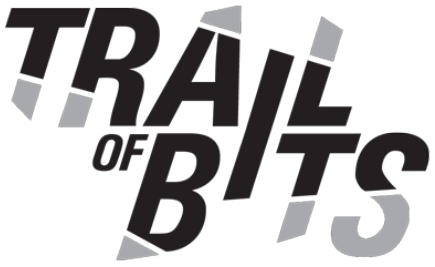

Standard Notes

Security Assessment

March 9, 2020

Prepared For:
Mo Bitar | *Standard Notes*
mo@standardnotes.org

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

Claudia Richoux | *Trail of Bits*
claudia.richoux@trailofbits.com

Executive Summary

From March 2 through March 6, 2020, Standard Notes engaged Trail of Bits to review the security of SNJS and SNCrypto. Trail of Bits conducted this assessment over the course of one person-week with two engineers working from commit `b9d7b79` on branch 004 from the `standardnotes/snjs` repository, along with commit `0059a66` on branch 004 of the `standardnotes/sncrypto` repository.

At the beginning of this one-week assessment, we reviewed the relevant documentation for SNJS and SNCrypto, and gained an overall understanding of the system. From there, we began manually reviewing the components of both SNJS and SNCrypto, and actively engaged with the Standard Notes team to discuss our findings. We also integrated CodeQL, a static analyzer, into both codebases to help understand them and identify common security issues (see Appendix C for more details).

Our manual review of the codebase revealed four findings. We report one medium-severity issue, TOB-SNOTES-001, related to insecure passwords. The remaining three, TOB-SNOTES-002–TOB-SNOTES-004, are informational findings related to values leaked to timing side-channels, and values not being cleared after they are no longer needed.

Trail of Bits performed an assessment of protocol version 004. Standard Notes provided thorough documentation for this protocol. We recommend adjusting this protocol to guarantee backward secrecy, which can be found in Appendix F.

Besides that, we found this protocol to be robust, and we report no findings related to the design of this protocol. Further, we found that Standard Notes uses strong and modern cryptographic primitives in their design; the strength of these primitives limited the feasibility of exploitation for some of our findings. In addition, we found that Standard Notes employs well-accepted coding practices. We have included recommendations for improving the code quality and architecture in Appendix B and Appendix D (respectfully), but these represent improvements to an already strong codebase. Lastly, due to time limitations, our assessment consisted mainly of manual review. Given more time, we would like to integrate fuzzing into the codebases and perform a more in-depth manual review of SNJS.

We encourage Standard Notes to integrate fuzzing into their codebases. Fuzzing is a great way to find bugs from unexpected behavior not encapsulated in the unit tests. We also encourage Standard Notes to vigilantly protect secrets stored in memory, and, when possible, ensure these values are cleared once they are no longer needed. Further, we encourage Standard Notes to consider our recommendations detailed in our appendices: Appendix D details a recommendation for refactoring the codebase using TypeScript to

help achieve more secure, maintainable code; Appendix E details recommendations for enforcing secure passwords; and Appendix F details a recommendation for guaranteeing backward secrecy.

Update: On September 8, 2020, Trail of Bits reviewed fixes implemented for the issues presented in this report. Standard Notes also implemented the recommendations described in Appendix D and Appendix F. For more details on the review of these changes, see Appendix G.

Project Dashboard

Application Summary

Name	SNJS, SNCrypto
Version	004
Type	JavaScript
Platforms	Desktop, web, mobile

Engagement Summary

Dates	March 2–6, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	1 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	0	
Total Informational-Severity Issues	3	■ ■ ■
Total	4	

Category Breakdown

Configuration	1	■
Data Exposure	2	■ ■
Data Validation	1	■
Total	4	

Engagement Goals

The engagement was scoped to provide a security assessment of the 004 branches of both the SNJS and SNCrypto code repositories. The 004 branch represents the fourth version of the Standard Notes protocol. We assessed the cryptography and overall security of this protocol version.

