length array types, such as `[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**
Not Applicable	There is no exploit scenario for this issue, so the difficulty level does not apply.
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.

- A testing-only function is publicly exposed in release builds. 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-testing builds (`snow/src/builder.rs#L113–L117`)

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

```
#[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`, the code should use the `SodiumRng::default()` object. (`snow/src/resolvers/libsodium.rs#L287–L293`)

- The Kyber1024 KEM implementation uses an internal random number generator (RNG) for key generation and encapsulation, rather than the resolver-provided RNG. This may present issues when compiling for embedded devices, hardware security modules (HSMs), and other `no_std` environments.

- There are a few instances where `expect` and `unwrap` are used 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 following command to run the lints:

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

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.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From March 4 to March 6, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Snow team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 10 issues described in this report, the Snow team has resolved eight issues and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	sodiumoxide dependency is deprecated and unmaintained	Resolved
2	Pre-message ephemeral keys are not mixed into the encryption key	Unresolved
3	Received messages can exceed maximum message length	Resolved
4	Noise protocol names can have extra data appended with no effect	Resolved
5	Protocol builder panics on invalid PSK indices	Resolved
6	Builder setter methods silently overwrite existing values	Resolved
7	CipherState::decrypt_ad increments nonce even after authentication failure	Resolved
8	Ephemeral keys are not cleared from memory	Unresolved
9	PSK modifiers can be repeated in handshake pattern names	Resolved

10	Resolvers behave differently with long and short keys	**Resolved**

Detailed Fix Review Results

TOB-SNOW-1: sodiumoxide dependency is deprecated and unmaintained
Resolved in commit `87b7f608e4c7f60b397bcfb88df73f2579a0bc4f`. The `sodiumoxide` dependency has been deprecated.

TOB-SNOW-2: Pre-message ephemeral keys are not mixed into the encryption key
Unresolved. The Snow team has reported that this is considered low priority until fallback modifiers are supported in Snow.

TOB-SNOW-3: Received messages can exceed maximum message length
Resolved in commit `c6657a7cadaa6e67cd636d6ed2c04dd354157d94`. The `read_message` implementations now compare the length of the message to the `MAXMSGLEN` constant.

TOB-SNOW-4: Noise protocol names can have extra data appended with no effect
Resolved in commit `8161a5fe2fcd8c006874217359074e5e0bb181d4`. The `from_str` implementations now check that the string-split iterator has been depleted and return an error if there is an extraneous underscore.

TOB-SNOW-5: Protocol builder panics on invalid PSK indices
Resolved in commit `270c9fd388a7cd8c9d12e114b13e04832c9bf858`. The `apply_psk_modifier` function now returns a `Result` value and returns an error if the index is out of bounds.

TOB-SNOW-6: Builder setter methods silently overwrite existing values
Resolved in commit `23bc38f054542903889d3de06a849843daadc593`. The `psk`, `local_private_key`, `prologue`, and `remote_public_key` functions now detect attempted overwrites and return a `ParameterOverwrite` error instead of overwriting the parameter.

TOB-SNOW-7: CipherState::decrypt_ad increments nonce even after authentication failure
Resolved in commit `5b451e9507b6a70d762600cda40bfd9a7428e093`. Decryption failure now uses the question mark [?] operator to exit early, and a unit test has been added to explicitly check that a corrupted message does not modify the state but a successful message does.

TOB-SNOW-8: Ephemeral keys are not cleared from memory
Unresolved. The Snow team has reported that a patch is being developed, but it was not available for the fix review period.

TOB-SNOW-9: PSK modifiers can be repeated in handshake pattern names

Resolved in commit `671e0b9699e140c55d6f33d3d507338b65892c2e`. Handshake modifier list parsing now checks for repeated modifiers and returns a `DuplicateModifier` error if such a modifier is found.

TOB-SNOW-10: Resolvers behave differently with long and short keys

Resolved in commit `357fbac3f3894ebe1bbbf66d7e2271b418c58b6d`. `Cipher::set` now takes a fixed-length array. The `Dh::set` method in the default resolver still accepts short byte sequences for the secret key. The Snow team should consider modifying the `Dh::set` method to take a fixed-length byte sequence. However, since the `sodiumoxide` resolver has been deprecated, the behavior between supported resolvers no longer differs.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.