Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

VENTURE 23

Verulink - Coinbase Quest

Veridise Inc.

October 14, 2024

► **Prepared For:**

Venture23 Inc.
https://venture23.xyz/

► **Prepared By:**

Mark Anthony
Nicholas Brown

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 06, 2024 V1
Oct. 14, 2024 V2

Contents

1 Executive Summary

From Aug. 26, 2024 to Sept. 2, 2024, Venture23 Inc. engaged Veridise to review the security of their Verulink - Coinbase Quest protocol. The review covered an Aleo program that allows Coinbase authenticated users to claim rewards and a backend to facilitate interaction with the Coinbase APIs. Veridise conducted the assessment over 12 person-days, with 2 engineers reviewing code over 6 days on commit `440b876d`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual code review.

Code assessment. The Verulink - Coinbase Quest developers provided the source code of the Verulink - Coinbase Quest contracts for review. The source code appears to be mostly original code written by the Verulink - Coinbase Quest developers. It contains some documentation in the form of documentation comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the Verulink - Coinbase Quest developers met with the Veridise auditors to discuss the expected behavior of the protocol and shared some additional relevant documentation.

The Aleo portion of the code contained a test suite, which the Veridise auditors noted covered some success and failure conditions when calling the protocol. The backend portion of the code did not contain any tests.

Summary of issues detected. The audit uncovered 14 issues, 4 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-VCQ-VUL-001 highlights how a user's reward claim can be frontrun by an attacker to claim rewards, V-VCQ-VUL-002 details how the current reward claim process will always fail because the Coinbase user ID cannot be converted to a valid Aleo field element, V-VCQ-VUL-003 depicts how an incorrect threshold setting can cause the protocol to become permanently stuck, and V-VCQ-VUL-004 shows how potential hash collisions in the proposal hashes can be exploited.

The Veridise auditors also identified 5 medium-severity issues, including V-VCQ-VUL-007 which lists inconsistencies observed with the Coinbase Quest API specifications, V-VCQ-VUL-006 which illustrates how proposals not having an execution deadline can cause issues, as well as 3 warnings, and 2 informational findings.

Of the 14 acknowledged issues, Venture23 Inc. has fixed 11 issues. This includes all 2 high-severity issues and 2 critical issues. Venture23 Inc. does not plan to fix the other 3 acknowledged issues at this time.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Verulink - Coinbase Quest.

The auditors recommend adding unit tests and integration tests for the backend, which test each component involved in the reward claim process thoroughly to avoid issues like V-VCQ-VUL-002

and V-VCQ-VUL-012. Additionally, the auditors recommend strictly following Coinbase API specs. Current deviations include using an incorrect activity name for generating user payloads and not using a randomly generated nonce field in the JWT token header.

The project makes use of multiple private keys, which are currently stored in environment variables or read from a JSON file. The auditors suggest adding secure private key management to mitigate the risk of them being stolen.

This audit identified several substantial issues and we would recommend further testing. In particular, we strongly recommend adding negative tests to ensure certain bad scenarios are not possible.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Verulink - Coinbase Quest	`440b876d`	Leo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 26–Sept. 2, 2024	Manual & Tools	2	12 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	2	2	2
High-Severity Issues	2	2	2
Medium-Severity Issues	5	5	4
Low-Severity Issues	0	0	0
Warning-Severity Issues	3	3	2
Informational-Severity Issues	2	2	1
TOTAL	14	14	11

Table 2.4: Category Breakdown.

Name	Number
Data Validation	5
Maintainability	3
Denial of Service	2
Frontrunning	1
Authorization	1
Logic Error	1
Access Control	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Verulink - Coinbase Quest's Aleo programs and backend implementation. In our audit, we sought to answer questions such as:

- Is it possible for a malicious user to steal funds from the Aleo programs?
- Is it possible for a malicious user to perform a denial of service attack on the Aleo programs?
- Can members or non-members of the council manipulate proposals in an unfair way?
- Are inputs to the Aleo programs properly validated?
- Does the backend interact properly with the Coinbase APIs?
- Are user inputs to the backend properly validated?
- Can unauthorized users claim rewards?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a manual analysis by human experts.

