Calif

Shamir39 Audit Report

hacking-team@calif.io | September 27, 2025

Table Of Contents

Executive Summary

Synopsis

From August 2025 to September 2025, Calif was engaged by an unnamed client to audit Shamir39, a tool that combines the usability of BIP39 mnemonics with the threshold security of **Shamir's Secret Sharing** (**SSS**), allowing users to split and recover wallet mnemonic phrases in a human-readable yet cryptographically secure manner.

Key Findings

Calif's audit uncovered seven findings in the Shamir39 implementation:

- One medium-severity vulnerability (Finding 1)
- Six informational issues (Findings 2–7)

Although Finding 1 could theoretically allow an attacker to recover a BIP39 mnemonic with fewer shares than required, it is difficult to exploit in practice because it demands access to a very large number of secret shares. Still, the issue is noteworthy because it undermines the core secrecy guarantee of SSS and should be remediated.

The six informational issues may affect usability and correctness under certain conditions. Of particular note is the potential fallback to JavaScript's weak `Math.random()` random number generator (**RNG**), which could completely destroy the security of Shamir39, if inadvertently used.

Recommendations

Based on our findings, it is recommended that the following measures are taken:

- Replace insecure RNGs with cryptographically secure random number generators (**CSRNGs**) in all contexts.
- Ensure polynomial coefficients in Shamir's Secret Sharing are uniformly sampled from the appropriate field.
- Implement comprehensive parameter validation to safeguard against invalid inputs and future API or configuration changes.

Remediation

Per the client's request, Calif has implemented fixes for Findings 1–6. Finding 7 (use of non-verifiable secret sharing) was excluded from remediation at the client's direction. Full details of the applied fixes are provided in the Remediation section of this report.

Preliminary

This section provides an overview of BIP39, Shamir's Secret Sharing, and Shamir39.

BIP39

BIP39[1] defines a human-readable encoding format for wallet seeds, enabling users to manage cryptographic keys through mnemonic phrases rather than raw binary or hexadecimal data.

Entropy and Checksum

- A source of entropy of length `ENT` bits is selected.
- The entropy is extended with a checksum, computed as the first *k* bits of `SHA256(entropy)`, where `k = ENT / 32`.
- The resulting bitstring has a length of `ENT + k`.

Mnemonic Encoding

- The bitstring is split into **11-bit segments**, each mapping to a word in a standardized **2048-word list**.
- The concatenated words form a **mnemonic secret** (`ms`).

Wordlist Requirements

The ideal wordlist should satisfy the following criteria:

- The first 4 characters must uniquely identify each word.
- Words must avoid similarities (e.g., “build” vs. “built”).
- Lists must be lexicographically sorted.

Standard wordlists for different languages are maintained in the BIP39 repository.

Seed Derivation

The mnemonic secret is converted into a wallet seed using `PBKDF2` with `HMAC-SHA512`:

```
seed = PBKDF2(ms, "mnemonic" || ps, 2048, HMAC-SHA512, 64)
```

The salt includes a user-chosen passphrase (`ps`), defaulting to an empty string if omitted. The derived seed serves as the root for hierarchical deterministic wallets.

[1] https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

Shamir's Secret Sharing

Shamir's Secret Sharing (**SSS**) is a cryptographic scheme that enables secure distribution of a secret among multiple participants.

Goal

For a (t, n) SSS scheme:

- A secret s is divided into s shares.
- Any subset of t shares can reconstruct the secret.
- Knowledge of fewer than t shares reveal no information about s.

Construction

1. Select a finite field or group $\mathbb{G}$ (e.g., $GF(p)$).
2. Construct a random polynomial f of degree $t - 1$:
   $f(x) = s + a_1x + a_2x^2 + \ldots + a_{t-1}x^{t-1}$.
3. Compute shares as points $(x_i, f(x_i))$ where x_i is unique and non-zero.
4. Distribute these shares to participants.

Reconstruction

- Any t shares can reconstruct $f(x)$ via **Lagrange interpolation**.
- Evaluating $f(0)$ yields the original secret s.

Shamir39

Shamir39 combines the mnemonic-based usability of BIP39 with the threshold security of SSS. It enables mnemonic secrets to be split into shares while preserving human readability.