Specifically, we sought to answer the following questions:

- Does the provided specification of version 004 achieve its design goals?
- Does the provided specification have any flaws?
- Does the protocol use safe cryptographic primitives, and use them correctly?
- Do the SNJS and SNCrypto implementations comply with the claims of the provided specification?
- Are secret values cleared from memory after they are no longer needed?
- Is any secret information leaked to timing side-channels?
- Client root keys and server passwords are generated with `argon2id` and then split. Should each key also be input into a HKDF?
- Item's `uuid` and protocol version are included as part of the additional authentication parameters for authenticated encryption. Should `items_key_id` also be included in the additional authentication parameters?

Coverage

The assessment of SNJS and SNCrypto was primarily performed through manual review. First, we manually reviewed the protocol described in the provided specification. Then we manually reviewed both codebases for their compliance with the specification, and conducted general security and cryptographic reviews.

In addition to the manual review, we also integrated CodeQL, a static analysis tool, into both codebases (see Appendix C for more details). This allowed us to better understand how portions of the codebases interacted with each other. CodeQL also helps identify common security and code quality issues, and fuzzing targets. Unfortunately, due to the brevity of the assessment, we were unable to integrate fuzzing into these codebases. If Standard Notes is interested in integrating fuzzing into their codebases, we encourage them to use Burp Suite.

Recommendations Summary

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

Short Term

❑ **Adjust the code responsible for handling password changes to enforce a minimum length.** This will help prevent users from using very insecure passwords. (TOB-SNOTES-001)

❑ **Whenever possible, use Javascript array buffers to store long-lived cryptographic keys instead of strings.** These are mutable and do not live on the heap, which allows you to control the storage and expiration of sensitive data. Be sure to add routines zeroing the buffers as soon as they are no longer necessary. (TOB-SNOTES-002)

❑ **Further, ensure all references to the password or passcode strings are out of scope after the application has finished using them.** This will help ensure secret values are not leaked. (TOB-SNOTES-002)

❑ **Refactor the `compare` function to include a secure, constant-time comparison function.** This will prevent timing information from leaking information about the root key. (TOB-SNOTES-003)

❑ **Adjust the mobile migration code to clear the keychain of the `keys.offline.pw` values once migration has occurred.** This value is no longer needed and should be cleared from memory to comply with the protocol specification. (TOB-SNOTES-004)

Long Term

❑ **Consider enforcing stronger password requirements in addition to a minimum length.** Users tend to choose weak passwords, and adding stronger password requirements is a good security practice. See Appendix E for further details. (TOB-SNOTES-001)

❑ **Consider the tradeoffs of storing data in a mutable data type that can be zeroed out, or storing it in a data type that doesn't live on the heap.** If the data is not zeroed out when the application is done using it, this gives an attacker a window to dump the memory and try to recover it. If the data is on the heap, JS engines with a GGC could move it to multiple locations and take much longer to overwrite it once all references are gone. (TOB-SNOTES-002)

❑ **Be vigilant of all comparisons between secret values, and ensure that the operations do not leak any information.** Timing information can unintentionally leak secrets, with devastating results. (TOB-SNOTES-003)

❑ **Document where imperative secret values are stored and clear them from those locations when they are no longer needed.** Leaving unneeded values in memory gives no benefit to the user and can help an attacker learn more information. (TOB-SNOTES-004)

Findings Summary

#	Title	Type	Severity
1	Small, insecure passwords are allowed when users change passwords	Configuration	Medium
2	Secrets remain in memory for undetermined amount of time	Data Exposure	Informational
3	Timing information on root key comparison could leak part of root key	Data Exposure	Informational
4	`Keys.offline.pw` value not cleared in `migrateStorageStructureForMobile`	Data Validation	Informational

1. Small, insecure passwords are allowed when users change passwords

Severity: Medium
Difficulty: High
Type: Configuration
Finding ID: TOB-SNOTES-001
Target: `snjs/lib/services/api/session_manager.js`

Description
Standard Notes provides an API for account management, which includes registering an account (email and password), changing passwords, and more. When a user attempts to register for an account, Standard Notes verifies that the password is at least eight characters (see Figure 1.1).

```
async register({ email, password }) {
   if (password.length < MINIMUM_PASSWORD_LENGTH) {
     return this.apiService.error(
       messages.InsufficientPasswordMessage(MINIMUM_PASSWORD_LENGTH)
     );
   }
   ...
   return this.apiService.register({
     email,
     serverPassword,
     keyParams
   }).then(async (response) => {
     await this.handleAuthResponse(response);
     return this.returnAfterTimeout({
       response: response,
       keyParams: keyParams,
       rootKey: rootKey
     });
   });
 }
```

Figure 1.1: Function for registering email and password.

