Veridise Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Sismo Commitment Mapper

Veridise Inc.

May 23, 2024

► **Prepared For:**

Sismo
https://www.sismo.io/

► **Prepared By:**

Jon Stephens
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Apr. 11, 2023 Initial Draft

Contents

1 Executive Summary

From Mar. 21, 2023 to Apr. 4, 2023, Sismo engaged Veridise to review the security of their Sismo Commitment Mapper. The review covered the implementation of the Sismo Commitment Mapper service, a web application that allows a user to construct a cryptographic signature that proves their identity. This "proof-of-identity" can then be used with Sismo's ZK circuits, one of which Veridise also covered in a security review in parallel*. Veridise conducted the assessment over 4 person-weeks, with 2 engineers reviewing code over 2 weeks on commit 4ba5c23. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Sismo developers provided the source code of the Sismo Commitment Mapper service for review†, which comprises TypeScript code that is executed on AWS Lambda. To facilitate the Veridise auditors' understanding of the code, the Sismo developers directed the Veridise auditors to publicly available documentation written by Sismo‡. The source code also contained some documentation in the form of READMEs.

The source code contained a test suite, which the Veridise auditors noted provides partial test coverage of the behaviors. In particular, the test suite allows developers to test the service against (1) mocks of third-party services for local debugging; as well as (2) real third-party services used by the Sismo Commitment Mapper. Several files in the source code also indicate that the developers use linting and static analysis tools such as ESLint.

Summary of issues detected. The audit uncovered 8 issues, all of which are assessed to be medium severity or lower by the Veridise auditors. Specifically, a race condition allows the same account to have multiple commitments signed by the Sismo Commitment Mapper (V-SCM-VUL-001), and there are no countermeasures for identity theft scenarios (V-SCM-VUL-002). The Veridise auditors also identified several low-severity issues, including the possibility that social user identifiers that may collide with Ethereum addresses (V-SCM-VUL-004), as well as a number of warnings.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* Veridise auditors also reviewed the ZK circuits implementing Sismo's Hydra-S2 Proving Scheme. See `https://veridise.com/audits`.
† The source code is publicly available at `https://github.com/sismo-core/sismo-commitment-mapper`
‡ `https://docs.sismo.io/sismo-docs/`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Sismo Commitment Mapper	4ba5c23	TypeScript	AWS Lambda

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 21 - Apr. 4, 2023	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	2	1
Low-Severity Issues	3	1
Warning-Severity Issues	3	0
Informational-Severity Issues	0	0
TOTAL	8	2

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Race Condition	1
Theft	1
Access Control	1
Hash Collision	1
Authentication	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Sismo's web application. In our audit, we sought to answer the following questions:

- Can a user construct a proof-of-identity without providing sufficient information?
- Is it possible for an attacker to construct two valid proof-of-identities for the same account?
- Are the authentication flows for the social identity providers implemented correctly?
- Can attackers steal the signing key used by the Sismo Commitment Mapper?
- Is it possible for attackers to craft API queries in a way that allows them to bypass authentication mechanisms?
- Do all API endpoints validate their arguments?
- Does the Sismo Commitment Mapper have any mechanisms to mitigate social engineering attacks?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved extensive manual code review.

Scope. The scope of this audit is limited to the `src` folder of the source code provided by the Sismo developers, which contains the TypeScript implementation of the Sismo Commitment Mapper. While other files were included in the source code, they were not in the scope of the audit. During the audit, the Veridise auditors referred to the excluded files but assumed that they have been implemented correctly.

Methodology. Veridise auditors first inspected the provided tests and read the documentation provided by the Sismo developers. They then began a manual audit of the code assisted by automated testing, and they used a local development environment to experiment with the code and construct proof-of-concept exploits. During the audit, the Veridise auditors regularly met with the Sismo developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue. In particular, we judge the likelihood and impact of a vulnerability as follows in Table 3.2 and Table 3.3, respectively.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SCM-VUL-001	Race Condition Allows Same Account to Have . . .	Medium	Fixed
V-SCM-VUL-002	No Remediation for Identity Theft	Medium	Open
V-SCM-VUL-003	commitment-store-dynamodb IAM Role is . . .	Low	Fixed
V-SCM-VUL-004	Theoretically Possible Hash Collision with Soci. . .	Low	Open
V-SCM-VUL-005	Consider Using force_login Parameter in . . .	Low	Open
V-SCM-VUL-006	commit-ethereum-eddsa Endpoint Accepts. . .	Warning	Open
V-SCM-VUL-007	Commitment Parameter Not Validated	Warning	Open
V-SCM-VUL-008	Call to GitHub OAuth Flow Omits Check for. . .	Warning	Open