Field Selection

Computations are performed in finite fields $GF(2^b)$ with $2 \leq b \leq 30$. The default field is $GF(2^{11})[X]/(x^{11} + x^2 + 1)$ or short as $GF(2^{11})$.

Share Splitting

A BIP39 secret consists of a 24-word mnemonic phrase. In Shamir39, In Shamir39, this secret is divided using an (m, n) threshold scheme as follows:

- **Word-to-field mapping**: Each mnemonic word is mapped to an element x_i of the finite field $GF(2^{11})$.
- **Secret sharing per word:** For each x_i, SSS is applied to generate n subshares. Any m of these subshares can be combined to recover the original x_i.
- **Share assembly and distribution**: Subshares are grouped by index. The first share contains the first subshare of every x_i, the second share contains the second subshare, and so on.
- **Field-to-word mapping**: Each subshare is converted back into a mnemonic word. Concatenating these words produces n Shamir39 shares, each consisting of 24 words. Any m of these shares can be combined to recover the original BIP39 secret.
- **Final encoding**: Versioning and scheme parameters are prepended to each share, before being distributed to shareholders.

Example $(m = 2, n = 3)$:

- The BIP39 secret is split into 24 integers $x_1, x_2, x_3, \ldots, x_{24}$, each representing one mnemonic word.
- For each x_i, SSS produces three subshares (a_i, b_i, c_i), of which any two can reconstruct x_i.
- These subshares are converted to mnemonic words and assembled into final shares:

```None
First share:  version || params || a1 || a2 || a3 || … || a24
Second share: version || params || b1 || b2 || b3 || … || b24
Third share:  version || params || c1 || c2 || c3 || … || c24
```

Here, `||` denotes string concatenation.

Scope

The audit focuses on a specification that converts BIP39 mnemonics to SSS parts (Shamir39 mnemonics for short) and vice versa, as implemented in the HTML file `standalone.html`.

After inspecting the `standalone.html` file, we found that the file was forked from a publicly available source code, i.e., https://github.com/iancoleman/shamir39 (with some differences that do not affect the overall logic, see Appendix A).

Based on the origin of each JavaScript file, we decided to focus on three JavaScript files in the `src/js` folder that were written by the author of the source Git repository (the remaining files are imported from well-known libraries), namely `shamir39.js`, `jsbip39.js,` and `levenshtein.js`. The target version is `shamir39-p1`.

In these files, we evaluated the correctness and safety of the following public methods:

- `levenshtein.js`:
  - `Levenshtein.get`: Calculate Levenshtein distance of two strings.
  - `Levenshtein.getAsync`: Asynchronously calculate Levenshtein distance of two strings.
- `jsbip39.js`:
  - `Mnemonic.generate`: Generate a BIP39 mnemonic from cryptographically secure random bytes of given strength (bits).
  - `Mnemonic.toMnemonic`: Convert a byte array (Uint8Array) into a BIP39 mnemonic.
  - `Mnemonic.check`: Validate a mnemonic string against the configured wordlist and checksum; returns true/false.
  - `Mnemonic.toSeed`: Derive a seed (hex) from a mnemonic and optional passphrase using PBKDF2-HMAC-SHA512 (2048 rounds); returns hex string.
  - `Mnemonic.splitWords`: Split a mnemonic string into an array of words (splits on whitespace and removes empty entries).
  - `Mnemonic.joinWords`: Join an array of words into a mnemonic string using the correct separator.
  - `Mnemonic.normalizeString`: Normalize a string using `String.prototype.normalize("NFKD")` when available; otherwise returns the input unchanged.
- `shamir39.js`:
  - `Shamir39.split`: Convert a BIP39 mnemonic to Shamir39 mnemonics.
  - `Shamir39.combine`: Convert Shamir39 mnemonics to a BIP39 mnemonic.

Threat Model

Since the cryptographic primitive behind the core functionalities of `standalone.html` is SSS, our audit will focus on ensuring that the following fundamental properties of a t-out-of-n secret sharing scheme hold for this implementation:

- **Recoverability (or Correctness)**: Given any t shares, we can recover secret s.
- **Secrecy (or Security)**: Given any $t' < t$ shares, absolutely nothing is learned about s. In other words, the conditional distribution given the known shares for s should be the a priori distribution for s, so $Pr(s\,|\,t'\ shares) = Pr(s)$.