However, the logic for a password change request does not include this same check (see Figure 1.2). Therefore, the API does not protect against users submitting very short and very insecure passwords when requesting a password change.

```
async changePassword({ email, currentPassword, currentKeyParams, newPassword }) {
   const currentServerPassword = await this.protocolService.computeRootKey({
     password: currentPassword,
     keyParams: currentKeyParams,
```

```
    }).then((key) => {
      return key.serverPassword;
    });
    const { newServerPassword, newRootKey, newKeyParams } = await
this.protocolService.createRootKey({
      identifier: email,
      password: newPassword
    }).then((result) => {
      return {
        newRootKey: result.key,
        newServerPassword: result.key.serverPassword,
        newKeyParams: result.keyParams
      };
    });
    return this.apiService.changePassword({
      email,
      currentServerPassword,
      newServerPassword,
      newKeyParams
    }).then(async (response) => {
      await this.handleAuthResponse(response);
      return this.returnAfterTimeout({
        response: response,
        keyParams: newKeyParams,
        rootKey: newRootKey
      });
    });
  }
```

Figure 1.2: Function for changing password.

Exploit Scenario

An attacker, Eve, controls a server, and she is aware that the Standard Notes API allows for weak passwords upon change requests. Therefore, Eve keeps track of all accounts that change their passwords and attempts to attack their accounts.

Alice registers an account with Standard Notes, requests a password change, and submits a very small, insecure password. Eve notices Alice's password change, successfully attacks her account, and steals all of her files.

Recommendation

Short term, adjust the code responsible for handling password changes to enforce a minimum length.

Long term, consider enforcing stronger password requirements in addition to a minimum length. See Appendix E for further details.

2. Secrets remain in memory for undetermined amount of time

Severity: Informational
Type: Data Exposure
Target: `various`

Difficulty: High
Finding ID: TOB-SNOTES-002

Description
Part of the Standard Notes threat model asserts that private data should be inaccessible when the application is locked. Item keys and encrypted item keys remain on the heap with references to them until a garbage collector (GC) sweep.

Because of optimizations like generational garbage collectors (GGCs) and GCs that wait for low CPU usage to do a sweep, this could be quite some time, especially if the GGC moves data from the nursery to the tenured heap because it has been on the heap for some time. The Javascript GC also makes no promises as to when various parts of the heap will be overwritten with other data, so sensitive data will likely also be accessible for some time after a sweep, with a bit of heap analysis.

Unfortunately, this issue is unavoidable to some extent. In this system, secrets must be stored in strings in order to interact with particular APIs. Once data is stored in strings, there is no guarantee when the values will be cleared by the GC. Therefore, this issue will only be completely avoidable when strings are not required.

Exploit Scenario
An attacker who can run code on the client device could potentially stall a GC sweep while dumping the application's process memory. If the attacker stalls the sweep long enough, they can access sensitive data and some useful references and associated data to make recovering compromised keys even easier. Even without a sweep, sensitive data—including passwords, the root key, decrypted notes, or item keys—could remain in memory leaked to an attacker.

Recommendation
Short term, whenever possible, use Javascript array buffers to store long-lived cryptographic keys instead of strings. These are mutable and do not live on the heap, which allows you to control the storage and expiration of where the sensitive data goes and its destruction when it is destroyed. Be sure to add routines zeroing the buffers as soon as they are no longer necessary. Further, ensure all references to the password or passcode strings are out of scope after the application has finished using them.

Long term, consider the tradeoffs of storing data in a mutable data type that can be zeroed out, or storing it in a data type that doesn't live on the heap. If the data is not zeroed out when the application is done using it, this gives an attacker a window to dump the memory and try to recover it. If the data is on the heap, JS engines with a GGC could move it to multiple locations and take much longer to overwrite it once all references are gone.