4.1 Detailed Description of Issues

4.1.1 V-SCM-VUL-001: Race Condition Allows Same Account to Have Multiple Signed Commitments

Severity	Medium	Commit	4ba5c23
Type	Race Condition	Status	Fixed
File(s)	`commitment-mapper/commitment-mapper.ts`		
Location(s)	commit()		

After one of the "commit" endpoints authenticates the caller, the endpoint will invoke the `CommitmentMapper.commit()` method on to save the caller-provided commitment to the database and return the commitment receipt. This is done as follows:

1. First, the `_isAlreadyUsedForACommitment()` method will be invoked to query the database for any commitment that is already associated with the account. A boolean representing the existence of such a commitment will be stored in `accountAlreadyStore`.
2. If an associated commitment already exists, then the `_getCommitment()` method will be invoked to make a second query will be made to the database. The query is the same as the first one, except this time it will retrieve the actual value of the previously stored commitment. If the stored commitment does not match the user-provided one, then an error will be thrown.
3. Lastly, the `account` and `commitment` will be stored as a row in the database.

Note that this transaction is not atomic, so it may be possible for a malicious caller to bypass the existence check by sending multiple simultaneous requests.

```
async commit(account: string, commitment: string): Promise<any> {
  if (!account || !commitment) {
    throw new Error("account and commitment should always be defined!");
  }

  // verify the account is not already linked to a commitment
  const accountAlreadyStore = await this._isAlreadyUsedForACommitment(
    account
  );
  if (accountAlreadyStore) {
    // retrieve the commitment then verified it has not been modified
    const retrieveCommitment = await this._getCommitment(account);
    if (commitment != retrieveCommitment) {
      throw new Error("Address is already used for a commitment!");
    }
  }
  await this._storeCommitment(account, commitment);

  // send back a commitment receipt linked to this commitment
  return this._constructCommitmentReceipt(account, commitment);
}
```

Snippet 4.1: Definition of `commit()`

Exploit Scenario Suppose a malicious caller does not already have a commitment stored. They send two requests ("Request 1" and "Request 2") to the commitment mapper service at the same time, where endpoint and the `account` is the same but different `commitment` values are used. It is possible for the following situation to occur:

1. `_isAlreadyUsedForACommitment()` evaluates to `false` for Request 1.
2. `_isAlreadyUsedForACommitment()` evaluates to `false` for Request 2.
3. `_storeCommitment()` saves the commitment for Request 1.
4. `_storeCommitment()` saves the commitment for Request 2, overwriting the commitment that was stored for Request 1.
5. Request 1 returns a valid commitment receipt.
6. Request 2 returns a valid commitment receipt.

Note that future requests will only allow the commitment in Request 2 to be used; however, the commitment receipt for Request 1 can still be verified using the public key of the commitment mapper service.

Impact If such a situation occurs, it may be possible for an attacker to produce two valid, different commitment receipts, even though there should only be one commitment associated with each account in the commitment mapper service.

However, note that while this scenario is theoretically possible, it is also highly unlikely. In order for the exploit scenario described above to occur, the attacker must be lucky enough for (1) the requests to be received at roughly the same time; and (2) for the round-trip latency between the AWS Lambda and the Amazon DynamoDB services to be high enough that it is possible for both existence queries to complete before any `PutItem` operation is processed by DynamoDB. An attacker can boost their chances of this situation occurring by employing multiple simultaneous requests.

Recommendation To completely eliminate the possibility of a race condition, all of the database actions should be replaced with a single `PutItem` operation, which is atomic (according to the DynamoDB documentation). This operation can be configured (e.g., with the `ConditionExpression` query parameter) so that it only succeeds if there is no existing entry or if the existing commitment matches the caller-provided commitment.

4.1.2 V-SCM-VUL-002: No Remediation for Identity Theft

Severity	Medium	**Commit**	4ba5c23
Type	Theft	**Status**	Open
File(s)	`handler.ts`		
Location(s)	N/A		

Sismo's commitment mapper ensures that an identity account (i.e. Ethereum address, GitHub account, Twitter account, etc) can only ever be associated with a single signed commitment. This allows a commitment to be used as a proxy for one's identity and allows the returned receipt to act as a proof of ownership. However, such a system does not account for the possibility of theft, as there is no remediation process for users if their identity is stolen. This could occur if a user's identity account were compromised or if a user's secret information were stolen. Since only a single commitment can ever be associated with an address, this would allow an attacker to permanently prove ownership to a stolen identity and in some cases, lock the user with the true identity out of the application.