An adversary is usually more concerned with the second property of an implementation of SSS. In this audit, we will assume that an adversary can access up to $t - 1$ shares and check that the secrecy property still holds.

Note on Verifiability: Our analysis assumes the trusted dealer model of original Shamir's Secret Sharing, meaning the dealer is honest when generating and distributing shares. This is appropriate for Shamir39, where the wallet owner typically acts as the sole dealer. However, because the scheme is non-verifiable, shareholders cannot detect invalid shares during reconstruction. If multi-party use is anticipated, a verifiable secret sharing scheme would be preferable to mitigate this class of attacks (see Findings 7).

Methodology

Manual Analysis

We performed a manual analysis to ensure that the functionalities of the methods listed in Scope matched their intended descriptions. For SSS methods, we verified that both the recoverability and secrecy properties held.

Differential Analysis of the Forked Version

The target of this assessment is the client's customized `standalone.html` file, which was forked from the public repository iancoleman/shamir39. Because any code changes could potentially affect security, it was important to analyze these differences carefully.

We compared the client's fork against the upstream repository and confirmed that the modifications are limited to interface, configuration defaults, and error handling. These changes do not affect the logic of the core cryptographic functionalities (the methods listed in Scope). Based on this conclusion, we did not conduct additional differential testing for logical discrepancies.

A detailed line-by-line comparison of the two versions is provided in Appendix A for completeness.

Findings

This section outlines vulnerabilities and design weaknesses discovered during our audit of the given Shamir39 implementation. Each issue includes a severity rating and rationale, a technical summary, and remediation guidance. Findings are sorted by severity.

ID	Finding	Severity
1	Skewed RNG for Random Coefficient Generation in Shamir39.split, Leading to Secret Recovery With Only t - 1 Shares	**Medium**
2	Lack of Parameter Validation Leading to Unrecoverable Secret	**Informational**
3	Inconsistent Check in Maximum Number of Shares	**Informational**
4	Inconsistent Check in Maximum Required Number of Shares	**Informational**
5	Lack of Mnemonic Words Validation	**Informational**
6	Use of Insecure Random Number Generator	**Informational**
7	Use of Non-Verifiable Secret Sharing	**Informational**

Finding 1: Skewed RNG for Random Coefficient Generation in Shamir39.split, Leading to Secret Recovery With Only t - 1 Shares

Severity	**Medium**
CVSS 3.1	4.4 (AV:N/AC:H/PR:H/UI:N/S:U/C:H/I:N/A:N)

Summary

The `Shamir39.split` method is used to convert BIP39 mnemonic words into Shamir39 shares. More specifically, `Shamir39.split` takes in the following parameters:

- `bip39MnemonicWords`: An array of stripped words of a BIP39 mnemonic
- `wordlist`: The wordlist that is supposed to contain all of the words in `bip39MnemonicWords`
- `m`: The threshold of SSS used to generate Shamir39 shares. This parameter corresponds to the t variable defined in the Impact section.
- `n`: The total number of Shamir39 shares to output

This method first splits the mnemonic into L b-bit integers and then uses SSS to generate Shamir39 shares for each integer (as an element of the Galois extension field $GF(2^b)$). However, the random number generator (RNG) used to generate coefficients of the secret polynomial in SSS is skewed, i.e., the RNG cannot output zero. The relevant code snippet taken from `standalone.html` is shown below.

```javascript
function getRNG(){
        var randomBits, crypto;

        function construct(bits, arr, radix, size){
            var str = '',
                i = 0,
                len = arr.length-1;
            while( i<len || (str.length < bits) ){
                str += padLeft(parseInt(arr[i], radix).toString(2), size);
                i++;
            }
            str = str.substr(-bits);
            // Calif: this condition forbids returning 0
            if( (str.match(/0/g)||[]).length === str.length){ // all zeros?
                return null;
            }else{
                return str;
            }
        }
        ...
        // browsers with window.crypto.getRandomValues()
        if(window['crypto'] && typeof window['crypto']['getRandomValues']
=== 'function' && typeof window['Uint32Array'] === 'function'){
            crypto = window['crypto'];
            return function(bits){
                var elems = Math.ceil(bits/32),
                    str = null,
                    arr = new window['Uint32Array'](elems);

                while( str === null ){
                    crypto['getRandomValues'](arr);
                    str = construct(bits, arr, 10, 32);
                }

                return str;
            }
        }
        ...
    };
```

Impact

Let t denote the threshold of the SSS scheme of `Shamir39.split`. Because the RNG cannot output zero, an attacker who can collect up to $t - 1$ Shamir39 shares of the same BIP39 mnemonic for k times can recover the mnemonic without the final Shamir39 share (as intended for SSS), which means the secrecy property of secret sharing does not hold.