Scope. The scope of this audit is limited to the `aleo/programs`, and the `backend/src` folders of the source code provided by the Verulink - Coinbase Quest developers, which contains the Aleo program and Typescript backend implementations of the Verulink - Coinbase Quest.

Methodology. Veridise auditors inspected the provided tests, and read the Verulink - Coinbase Quest documentation. They then began a manual review of the code. During the audit, the Veridise auditors regularly met with the Verulink - Coinbase Quest 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.

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

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

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

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

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., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-VCQ-VUL-001	The finalize_claim can be Frontrun	Critical	Fixed
V-VCQ-VUL-002	Coinbase user IDs are larger than the . . .	Critical	Fixed
V-VCQ-VUL-003	Missing Threshold check in add_member	High	Fixed
V-VCQ-VUL-004	Potential proposal hash collisions	High	Fixed
V-VCQ-VUL-005	Quest Initialization missing access controls	Medium	Fixed
V-VCQ-VUL-006	Proposals have no deadline	Medium	Acknowledged
V-VCQ-VUL-007	Inconsistencies with Coinbase API specs	Medium	Fixed
V-VCQ-VUL-008	Missing input validation in claimSignature()	Medium	Fixed
V-VCQ-VUL-009	ConvertFieldTypeToUserId Used improperly	Medium	Fixed
V-VCQ-VUL-010	Improper interaction with the Coinbase APIs	Warning	Fixed
V-VCQ-VUL-011	Claim() accepts any signature from the signer	Warning	Acknowledged
V-VCQ-VUL-012	Missing Await When Using Result of . . .	Warning	Fixed
V-VCQ-VUL-013	Centralization Risk	Info	Acknowledged
V-VCQ-VUL-014	Typos, Minor Errors, and Missing . . .	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-VCQ-VUL-001: The finalize_claim can be Frontrun

Severity	Critical	**Commit**	440b876
Type	Frontrunning	**Status**	Fixed
File(s)	quest.leo		
Location(s)	claim()		
Confirmed Fix At	81489e1		

The `claim()` function in `quest.aleo` allows a user to pass a `uuid` along with a `signature` and a `signer` . The `signature` is generated by the backend for a given `uuid` which corresponds to a Coinbase user. The signature is validated, then passed to the finalize function which will mark the `uuid` as claimed and send the reward to the caller. However, all three inputs to `claim()` are public. This means that a malicious user can see the value of the signature and could frontrun the finalize part of the claim function.

```
async transition claim(public uuid: field, public sig: signature, public signer:
    address) -> Future{
  // Mint the tokens publicly by invoking the computation on-chain.
  assert(verify_signature(uuid, sig, signer));
  let f: Future = multi_token_support_program.aleo/transfer_public(vusdc_id, self.
    caller, amount);
  return finalize_claim(f, uuid, self.caller, signer);
}

async function finalize_claim(f:Future, public uuid: field, public recipient:address
    , public signer: address) {
  f.await();

  assert_eq(status.get(true), UNPAUSED_STATUS);

  assert(is_claimed.contains(uuid).not());
  // set UUID to claimed
  is_claimed.set(uuid, true);

  let stored_signer: address = auth_accounts.get(false);
  assert_eq(stored_signer, signer);

  account.set(uuid, recipient);

  let total_claims_count: u64 = total_claims.get(true);
  total_claims.set(true, total_claims_count+1u64);
}
```

Snippet 4.1: Definition of `claim()`

Impact Since the `uuid` is not tied to an Aleo account, if the claim is frontrun, the attacker will receive the rewards instead of the legitimate user. This will prevent the user from ever receiving rewards for their Coinbase `uuid` because it will have already been claimed.

Recommendation Implement one of the following:

- Make the signature a private input
- Include the Aleo recipient address in the signature to ensure that only the intended recipient can claim the rewards for the given `uuid`

Developer Response The developers have made the signature a private input.