Impact Consider a scenario where some attacker engages in a phishing campaign. If they successfully convince a user to register an attacker-controlled commitment with the user's GitHub account, Twitter account, or an Ethereum wallet (via a signed message), they could register that account with Sismo, which permanently locks out the true user. As the attacker would not necessarily maintain permanent control over the compromised identity account, the commitment receipt would not necessarily correspond to ownership.

Recommendation Include a mechanism that allows commitments to be revoked or replaced. To do so, Sismo could maintain an incremental merkle tree of valid commitments and publish the root of this merkle tree. When commitments are added, removed or replaced, a new root can be published so that applications can validate a commitment is still active. This would require providing an API for users to request a path to their commitment in the current merkle tree, which could reveal the size of the anonymity set. However, such an API could also be guarded with the current commitment receipt to ensure only the current identity holder can request their merkle tree path.

Developer Response The developer responded along the following lines (paraphrased):

> We are currently working on a way to revoke previous commitments. Both our plan and the one proposed by Veridise have a similar issue where changes to the commitment will impact the nullifiers. This would allow a user to interact with application that builds on top of Sismo, then create a new commitment and interact with the application again while appearing to be a new user.

4.1.3 V-SCM-VUL-003: commitment-store-dynamodb IAM Role is Overly Permissive

Severity	Low	**Commit**	4ba5c23
Type	Access Control	**Status**	Fixed
File(s)	`terraform/commitment-mappter-store/main.tf`		
Location(s)	commitment-store-dynamodb		

The `commitment-store-dynamodb` IAM role is used to limit the scope of access that the `DynamoDBCommitmentStore` class has access to. However, it grants overly broad permissions in two ways:

- A set of `commitment_mapper_accounts` , corresponding to AWS accounts, is allowed to assume the role. For each AWS account, the ARN used for the principal corresponds to the root account.
- The actions include wildcards.

Impact

- Any IAM user in any of the commitment mapper accounts is allowed to assume the role and will therefore be able to perform the DynamoDB actions specified in the policy.
- The `dynamodb:Update*` action allows access to privileged actions such as `dynamodb:UpdateContinuousBackups` and `dynamodb:UpdateTable`. An attacker that gains access to the IAM role will be able to, for example, disable continuous backups and deletion protection on the commitment store table.

Recommendation

- For each commitment mapper account, define an IAM role that is used for DynamoDB access (e.g., such as the lambda's execution role) rather than granting blank access to the root account. This IAM role should be used as the principal in the `data "aws_iam_policy_document" "assume_role"` block.
- Instead of using wildcard actions, the allowed actions should be listed explicitly. See also: https://docs.aws.amazon.com/lambda/latest/operatorguide/wildcard-permissions-iam.html

Developer Response The developers responded:

> Only a single account is given this role and we have strict access controls and logging over every access to that account.

```
data "aws_iam_policy_document" "assume_role" {
  statement {
    actions = ["sts:AssumeRole"]
    principals {
      type = "AWS"
      identifiers = [
        for account_id in local.env.commitment_mapper_accounts : "arn:aws:iam::${
    account_id}:root"
      ]
    }
  }
}

data "aws_iam_policy_document" "dynamodb_access" {
  statement {
    actions = [
      "dynamodb:Get*",
      "dynamodb:PutItem",
      "dynamodb:Update*",
    ]
    effect    = "Allow"
    resources = [aws_dynamodb_table.commitment_store.arn]
  }
}

resource "aws_iam_role" "dynamodb_access" {
  name               = "commitment-store-dynamodb"
  assume_role_policy = data.aws_iam_policy_document.assume_role.json

  inline_policy {
    name   = "dynamodb_access"
    policy = data.aws_iam_policy_document.dynamodb_access.json
  }
}
```

Snippet 4.2: Definition of the `commitment-store-dynamodb` IAM role

4.1.4 V-SCM-VUL-004: Theoretically Possible Hash Collision with Social Identity Providers

Severity	Low	**Commit**	4ba5c23
Type	Hash Collision	**Status**	Open
File(s)	ownership-verifiers, `commitment-mapper/commitment-mapper.ts`		
Location(s)	verify()		