The simplest scenario for this attack is when $t = 2$. In this case, the secret polynomial is of the form $f(x) = c_1 x + s$, in which s denotes the secret element in $GF(2^b)$, c_1 denotes a random element in $GF(2^b)$ generated by the skewed RNG, and x is substituted with the index of a Shamir39 share. Since x and c_1 cannot equal zero, there is an impossible value for $y = f(x)$ and that value is exactly s, i.e., $c_1 x$ cannot evaluate to 0. Hence, if a party receiving one out of two Shamir39 shares of the same mnemonic can still infer the secret s without the other share via elimination of the possible values of y. Through testing, we found that the attacker can determine the secret s this way if they can collect about $25,000$ Shamir39 shares of the same BIP39 mnemonic. In fact, the probability that an attacker can determine an b-bit secret s (all $2^b - 1$ possible values of y are collected and eliminated) given one Shamir39 share for each of k invocations of `Shamir39.split` is:

$$Pr(Rec_{b,k}) = \frac{(2^b-1)!\, S(k, 2^b-1)}{(2^b-1)^k}$$

in which $S(x, y)$ denotes the Stirling number of the second kind, which can be calculated as

$$S(x, y) = \frac{1}{y!}\sum_{i=0}^{y}(-1)^{y-i}\binom{y}{i} i^x = \sum_{i=0}^{y}\frac{(-1)^{y-i} i^x}{(y-i)!\, i!}.$$

Based on the given formula, the probability that s is uniquely identified for $b = 11$ and $k = 25,000$ (the same configuration used in the PoC mentioned below) is approximately 0.9899. In practice, the number of shares needed can be lower, and an attacker can run an offline brute force attack on the candidates for s with a much smaller search space given the public key or wallet address.

For the cases that $t \geq 3$, we can still perform a similar attack, but it requires collecting up to $t - 1$ Shamir39 shares each time the BIP39 mnemonic is split. In other words, $t - 1$ parties collude with each other. In order to convert the attack for $t \geq 3$ into the attack for $t = 2$ described above, we can solve some linear equations to obtain the linear relation $y' = c_1 x' + s$ similar to that of the $t = 2$ case. When $t \geq 3$, the secret polynomial has degree greater than or equal to 2, i.e., $f(x) = c_{t-1}x^{t-1} + \ldots + c_1 x + s$. If we can collect $t - 1$ shares, $t - 1$ linear equations over $GF(2^b)$ (represented by the matrix equation below) can be established to eliminate $t - 2$ random coefficients $c_2, c_3, \ldots, c_{t-1}$ and obtain the linear relation $y' = c_1 x' + s$ in which $x' \neq 0$.

$$v \cdot M = \begin{bmatrix} v_1 & v_2 & \ldots & v_{t-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & x_1^2 & x_1^3 & \ldots & x_1^{t-1} \\ 1 & x_2^2 & x_2^3 & \ldots & x_2^{t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{t-1}^2 & x_{t-1}^3 & \ldots & x_{t-1}^{t-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \ldots & 0 \end{bmatrix}$$

It can be observed that the matrix M by which vector v is multiplied is a Vandermonde matrix (without the column corresponding to the linear term of $f(x)$), which means it is invertible because it is a $(t - 1) \times (t - 1)$ matrix and all x_i $(1 \leq i \leq t - 1)$ are pairwise distinct. After solving the equation above, we can obtain $y' = c_1 x' + s = v_1 y_1 + v_2 y_2 + \ldots + v_{t-1} y_{t-1}$. Similar to the $t = 2$ case, the attacker repeats this process k times and infer s from the k collected y' values. The probability of secret determination for the attack when $t \geq 3$ is the same with that of $t = 2$.