3. Timing information on root key comparison could leak part of root key

Severity: Informational | Difficulty: High
Type: Data Exposure | Finding ID: TOB-SNOTES-003
Target: `snjs/lib/services/key_manager.js`, `snjs/lib/protocol/root_key.js`

Description
Standard Notes derives a user's root key from their password. The Standard Notes application interface provides a method, `validateAccountPassword`, which will validate a given password. To perform this validation, the password is input into the key derivation function (`argon2id` in version 004), and the result is compared against the root key (see Figure 3.1).

```
/**
  * @param {string} password  The password string to generate a root key from.
  * @returns {key|null}  The computed rootKey if valid password, otherwise null.
  */
 async validateAccountPassword(password) {
   const keyParams = await this.getRootKeyParams();
   const key = await this.protocolService.computeRootKey({ password, keyParams });
   const success = key.compare(this.rootKey);
   return success ? key : null;
 }
```

Figure 3.1: Function for validating account password.

To compare the generated key with the root key, the `compare` function is called. This function performs a comparison using the JavaScript `===` operator (see Figure 3.2). While any attacker with access to this function would also have direct access to the root key, this operator in general does not have any timing guarantees. Timing information on this comparison could be leveraged to learn parts of the root key.

```
/**
  * Compares two keys for equality
  * @returns {boolean} true if equal, otherwise false.
 */
 compare(otherKey) {
   if (this.version !== otherKey.version) {
     return false;
   }
   const hasServerPassword = this.serverPassword && otherKey.serverPassword;
   return (
```

```
      this.masterKey === otherKey.masterKey &&
      (!hasServerPassword || this.serverPassword === otherKey.serverPassword)
    );
  }
```

Figure 3.2: Function for comparing root keys.

Exploit Scenario

Alice sets up an account with Standard Notes. Eve wants to break into Alice's account and notices that Standard Notes is using a comparison that leaks information about the root key. Eve then attempts various passwords, some of which leak information about the root key.

Recommendation

Short term, refactor the `compare` function to include a secure, constant-time comparison function.

Long term, be vigilant of all comparisons between secret values and ensure that the operations do not leak any information.

References

- Preventing Timing Attacks on String Comparison

4. `keys.offline.pw` value not cleared in `migrateStorageStructureForMobile`

Severity: Informational | Difficulty: High
Type: Data Validation | Finding ID: TOB-SNOTES-004
Target: `snjs/lib/migration/migrations/2020-01-15.js`

Description
Standard Notes includes a few files related to a system migration. This migration primarily shifts how certain values are stored. Among other functions, `migrateStorageStructureForMobile` performs storage migration for legacy versions. This function is responsible for migrating the wrapped account key into the `rawStructure`.

In the old version, mobile systems stored the `keys.offline.pw` value in the device's keychain. This value, also known as `serverPassword`, corresponds to the second half of the output of `argon2id` computed from the user's password. To verify a password, the old systems would input a password into the KDF (`pbkdf2` in the old system) and compare the second half against this stored value. The new version will instead verify passwords by attempting decryption and accepting the password if it does not fail.

To perform this migration, the password must be verified one last time via the old method. Once the password is verified, the wrapped account key can be placed in the `rawStructure`.

The goal is to migrate the wrapped key into the new storage and switch password verification to the new version. This means that passwords will not be verified by comparison against `keys.offline.pw`, and this value should be cleared from the keychain. However, in the code performing this migration, this value is not cleared.

Exploit Scenario
Alice uses Standard Notes on a legacy mobile platform and performs this migration. Normally, an attacker learning `keys.offline.pw` would not be a problem. However, Alice believes that after the migration is performed, `keys.offline.pw` is no longer statically stored and decides to use this value elsewhere to secure some other secret information. An attacker, Eve, is able to read this value from the keychain and recover Alice's secrets.

Recommendation
Short term, adjust this code to clear the keychain of these values once migration has occurred.