4.1.2 V-VCQ-VUL-002: Coinbase user IDs are larger than the largest Aleo field element

Severity	Critical	Commit	440b876
Type	Data Validation	Status	Fixed
File(s)	users.service.ts		
Location(s)	convertUserIdToFieldType()		
Confirmed Fix At	bb94697		

The method `claimSignature` takes as input a string `userId` which corresponds to a Coinbase user ID. This `userId` is converted to an Aleo field type and a signature is created by signing it, which can then be used to claim rewards from the Aleo Quest program. The conversion from the string `userId` to an Aleo field element is performed through `convertUserIdToFieldType()`. See snippet below for the implementation.

Here, the output of `convertUserIdToFieldType()` for a `userId` is assumed to fit inside an Aleo field element. However, that is not the case. We can verify this with an example.

Let's take a sample `userId` from the Quest API docs: `ztTeMxhzYMCgFpxLf41aWvpVB4exFAP_FUZTmXD_Nt7NZ3siwvISTjKxtmJazEgrpljqewek`. The output of `convertUserIdToFieldType()` for this input `userId` is `8960554293856004467198740364370623737579809247134810895954261437967321907557115922972777546839551366613397091142013586820761847716field`. This value is much much larger than the largest Aleo field element which is `8444461749428370424248824938781546531375899335154063827935233455917409239040field`.

If the string above is passed as an input to the Aleo Quest program, it will be rejected as invalid input. And users will not be able to claim rewards, because the calls to the Aleo Quest program will revert.

```
export const convertUserIdToFieldType = (userId: string): string => {
  // Replace URL-safe characters with standard Base64 characters
  const base64 = userId.replace(/-/g, '+').replace(/_/g, '/');

  // Decode the Base64 string to a binary string
  const binaryString = Buffer.from(base64, 'base64');

  // Decode the binary string to a byte array
  const byteArray = Uint8Array.from(binaryString);

  // Convert byte array to big integer
  let bigInt = BigInt(0);
  for (let i = 0; i < byteArray.length; i++) {
    bigInt = (bigInt << BigInt(8)) + BigInt(byteArray[i]);
  }

  return bigInt.toString() + 'field';
};
```

Snippet 4.2: Definition of `convertUserIdToFieldType()`

Impact Users will not be able to claim rewards, because the output of `convertUserIdToFieldType()` will not be accepted as a valid field input for the Aleo Quest program.

Recommendation Instead of using the Coinbase user IDs directly, use a cryptographic hash of the user ID as a unique identifier for on-chain reward claims. The hash should be computed using a library that produces a hash as a finite field to ensure that the hash output can fit inside an Aleo field element.

Additionally the `convertUserIdToFieldType()` and `convertFieldTypeToUserId()` should be removed in favor of using a finite field library.

4.1.3 V-VCQ-VUL-003: Missing Threshold check in add_member

Severity	High	Commit	440b876
Type	Data Validation	Status	Fixed
File(s)	council.leo		
Location(s)	add_member()		
Confirmed Fix At	3df46ae		

The `council.aleo` program allows members to vote on proposals, which will pass if the number of votes exceeds a specified threshold. Due to limitations of Aleo, arrays must have a fixed size. Since the `council.aleo` program uses an array to handle the votes for a proposal, any votes in excess of the size of the array cannot be processed by the proposal handling functions. Thus if the threshold is higher than the max size of the vote array (defined as `SUPPORTED_THRESHOLD` in the program), it will be impossible for any future proposal to pass.

The `add_member` function allows for increasing the number of members of the council and updating the threshold accordingly. This will need to be approved by the council for the member to be added. When such a proposal is executed, the new threshold is checked to be less than or equal to the new council size. However, this function can allow the council size to increase above the `SUPPORTED_THRESHOLD` amount, which could allow the new threshold to be greater than the `SUPPORTED_THRESHOLD`. Consider the following example:

Start:

- `SUPPORTED_THRESHOLD = 5`
- `current_members_count = 5`

Calling `add_member()` to add new member and specify new threshold to be 6:

- The assertion that the `new_threshold <= current_members_count + 1` succeeds.
- The new threshold is set to 6

After:

- The current threshold is 6 which exceeds the `SUPPORTED_THRESHOLD`, so no future proposals can pass

Impact This proposal will have to be approved by the council, but if it gets approved the entire protocol will be broken and cannot be recovered because changing the threshold requires a proposal to pass. This could occur by accident, or if members of the council are unaware of the `SUPPORTED_THRESHOLD` restriction.