Steps to reproduce

To exploit this issue to recover a BIP39 mnemonic m, the attacker(s) can perform the following steps:

1. Obtain $t - 1$ Shamir39 shares for $t - 1$ distinct indices outputted by `Shamir39.split` given m.

2. If $t \geq 3$, solve the linear equations based on the $t - 1$ shares as described in Impact section above to obtain the y' value as output of this step. Otherwise, output the y value of the Shamir39 share collected in step 1.
3. Eliminate the output of step 2 from the set of candidates for the secret b-bit integer.
4. Repeat steps 1-3 for each of L b-bit integers which m is split into. In the PoC, a BIP39 mnemonic with 256-bit strength is split into 25 11-bit integers by `Shamir39.split`.
5. Repeat steps 4 for k times until $2^b - 1$ distinct values are eliminated for each of L b-bit integers. Output the remaining b-bit value that has not been eliminated for each integer.
6. Combine the L values outputted by step 5 to obtain the BIP39 mnemonic m.

A working PoC that demonstrates this attack in the $t = 2$ case is available in the following Git commit:

- Commit: https://github.com/califio/sss-audit/commit/2ef9ed0b835a901f6b540338bf4bb20ddb3373d2
- Shamir39 version: `shamir39-p1`
- Test case:
  - $t = 2$: Test for skewed RNG with t = 2
  - $t \geq 3$: Test for skewed RNG with t >= 3
- Branch: https://github.com/califio/sss-audit/tree/poc/skewed-rng
- Setup instructions:
  - $t = 2$: https://github.com/califio/sss-audit/blob/poc/skewed-rng/poc/playwright/readme.md
  - $t \geq 3$: https://github.com/califio/sss-audit/blob/poc/skewed-rng/poc/sage/readme.md

Recommendations

The zero element should not be excluded from the set of possible outputs of the RNG used for coefficient generation (the inclusion of zero is also explicitly stated in Adi Shamir's original paper). In fact, zero is an element in $GF(2^b)$ although it does not belong to the multiplicative group of the field. More specifically, the `getRNG` method should be fixed as follows.

```JavaScript
function getRNG(){
        var randomBits, crypto;

        function construct(bits, arr, radix, size){
            var str = '',
                i = 0,
                len = arr.length-1;
            while( i<len || (str.length < bits) ){
                str += padLeft(parseInt(arr[i], radix).toString(2), size);
                i++;
            }
            str = str.substr(-bits);
            // Calif: return the random value even if it's zero
            return str;
        }

        ...
    };
```

Finding 2: Lack of Parameter Validation Leading to Unrecoverable Secret

Severity	**Informational**
CVSS 3.1	N/A

Summary

The `Shamir39.split` method is used to convert BIP39 mnemonic words into n Shamir39 shares, which can be recovered with m shares. However, if $m > n$, the original secret cannot be recovered according to this link. Apparently, there is no check to enforce this, so the generated shares may be insufficient to recover the original mnemonic words.

Steps to reproduce

1. Generate a random 24-word mnemonic sentence `secret`.
2. Call `Shamir39.split` method with `secret` , $m = 3$, $n = 2$ to get `shares`.
3. Call `Shamir39.combine` with `shares` to recover a `secret'` ≠ original `secret`.

A working PoC that demonstrates this issue is available in the following Git commit:

- Commit: https://github.com/califio/sss-audit/commit/9f2fa9f14328f844b0f59cb62e0874e39f230091.
- Test function: `Test_split_does_not_reject_m_>_n`.
- Branch: https://github.com/califio/sss-audit/tree/poc/informational-findings.
- Setup instructions: https://github.com/califio/sss-audit/blob/poc/informational-findings/readme.md

Recommendations

Add a check to verify that $m \leq n$.

Finding 3: Inconsistent Check in Maximum Number of Shares

Severity	**Informational**
CVSS 3.1	N/A

Summary

The `Shamir39.split` method is used to convert BIP39 mnemonic words into n Shamir39 shares and can be recovered with m shares, where each mnemonic word (the secret) is treated as an element of the Galois extension field $GF(2^b)$. By default, the setting uses $GF(2^{11})$ - which has only 2048 elements. This means that n cannot exceed 2047 (excludes the 0 element). One line of code allows $n \geq 2047$, while another enforces that $n \leq 2047$ (`config.max = 2^config.bits - 1 = 2^11 - 1 = 2047`).