Long term, document where imperative secret values are stored and clear them from those locations when they are no longer needed.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

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

B. Non–Security-Related Findings

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

- The Standard Notes account management API provides a function for registering an email and password. The function attempts to prevent multiple registrations from occurring at the same time by defining a mutex. However, as written, the incorrect variable names are used. The value `this.registering` is set when registration occurs, but the function checks the value `this.registerInProgress`. Therefore, this will not prevent multiple registrations from occurring at the same time.
- SNCrypto implements various cryptographic primitives by wrapping around libraries such as `libsodium`. There are a few functions in SNCrypto that do not catch the generic exceptions thrown by `libsodium`, e.g. `xchacha20Encrypt` and `argon2`. Catching these exceptions and throwing more descriptive errors could improve the overall usability of the codebase. However, be vigilant of error messages for decryption functions as these can introduce padding oracles.

C. CodeQL Analysis

CodeQL is a static analysis tool created by Semmle (now part of GitHub) that runs on code to build call and dataflow graphs, and to make it possible to query a codebase like a database. It allows for queries for several languages, including Javascript, that target common code hygiene and security issues. During the course of the audit, we used CodeQL to help us understand how various pieces of code interact with each other. We also ran the provided Javascript query packs against the `sncrypto` and `snjs` libraries to identify common issues.

CodeQL produced over 20 pages of output, most of which had to do with malformatted, incomplete, or inaccurate docstrings. The output for `sncrypto` was entirely these, so we do not provide them. The output from `snjs` (with the docstring warnings removed for brevity) is in Figure C.1. If Standard Notes is interested in the entire output of CodeQL, we encourage them to run the tool again.

```
"Superfluous trailing arguments","A function is invoked with extra trailing
arguments that are ignored.","warning","Superfluous argument passed to
[[""function
addSeparator""|""relative:///services/component_manager.js:1177:26:1190:5""
]]."","/services/component_manager.js","1221","62","1221","65"
"Useless conditional","If a conditional expression always evaluates to true
or always evaluates to false, this suggests incomplete code or a logic
error.","warning","This use of variable 'interval' always evaluates to
true.","/device_interface.js","29","21","29","28"
"Useless conditional","If a conditional expression always evaluates to true
or always evaluates to false, this suggests incomplete code or a logic
error.","warning","This use of variable 'args' always evaluates to
true.","/services/pure_service.js","45","11","45","14"
"Missing space in string concatenation","Joining constant strings into a
longer string where two words are concatenated without a separating space
usually indicates a text error.","warning","This string appears to be
missing a space after
'Notes,'.","/services/component_manager.js","445","15","445","74"
```

Figure C.1: CodeQL output from running the `javascript-lgtm-full` *tests against* `snjs`.

D. Recommendation for Refactoring Code with TypeScript

In many places, the client maintains a sync state, item state, and session state that are controlled by setting fields on objects. Also, the client may accept input from the server that is not correctly formatted per the specification (i.e., contains extra `JSON` fields in the response). This can lead to bugs where the code is not fully in line with the specification.

To guarantee there are no logic bugs where the client can enter into an invalid state or inadvertently expose data, we recommend a potential refactor of portions of the code with certain TypeScript libraries. These will help fully constrain the client's possible states and transitions between those states.

Maintaining state by conditionally setting fields on objects can be dangerous, because it is common to forget to check or set a relevant field and thus introduce a vulnerability. Relying on the type system for this functionality puts the reponsibility on the compiler to make sure all of the code is correct, and has an added benefit of strictly constraining client behavior.

We recommend considering XState or TypeState to model and implement complex logic, like the authentication process, sync, password updates, login and logout, and transitioning from offline to online. The state diagrams can correspond one-to-one with the specification and ensure that the code is in agreement with it.

We also recommend considering a strongly typed API using Express with TypeScript to ensure that server response fields conform precisely to a specification and handle protocol versioning issues cleanly. This code could also be reused on the server for easier implementation and versioning.

Generally, using more language features and leaning on automatic type-checkers for extra safety make codes more secure, clean, and maintainable.

E. Recommendations for Enforcing Secure Passwords

While reviewing the Standard Notes codebase, Trail of Bits discovered TOB-SNOTES-001, a finding related to allowing insecure passwords when a password is changed. In reviewing this finding with the Standard Notes team, we discussed enforcing strong passwords in the system generally. Currently, when a user registers an email and a password, a minimum of eight characters is required for the password length. However, there are no other restrictions on the password. The Standard Notes team expressed a desire to enforce stronger password requirements in the future. We discussed a few recommendations in this area, and we will detail them further in this appendix.