The commitment mapper service provides two different flows that allow users to authenticate using GitHub or Twitter (which we will call "social identity providers", or "social IdP"), respectively. In each of these flows, the service requires the user to obtain an OAuth token from the IdP. Then, the service constructs an Ethereum address by (1) taking the numeric ID of the user from the IdP; (2) left-padding the ID with zeros until it is 20 bytes long (i.e., the size of an Ethereum address); and then setting the top two bytes to a constant that is unique to the IdP (e.g., for GitHub, the top two bytes will be set to `0x1001`, for Twitter it will be `0x1002`, etc.). This address is then passed to the `CommitmentMapper.commit()` function, which will construct the commitment receipt. However, note that this Ethereum address does not correspond to an

```
const { data } = await octokit.request("GET /user");
if (!data.id) {
  throw new Error("Github id not found");
}
const identifier = `0x1001${utils.hexZeroPad(`0x${data.id}`, 20).slice(6)}`;
```

Snippet 4.3: Example of how the fake Ethereum address is constructed in `GithubOwnershipVerifier.verify()`

actual address that the user has control over. For example, the user likely does not have the private keys corresponding to the address. Thus, it is possible for this address to collide with an actual Ethereum address.

Impact A user that uses a social IdP may end up storing a commitment at an address that collides with an actual Ethereum address. In this case, the actual address will not be able to store a commitment, as an address is only allowed to store a commitment once. It is unlikely for a collision to actually occur due to the large space of addresses, but it is theoretically possible.

Recommendation If the developers would like to handle this edge case, the commitment could be generated by hashing a tuple of `(IdP, user id, commitment)`, where `IdP` corresponds to some numeric ID for the IdP (e.g., 0 for Ethereum, 1 for GitHub, etc.) and `user id` is an integer corresponding to some unique identifier for the user (e.g., the Ethereum address, the GitHub user id, etc.). Note that because the user id is at most 20 bytes, the `(IdP, user id)` pair can be packed into a single field element.

Developer Response The developers responded:

> We are going to extend the size of an address so that the first portion of an address will identify the type of account that the address corresponds to.

4.1.5 V-SCM-VUL-005: Consider Using force_login Parameter in Twitter Flow

Severity	Low	**Commit**	4ba5c23
Type	Authentication	**Status**	Open
File(s)	`handler.ts`		
Location(s)	requestTwitterToken()		

The `requestTwitterToken()` endpoint can be used to initiate the Twitter API's three-legged OAuth flow, which will redirect the user to log in with Twitter. If the user is already logged into Twitter, then this will directly take the user to the page that requests the user to grant access to Sismo. A malicious actor with access to the user's phone or browser can take advantage of this fact to provide a commitment, even though they are not actually the Twitter account owner.

To mitigate this type of attack, the `GET oauth/authenticate` endpoint of the Twitter API provides an optional `force_login` parameter. The developers can set `force_login` to true so that the user will be required to provide their Twitter credentials, even if they are already logged in.

4.1.6 V-SCM-VUL-006: commit-ethereum-eddsa Endpoint Accepts Signature From Truncated Address

Severity	Warning	**Commit**	4ba5c23
Type	Data Validation	**Status**	Open
File(s)	`handler.ts`, `ownership-verifiers/ethereum.ts`		
Location(s)	commitEthereumEddsa(), EthereumOwnershipVerifier.verify()		

The `EthereumOwnershipVerifier.verify()` method is used to authenticate callers of the `/commit-ethereum-eddsa` endpoint. This is implemented by checking whether the caller-supplied signature corresponds to the caller-supplied Ethereum address.

Impact One major question arising from this problem is whether the "same" Ethereum address could actually be stored in the database by invoking the `/commit-ethereum-eddsa` endpoint with truncated variants of the address. Thankfully, this is not the case: the signature must correspond to the ownership message with the *truncated* address, and the `recoveredAddress` will always be in lowercase and zero-extended to the full Ethereum address length due to the way `utils.verifyMessage()` works. Thus, the commitment mapper service will consider any truncated address to be equivalent to the full-length version.