Recommendation Add an additional check to `finalize_add_member()`, `finalize_update_threshold` , and `finalize_remove_member` that the `new_threshold` does not exceed the `SUPPORTED_THRESHOLD` .

Additionally, consider adding a max size for the council, or a restriction on the ratio of the threshold to the council size to ensure that the threshold doesn't become too small relative to the size of the council.

```
async transition add_member(public id: u32, public new_member: address, public
    new_threshold: u8, public voters: [address; 5]) -> Future{
    assert(new_threshold > 0u8);
    let proposal: AddMember = AddMember {
        id,
        new_member,
        new_threshold
    };
    let proposal_hash: field = BHP256::hash_to_field(proposal);

    let votes: u8 = get_valid_unique_address_count(voters);
    let vote_keys: [field; 5] = get_proposal_vote_keys(proposal_hash, voters);

    return finalize_add_member(proposal_hash, new_member, new_threshold, voters,
    vote_keys, votes);
}

async function finalize_add_member(proposal_hash: field, new_member: address,
    new_threshold: u8, voters: [address; 5], vote_keys: [field; 5], votes: u8) {
    for i: u8 in 0u8..SUPPORTED_THRESHOLD {
        assert(Mapping::contains(members, voters[i]));
        assert(Mapping::get(proposal_votes, vote_keys[i]));
    }

    // Get the threshold
    let threshold: u8 = Mapping::get(settings, THRESHOLD_INDEX);

    assert(votes >= threshold);

    // Ensure that the proposal has not been executed
    assert(!Mapping::contains(proposal_executed, proposal_hash));

    // Mark the proposal as executed
    Mapping::set(proposal_executed, proposal_hash, true);

    // Execute the proposal
    assert(!Mapping::contains(members, new_member));
    Mapping::set(members, new_member, true);
    Mapping::set(settings, THRESHOLD_INDEX, new_threshold);

    // Update total members
    let current_members_count: u8 = Mapping::get(settings, TOTAL_MEMBERS_INDEX);
    assert(new_threshold <= current_members_count + 1u8);
    Mapping::set(settings, TOTAL_MEMBERS_INDEX, current_members_count + 1u8);
}
```

Snippet 4.3: Definition of `add_member()`

Developer Response The developers have added the recommended checks that the `new_threshold` doesn't exceed the `SUPPORTED_THRESHOLD` .

4.1.4 V-VCQ-VUL-004: Potential proposal hash collisions

Severity	High	**Commit**	440b876
Type	Data Validation	**Status**	Fixed
File(s)	`council.leo`		
Location(s)	See issue description		
Confirmed Fix At	`10d4092`		

When executing a proposal, `BHP256::hash_to_field` is called on the proposal struct to compare with `proposal_hash` that has been approved. This function considers the names of struct fields in the hash, but doesn't consider the name of the struct. This can lead to collisions between `PauseProgram` and `UnpauseProgram` since all their fields match. This means that the protocol can be paused when a proposal is approved to unpause the program if the approved `proposal_hash` and `id` are passed to `pause()` instead of `unpause()` . Since anyone can call `pause()` or `unpause()` a malicious user can prevent the protocol from being paused/unpaused by calling the wrong function when a proposal to pause or unpause is approved.

```
struct PauseProgram{
  id: u32,
}
struct UnpauseProgram{
  id: u32,
}
```

Snippet 4.4: Definitions of `PauseProgram` and `UnpauseProgram`

Impact Any malicious user can perform a DoS attack on the protocol when a proposal is accepted to pause or unpause the program. They would just need to quickly call `pause()` when such a proposal is approved and continue to call `pause()` when any future unpause proposal is approved. This could allow anyone to indefinitely stop the protocol from executing (as long as they are quick enough).