Steps to reproduce

1. Generate a random 24-word mnemonic sentence `secret`.
2. Call `Shamir39.split` method with `secret` , $m = 100$, $n = 2048$.

A working PoC that demonstrates this issue is available in the following Git commit:

- Commit: https://github.com/califio/sss-audit/commit/9f2fa9f14328f844b0f59cb62e0874e39f230091.
- Test function: `Test split rejects n > 2047`.
- Branch: https://github.com/califio/sss-audit/tree/poc/informational-findings.
- Setup instructions: https://github.com/califio/sss-audit/blob/poc/informational-findings/readme.md

Recommendations

Change this LoC to reject $n > 2047$.

Finding 4: Inconsistent Check in Maximum Required Number of Shares

Severity	**Informational**
CVSS 3.1	N/A

Summary

The `Shamir39.split` method is used to convert BIP39 mnemonic words into n Shamir39 shares and can be recovered with m shares, where each mnemonic word (the secret) is treated as an element of the Galois extension field $GF(2^b)$. By default, the setting uses $GF(2^{11})$ - which has only 2048 elements. This means that m cannot exceed 2047 (excludes the 0 element). This LoC allows $m \geq 2047$ while this LoC ensures that $m \leq 2047$ (`config.max = 2^config.bits - 1 = 2^11 - 1 = 2047)`.

Steps to reproduce

1. Generate a random 24-word mnemonic sentence `secret`.
2. Call `Shamir39.split` method with `secret` , $m = 2048$, $n = 2$.

A working PoC that demonstrates this issue is available in the following Git commit:

- Commit: https://github.com/califio/sss-audit/commit/9f2fa9f14328f844b0f59cb62e0874e39f230091.
- Test function: `Test split rejects m > 2047`.
- Branch: https://github.com/califio/sss-audit/tree/poc/informational-findings.
- Setup instructions: https://github.com/califio/sss-audit/blob/poc/informational-findings/readme.md

Recommendations

Change this LoC to reject $m > 2047$.

Finding 5: Lack of Mnemonic Words Validation

Severity	Informational
CVSS 3.1	N/A

Summary

According to BIP39 specification, the last 8 bits of the mnemonic words (after converting to bits string) are used as the checksum of previous bits. However, `Shamir39.combine` does not perform the check.

Steps to reproduce

1. Generate a random 24-word mnemonic sentence `secret`.
2. Call `Shamir39.split` method with `secret` , $m = 2, n = 3$ to obtain `shares`.
3. Tweak the last mnemonic words of `shares[0]` to another mnemonic word.
4. Call `Shamir39.combine` with tweaked shares to obtain a secret with an incorrect checksum.

A working PoC that demonstrates this issue is available in the following Git commit:

- Commit: https://github.com/califio/sss-audit/commit/9f2fa9f14328f844b0f59cb62e0874e39f230091
- Test function: `Test_combine_does_not_reject_invalid_checksum`.
- Branch: https://github.com/califio/sss-audit/tree/poc/informational-findings.
- Setup instructions: https://github.com/califio/sss-audit/blob/poc/informational-findings/readme.md

Recommendations

Implement the checksum check for `Shamir39.combine`.

Finding 6: Use of Insecure Random Number Generator

Severity	Informational
CVSS 3.1	N/A

Summary

The `Shamir39.split` method is used to convert BIP39 mnemonic words into Shamir39 shares. This process involves using a random number generator (RNG) to generate random numbers. By default the RNG uses values from cryptographically secure sources: `crypto.randomBytes` in NodeJS or `crypto.getRandomValues` in browsers. However in some specific cases the RNG can fall back to using `Math.random`, which is cryptographically insecure.

JavaScript
```javascript
function getRNG(){
        var randomBits, crypto;

        function construct(bits, arr, radix, size){
            var str = '',
                i = 0,
                len = arr.length-1;
            while( i<len || (str.length < bits) ){
                str += padLeft(parseInt(arr[i], radix).toString(2), size);
                i++;
            }
            str = str.substr(-bits);
            // Calif: this condition forbids returning 0
            if( (str.match(/0/g)||[]).length === str.length){ // all zeros?
                return null;
            }else{
                return str;
            }
        }

       ...
        // Calif: Remove the LoCs below
        // A totally insecure RNG!!! (except in Safari)
        // Will produce a warning every time it is called.
        config.unsafePRNG = true;
        warn();

        var bitsPerNum = 32;
        var max = Math.pow(2,bitsPerNum)-1;
        return function(bits){
            var elems = Math.ceil(bits/bitsPerNum);
            var arr = [], str=null;
            while(str===null){
                for(var i=0; i<elems; i++){
                    arr[i] = Math.floor(Math.random() * max + 1);
                }
                str = construct(bits, arr, 10, bitsPerNum);
            }
            return str;
        };
    };
```