Recommendation To guard against potential errors and reduce the attack surface in case the commitment mapper service will be modified in the future, the developers should add validation to the top of `verify()` that checks that the `ethAddress` is a full-length address.

```ts
async verify({
  ethAddress,
  ethSignature,
}: {
  ethAddress: string;
  ethSignature: string;
}): Promise<string> {
  // Verify inputs
  if (!ethAddress || !ethSignature) {
    throw new Error(
      "ethAddress and ethSignature and commitment should always be defined!"
    );
  }

  // verify the ownership of the ethereum account
  const ethAddressLowerCase = ethAddress.toLowerCase();

  // Verify the Eth account is well possessed by the caller of this function
  // by verifing the signature corresponds to the address.
  const recoveredAddress = utils.verifyMessage(
    getOwnershipMsg(ethAddressLowerCase),
    ethSignature
  );
  if (!recoveredAddress) {
    throw new Error("Signature not valid!");
  }
  if (ethAddressLowerCase.toLowerCase() != recoveredAddress.toLowerCase()) {
    throw new Error(
      `Address ${ethAddressLowerCase} does not corresponds to signature!`
    );
  }

  return ethAddressLowerCase;
}
```

Snippet 4.4: Definition of `verify()`

4.1.7 V-SCM-VUL-007: Commitment Parameter Not Validated

Severity	Warning	**Commit**	4ba5c23
Type	Data Validation	**Status**	Open
File(s)	`handler.ts`		
Location(s)	all "commit" endpoints		

The caller of the `/commit*` endpoints provide the commitment, which is a Poseidon hash. To construct a commitment receipt, the `_constructCommitmentReceipt()` method takes the Poseidon hash of the user identifier (`ethAddressBigNumber` in this case) and the user-provided `commitment`. Based on the way that `poseidon` is called inside `_constructCommitmentReceipt()`, it appears that

```
const poseidon = await buildPoseidon();
const ethAddressBigNumber = BigNumber.from(ethAddress.toLowerCase()).mod(
  SNARK_FIELD
);
const msg = poseidon([ethAddressBigNumber, commitment]);
// sign the receipt => this is the commitmentReceipt
const commitmentReceipt = await eddsaAccount.sign(msg);
```

Snippet 4.5: Lines in `CommitmentMapperEddsa._constructCommitmentReceipt()` where the receipt is created.

`commitment` is assumed to be a string representation of a single finite field element. However, along every path that call `_constructCommitmentReceipt()`, there is no logic that validates that commitment indeed corresponds to a valid finite field element.

Impact

- If the `commitment` is decoded to an integer that is at least the field size, then the `poseidon()` call will interpret the commitment parameter as `commitment` modulo the field size. This is likely to happen if the caller mistakenly provides a hex string that has too many digits, in which case they will get a commitment receipt that does not match the actual commitment they wanted to submit.
- If the `commitment` cannot be interpreted as a number (e.g., it is an arbitrary string), then `poseidon` will throw an error. In this case, the invalid commitment will have been stored in the database already, preventing the user from retrying with a correct commitment.

Recommendation The developers should insert checks that validate that the `commitment` argument is a valid finite field element before any user authentication is performed.

4.1.8 V-SCM-VUL-008: Call to GitHub OAuth Flow Omits Check for State Parameter

Severity	Warning	Commit	4ba5c23
Type	Data Validation	Status	Open
File(s)	ownership-verifiers/github.ts		
Location(s)	verify()		

When a user wants to use a GitHub identity with the commitment mapper service, they must undergo GitHub's OAuth web application flow. After a user logs in to GitHub, GitHub will redirect users back to a Sismo-controlled web application. Then the user will call the `/commit-github-eddsa` endpoint with their OAuth credentials and their commitment. In order to validate the OAuth credentials, the commitment mapper service will invoke a `POST` request to `https://github.com/login/oauth/access_token`:

```
const accessResponse = await axios({
  method: "post",
  url: `https://github.com/login/oauth/access_token?client_id=${clientID}&
    client_secret=${clientSecret}&code=${code}`,
  headers: {
    accept: "application/json",
  },
});

const accessToken = accessResponse.data.access_token;
const octokit = new Octokit({ auth: accessToken });
```

This code omits one part of the flow that is mentioned by the GitHub documentation:

> If the user accepts your request, GitHub redirects back to your site with a temporary `code` in a code parameter as well as the state you provided in the previous step in a `state` parameter. The temporary code will expire after 10 minutes. If the states don't match, then a third party created the request, and you should abort the process. (Source)

There is no code in the `GithubOwnershipVerifier` that validates the `state` parameter, which is described as follows:

> An unguessable random string. It is used to protect against cross-site request forgery attacks.

Recommendation If the developers are using cookie-based authentication for their web application frontend, the developers can reduce the risk of a CSRF attack occurring if they use the `state` parameter, as suggested by the GitHub documentation.