Recommendation Add a type field to all proposal structs that indicates the type of the proposal. This will prevent any hash collisions between proposal types.

Developer Response We have fixed those structs with different members inside of them to be able to distinguish between them.

4.1.5 V-VCQ-VUL-005: Quest Initialization missing access controls

Severity	Medium	**Commit**	440b876
Type	Authorization	**Status**	Fixed
File(s)	`quest.leo`		
Location(s)	initialize()		
Confirmed Fix At	`fdcea0f`		

The `initialize()` function in the `quest.aleo` program is used to set the address of the backend implementation and the address of the `admin` account. This function doesn't implement any checks on the caller or the value of the parameters, so anyone (if they are fast enough to call it first) could call initialize after the program is deployed.

```
async transition initialize(signer: address, admin:address)-> Future{
  return finalize_initialize(signer, admin);
}
async function finalize_initialize(signer:address, admin: address){
  // Mapping::set(settings, 0u8, token_id);

  // Assert bridge has not been initialized before
  assert(auth_accounts.contains(false).not());
  auth_accounts.set(false, signer);
  auth_accounts.set(true, admin);
  total_claims.set(true, 0u64);
}
```

Snippet 4.5: Definition of `initialize()`

Impact A malicious party can take over the quest program by calling `initialize()` with their accounts. This would require the program to be redeployed.

If the malicious party keeps calling `initialize()` on new deployments, this could cause delays and prevent the protocol from being released.

If funds are sent to a compromised instance of the program they will be lost because the admin can pause all claims.

Recommendation Similar projects have a hardcoded constant for the initial admin address (or an initializer address) and check that the caller of the initialize function is the specified address. This will ensure that only a trusted address can initialize the protocol.

4.1.6 V-VCQ-VUL-006: Proposals have no deadline

Severity	Medium	**Commit**	440b876
Type	Maintainability	**Status**	Acknowledged
File(s)	`council.leo`		
Location(s)	See description		
Confirmed Fix At	N/A		

In the council program, members can make new proposals which can then be executed if they get more than the threshold of votes required. These proposals include actions which can make changes to the council members like addition and removal. Upon additional and removal actions the threshold of votes required can also be changed to be in line with the updated council size.

These proposals do not have a deadline until which they are active. This can cause issues in the case where an older proposal might have an existing vote count which is close to the threshold and removing a member might put it across the threshold. This would allow execution of the proposal while it may not be the actual intention of the council.

Proposals should have a deadline after which they cannot be executed. In Leo this can be achieved by using block heights to ensure proposals are only active until a certain block height.

Impact Older proposals can be unknowingly executed if proposal actions put their votes over the threshold.

Recommendation Proposals should only be active until a certain block height, after which they can not be executed.

Developer Response Since the quest will run for relatively shorter time, we have decided not to include the deadline in the proposals but handle the cases you have mentioned very carefully.

4.1.7 V-VCQ-VUL-007: Inconsistencies with Coinbase API specs

Severity	Medium	**Commit**	440b876
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	`8052f77`		

The backend component of the Verulink - Coinbase Quest protocol interacts with Coinbase APIs to validate `userIds` before reward claims and to report the claimed `userIds` to Coinbase to validate quest completion.

The current implementation of the backend has some inconsistencies with the directions shared in the Coinbase API-Quest documentation. These could lead to improper usage of the APIs and the protocol not functioning as intended.

`coinbase.service.ts`

- In `generateUserPayload` , the `activityName` is defined as `aleo_bridge_xyz`, whereas for Venture23 the activity name listed in the document is `aleo_claim`.
- As per the API docs, it is recommended to report user activity in batches with a maximum batch size of `1000`. These user activities are reported through `reportUserActivity`, but while constructing the payload it is not verified that the batch size is less than the maximum.