Another issue is that `Math.floor(Math.random() * max + 1)` produces a slightly skewed distribution rather than a uniform one. The problem is with the number zero. Because `Math.random()` returns a double in the range $[0, 1)$, the expression above can evaluate to an integer in the range $[1, max)$ in which $max = 2^{32} - 1$. Additionally, the `bits` parameter of the unsafe PRNG is in the range $[2, 30]$ (inferred from the `primitivePolynomials` array of `defaults`), hence the number of elements in `arr` is always 1. Therefore, when the `bits` LSbs of `arr[0]` are taken as output of the unsafe PRNG, the probability that zero is outputted by the PRNG is roughly $\frac{1}{2^{32-bits}}$ less than that of other values. The closer `bits` is to 32, the more apparent the skewness becomes. An intuition for this issue is that if the `bits` LSbs are equal to zero, then the `32 - bits` MSbs cannot be all zeros while the MSbs can take all possible values if the LSbs correspond to a non-zero integer.

Recommendations

Although the use of `Math.random` will trigger a warning every time it is called, we recommend removing the above piece of code to avoid potential misuses.

Finding 7: Usage of Non-Verifiable Secret Sharing

Severity	**Informational**
CVSS 3.1	N/A

Summary

The current Shamir39 implementation inherits a well-known limitation of original Shamir's Secret Sharing (SSS): it is a non-verifiable scheme. Two weaknesses arise:

- Shareholders cannot independently verify whether the shares distributed by the dealer are valid.
- During secret reconstruction, if a dishonest shareholder submits a fake share and observes the output, they may be able to deduce the original secret.

In the Shamir39 context, the dealer and shareholders are usually the same user (the wallet owner). This mitigates the first issue under the trusted dealer assumption, but the second limitation remains. Without verifiability, a malicious party in a reconstruction process could still attempt to infer the secret by manipulating inputs.

Recommendations

Consider adopting a Verifiable Secret Sharing (VSS) scheme, such as Feldman's VSS, to allow parties to check the validity of distributed shares and to identify dishonest participants during reconstruction. While this is not strictly required under the current threat model, it would strengthen robustness in multi-party settings.

Remediation

Per the client's request, Calif has implemented fixes for Findings 1–6. Finding 7 (use of non-verifiable secret sharing) was excluded from remediation at the client's request.

The fixes were implemented in the pull request https://github.com/iancoleman/shamir39/pull/26.

Document Version History

Release date	Changes
September 5, 2025	Draft version for review.
September 8, 2025	Added Finding 7.
September 12, 2025	Added Remediation section describing the fixes for Finding 1-6.
September 27, 2025	Submitted https://github.com/iancoleman/shamir39/pull/26 to fix Finding 1-6.

Appendix A — Differential Analysis of the Forked Version

This appendix summarizes the differences between the client's customized `standalone.html` and the upstream version from [iancoleman/shamir39](). The diff file between the two versions can be found [here]().

We analyzed each change to assess whether it had a security impact.

Category	Example Change	Security Impact
Word input handling	Added `hydrateWord` and `hydratePhrase` functions; introduced prefix matching.	No impact on cryptographic logic. Affects input flexibility only.
User interface	Simplified interface: removed random generation, language selection, and "More Info" sections. Defaulted to English.	UI changes only. No effect on share generation or recovery.
Default parameters	Changed default from 3-out-of-5 to 2-out-of-3 sharing scheme.	Parameter defaults differ but underlying SSS logic remains unchanged.
Error messages	More descriptive prefix validation error.	Improves usability; no effect on security guarantees.
Removed features	Omitted banners, changelog, offline instructions, and multilingual support.	No effect on security.
Code simplification	Language handling logic simplified; added debug logs for prefix matching.	Cosmetic/logging changes only. No effect on security.