NIST SP 800-63B provides comprehensive guidance for enforcing secure passwords. For example, restrict the use of sequential or repeating characters (e.g., “aaaaaaaa” or “12345678”), restrict words that are related to the application, and restrict the use of common or previously breached passwords. The publication also suggests the use of a password meter to give users feedback on the strength of their password. Standard Notes could also enforce a minimum password strength according to this meter. NIST SP 800-63B details several other recommendations for increased security and usability, and we encourage Standard Notes to adhere to their guidance when making future design decisions.

We also discussed the use of randomly generated words from a predefined word list with Standard Notes. BIP 39 specifies a protocol for converting a passphrase into a secret. In addition, there are several existing tools, like Diceware, that generate random passphrases for a user. With a large enough word list, using a passphrase with only four or five easily-remembered words can already surpass the entropy of the average password.

F. Recommendation for Guaranteeing Backward Secrecy

The Standard Notes protocol defines a hierarchical key structure. The root key is derived from the account credentials. The `itemsKeys` are generated randomly and are encrypted with the root key. For each item to be encrypted, a random `item_key` is generated and is encrypted with an `itemsKey`. Then the `itemsKeys` and `item_keys` are stored encrypted on the server (along with encrypted files).

If an account's credentials are breached (and thus the root key is breached), the `itemsKeys` can be immediately recovered, and, from there, the `item_keys` and individual items can also be recovered. The Standard Notes protocol provides a mechanism for changing passwords. However, when a password is changed, the same `itemsKeys` are still used by default (unless there was a protocol version upgrade or the user sets an explicit flag).

Backward secrecy is a security property that guarantees that if past credentials are compromised, files created after those credentials were changed will not be at risk. Currently, the protocol does not achieve backward secrecy. If a user's old credentials are breached, a malicious server could still have access to `itemsKeys` encrypted with the old root key. Since the same `itemsKeys` are used by default, the server can use these to decrypt files created after the password change.

We recommend adjusting the protocol to always change `itemsKeys` whenever a password is changed. Since this option is already available (but not by default), it is a minor change. Furthermore, this adjustment will guarantee backward secrecy for the protocol.

G. Fix Log

Standard Note addressed the issues raised in this assessment. Each of the presented fixes were verified by Trail of Bits, as seen below. Standard Notes also addressed the recommendations presented in Appendix D and Appendix F. The reviewed code is available in git revision `7476ec05`.

ID	Title	Severity	Status
01	Small, insecure passwords are allowed when users change passwords	Medium	Fixed
02	Secrets remain in memory for undetermined amount of time	Informational	Risk Accepted
03	Timing information on root key comparison could leak part of root key	Informational	Fixed
04	`Keys.offline.pw` value not cleared in `migrateStorageStructureForMobile`	Informational	Fixed

Detailed Fix Log

This section includes brief descriptions of fixes implemented in Standard Notes and reviewed by Trail of Bits after the end of this assessment.

Finding 1: Small, insecure passwords are allowed when users change passwords
This issue has been resolved. When a password change is requested, the length of the password is verified.

Finding 2: Secrets remain in memory for undetermined amount of time
Risk accepted. Standard Notes found this recommendation to be infeasible for their modern user interface application.

Finding 3: Timing information on root key comparison could leak part of root key
This issue has been resolved. The key comparison now uses a constant-time comparison, which prevents timing information from leaking part of the key.

Finding 4: `Keys.offline.pw` value not cleared in `migrateStorageStructureForMobile`
This issue has been resolved. The keychain values are now cleared when migration is complete in `migrateStorageStructureForMobile`.

Appendix D: Recommendation for Refactoring Code with TypeScript
This recommendation has been implemented; the codebase has been refactored with TypeScript.

Appendix F: Recommendation for Guaranteeing Backward Secrecy
This recommendation has been implemented, and the `itemsKeys` are changed by default whenever the password is changed.