`jwtToken.service.ts`

- The method `getJWT` returns a JWT token which is to be included in the headers section of the data payload when interacting with the Coinbase API. The token object is created by signing a number of fields along with a header object and algorithm field. As per the directions in the API docs the, `nonce` field in the header object should be random for each request and no longer than 19 chars but in the current implementation it is assigned the value of the current timestamp (in seconds). The `nonce` should be generated using `crypto.randomBytes(8).toString('hex')` as recommended.

Impact The interactions with the Coinbase APIs might fail or update incorrect information if the directions mentioned are not followed. For example, while reporting user activities the reports will be made for activity completion of `aleo_bridge_xyz`, instead of `aleo_claim`.

Recommendation Follow the recommended directions mentioned in the Coinbase Quest docs.

4.1.8 V-VCQ-VUL-008: Missing input validation in claimSignature()

Severity	Medium	**Commit**	440b876
Type	Data Validation	**Status**	Fixed
File(s)	`users.service.ts`		
Location(s)	claimSignature()		
Confirmed Fix At	`dc3ba50`		

The method `claimSignature` accepts a `userId` as a string. If this `userId` has not been claimed in the Aleo contract, then it validates that the `userId` is a valid Coinbase user ID by interacting with the Coinbase API through `validateUser`. See snippet below for context.

The main issue here is that the input `userId` to `claimSignature` has no data validation performed on it. If it contains extra characters, these might be ignored by Coinbase as part of data sanitization on its API requests. The `userId` may be trimmed down to a valid user ID when the extra characters are ignored on the Coinbase side, but for the backend it will be treated as the raw string, and therefore many strings may exist on the backend which correspond to the same valid Coinbase user ID. An attacker could exploit this by padding the same `userId` in different ways to claim rewards multiple times.

```
1  public async claimSignature(userId: string): Promise<Signature> {
2    const parsedId = convertUserIdToFieldType(userId);
3    const isClaimed = this.aleoContract.getClaimStatus(parsedId);
4
5    if (isClaimed) {
6      throw new HttpException(403, 'User has already claimed the quest');
7    }
8
9    const userData = await this.userExists(userId);
10
11   if (!userData?.valid) throw new HttpException(400, 'Invalid user');
12
13   try {
14     await this.userClaim.create({
15       userId,
16     });
17
18     return this.signatureService.sign(userId);
19   } catch (e) {
20     logger.error(e);
21     throw new HttpException(400, 'Failed to generate signature');
22   }
23 }
```

Snippet 4.6: Definition of `claimSignature()`

Impact As part of sanitizing its API requests, Coinbase may ignore characters from the `userId` character sequence (escape characters can be ignored, for example). This can be exploited by an attacker by adding extra characters to a `userId` which will be trimmed down to a valid `userId`. Because the `claimSignature` method does not validate the input `userId` length, these padded `userId` strings will be considered as unique entities and will allow a user to claim rewards multiple times for the same corresponding Coinbase user ID.

Recommendation Perform input validation on the `userId` to ensure that the above attack vector is not possible. One way to do it would be to validate the length of the `userId` string, to verify that it meets expectations.

4.1.9 V-VCQ-VUL-009: ConvertFieldTypeToUserId Used improperly

Severity	Medium	**Commit**	440b876
Type	Denial of Service	**Status**	Fixed
File(s)	`src/services/users.service.ts`		
Location(s)	see issue description		
Confirmed Fix At	`bb94697`		

The `convertFieldTypeToUserId()` is used on each of the inputs to `UserService.bulkClaimSuccess()`. However, `bulkClaimSuccess()` is called with Coinbase `userIds`, not finite field values. This will cause `convertFieldTypeToUserId()` to crash when attempting to convert the `userId` value into a `BigInt` because it isn't a numerical value.

```
1  public async bulkClaimSuccess(userIds: string[]): Promise<void> {
2    const bulkOperations = [];
3
4    // Convert aleo based user identifiers(field values) to coinbase user identifiers
5    const parsedUserIds = userIds.map(id => convertFieldTypeToUserId(id));
6
7    // Find users data in app database
8    const usersData = await this.userClaim.find({ userId: { $in: parsedUserIds } });
```

Snippet 4.7: Snippet from `UserService.bulkClaimSuccess()`

```
1  public async bulkClaimSuccess(userIds: string[]): Promise<void> {
2    const bulkOperations = [];
3
4    // Convert aleo based user identifiers(field values) to coinbase user identifiers
5    const parsedUserIds = userIds.map(id => convertFieldTypeToUserId(id));
6
7    // Find users data in app database
8    const usersData = await this.userClaim.find({ userId: { $in: parsedUserIds } });
```

Snippet 4.8: Definition of `AleoPolling.reportClaims()` which calls `UserService.bulkClaimSuccess()`

Impact Calls to `AleoPolling.reportClaims()` will always throw an error. This means that the claims won't be reported to Coinbase and the data won't get updated with claim data.

This can be fully resolved by fixing the backend implementation.

Recommendation Remove `convertFieldTypeToUserId` because it shouldn't be used in this function. The only other place in which it is called is in `UserService.claimSuccess()` which is never used. Additionally if the recommendation in V-VCQ-VUL-002 is applied, it won't be possible to get the original `userId` from the field value since the `userId` is hashed.

4.1.10 V-VCQ-VUL-010: Improper interaction with the Coinbase APIs

Severity	Warning	**Commit**	440b876
Type	Data Validation	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`8ee5039`		

There are several issues with how the backend interacts with the Coinbase APIs in `coinbase.service.ts`:

- In `sendPayload()` the request is assumed to succeed and doesn't have any error handling.
  - There should be error handling if a network error causes the request to fail, or if Coinbase returns an error.
- The Coinbase documentation documents status codes for various results of `ReportUserActivities`, but no status codes are checked for any request made by the backend.
- The results of the Coinbase requests aren't validated to ensure that data of the expected type is present in each of the resulting fields.
  - This could result in crashes since the value of the resulting fields is assumed to be valid.

Impact These issues could result in crashes and downtime for the backend.

Recommendation Implement request and response validation as suggested above to ensure all data received and sent is properly sanitized. Also implement error handling to account for failing API calls or network errors.

4.1.11 V-VCQ-VUL-011: Claim() accepts any signature from the signer

Severity	Warning	Commit	440b876
Type	Logic Error	Status	Acknowledged
File(s)	quest.leo		
Location(s)	claim()		
Confirmed Fix At	N/A		

The transition `claim` accepts a signature along with a field `uuid`. It then verifies that the signature was signed by the address `signer` with respect to the field `uuid`. In the backend, the `ALEO_SIGNER_KEY` corresponds to the address `signer`. See snippet below for the implementation.

There is no verification done in the `claim` transition to ensure that the field `uuid` is a valid Coinbase userID. Therefore any data ever signed by the `ALEO_SIGNER_KEY` can be passed to `claim` along with the signature, to pass the signature verification and claim rewards.

For instance, if the `ALEO_SIGNER_KEY` is not a newly generated private key, and has been used to sign stuff before then it would allow someone privy to that knowledge to use those signatures to claim rewards.

```
1 async transition claim(public uuid: field, public sig: signature, public signer:
      address) -> Future{
2   // Mint the tokens publicly by invoking the computation on-chain.
3   assert(verify_signature(uuid, sig, signer));
4   let f: Future = multi_token_support_program.aleo/transfer_public(vusdc_id, self.
      caller, amount);
5   return finalize_claim(f, uuid, self.caller, signer);
6 }
```

Snippet 4.9: Snippet from `claim()`

Impact If the `ALEO_SIGNER_KEY` has been used to sign anything else, then that signature can be passed to `claim()` to claim rewards. This will work, as long as the data signed fits within the Aleo `field` type.

Recommendation Care should be taken to ensure that the `ALEO_SIGNER_KEY` has not been used to sign anything else. If the protocol intends to have a testnet then it should be ensured that the private keys used for signing are cycled and the same private key is never used twice.

Developer Response Accepted as a warning. Not much to do from the code side.

4.1.12 V-VCQ-VUL-012: Missing Await When Using Result of Async Call

Severity	Warning	**Commit**	440b876
Type	Denial of Service	**Status**	Fixed
File(s)	`src/services/users.service.ts`		
Location(s)			
Confirmed Fix At	N/A		

In the middle of the audit, the developers fixed an issue in `claimSignature()`. In this function, the Aleo quest contract is checked to see if the reward has been claimed for the given `userId` using `getClaimStatus()`. `getClaimStatus()` returns a Promise that will resolve to a boolean indicating whether the reward has been claimed. This promise is passed directly to an if condition without calling `await` which means that this function will always return an error saying that the reward has already been claimed because a Promise is a truthy value.

```
1  public async claimSignature(userId: string): Promise<Signature> {
2    const parsedId = convertUserIdToFieldType(userId);
3    const isClaimed = this.aleoContract.getClaimStatus(parsedId);
4
5    if (isClaimed) {
6      throw new HttpException(403, 'User has already claimed the quest');
7    }
```

Snippet 4.10: Snippet from `claimSignature()`

Impact Had this issue been present when the backend was deployed, users would be unable to claim rewards because the backend would reject any attempts to sign a uuid. This would have caused a temporary denial of service until this bug was fixed in the backend.

4.1.13 V-VCQ-VUL-013: Centralization Risk

Severity	Info	**Commit**	440b876
Type	Access Control	**Status**	Acknowledged
File(s)	`council.leo`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

The `council.aleo` program creates a council where members can vote on proposals which will only pass if enough votes to meet specified threshold are in favor of the proposal. Such a protocol mitigates some of the centralization risks associated with admin roles. However, this program allows any threshold (within a valid range) to be specified. For instance if the threshold is set to one, any member can pass any proposal, opening the program up to risks if private keys are stolen.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, if `threshold = 1` a malicious user could remove all other members of the council, and permanently pause the connected quest protocol. Additionally if the `threshold = 2` with only 2 members of the council, a malicious user could block all proposals from passing.

Recommendation To mitigate these risks it is recommended that the threshold be >=2 and the size of the council be >=3. Additionally, to prevent a single compromised account from performing a denial of service, the threshold should be less than the council size.

Developer Response We trust the council to set the threshold to the appropriate value. This is not enforced in the program and rather through the council.

4.1.14 V-VCQ-VUL-014: Typos, Minor Errors, and Missing documentation

Severity	Info	**Commit**	440b876
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`354bb36`		

Description In the following locations, the auditors identified minor errors and issues with the documentation of the protocol

- `council.leo`
  - When calling `propose()`, the `ZERO_ADDRESS` is included as a member and signs all proposals. This appears to be because there is a fixed size voters array when executing a proposal. So any extra spaces in that array can be filled with `ZERO_ADDRESS` values that won't fail the assertions in the finalize portions of those functions, and also won't be counted as a vote in when computing the number of unique voters.
    - This behavior is unintuitive and requires a complete understanding of the protocol to understand why this implementation makes sense. Thus, this logic should be included in documentation of this protocol to ensure future developers understand the relevant assumptions when modifying the code.
  - Inside the functions `finalize_update_signer`, `finalize_pause` and `finalize_unpause`, the variable `total_members` is assigned a value but it is never used.
  - line 41
    - The comment reads: `Tracks if the voters for the given proposal.`
    - It is unclear what this comment is intended to say.
  - `get_valid_unique_address_count()`
    - This will revert if any (non `ZERO_ADDRESS`) array elements are duplicated. This behavior isn't clear from the function name so it should be documented.
- `quest.leo`
  - The structs `TokenOwner` struct and `Balance` struct are unused.
  - line 46:
    - The comment here is incomplete.
- `src/services/users.service.ts`
  - The `claimSuccess()` function is unused.

Impact These minor errors and missing documentation may lead to future developer confusion.

Developer Response Removed all those inconsistencies mentioned.