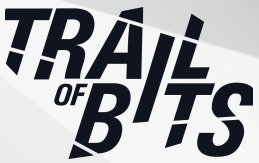

DFINITY Orbit

Security Assessment

September 5, 2025

Prepared for:
Robin Künzler
DFINITY

Prepared by: **Fredrik Dahlgren and Samuel Moelius**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering directors were associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Samuel Moelius, Consultant
sam.moelius@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 3, 2025	Pre-project kickoff call
February 25, 2025	Status update meeting #1
March 6, 2025	Delivery of report draft
March 6, 2025	Report readout meeting
April 28, 2025	Delivery of final comprehensive report
September 5, 2025	Completion of fix review

Executive Summary

Engagement Overview

DFINITY engaged Trail of Bits to review the security of Orbit, an Internet Computer–based platform for asset and canister management. Orbit authentication is based on the Internet Identity protocol. The platform has a built-in policy engine that allows Orbit users to define policy rules to control access to managed assets and canisters.

A team of two consultants conducted the review from February 14 to March 3, 2025, for a total of four engineer-weeks of effort. Our testing efforts focused on the security-critical components of the Orbit canisters and wallet front end. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

From September 3 to September 5, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the DFINITY team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue. For the results of this review, see appendix E.

Observations and Impact

Users log in to the Orbit wallet using Internet Identity. Concretely, this operation involves a WebAuthn-like flow where a short-lived public/private key pair is generated and the private key is stored in the user's browser. The bulk of the protocol is implemented by the `@dfinity/auth-client` library, which was out of scope for this review. We did not identify any concerns with how the library is used by the Orbit wallet, and the resulting delegated private key is stored securely in the browser in a way that ensures it is not accessible to other tabs.

The Orbit wallet's web UI has some confusing elements that could have security implications. For example, when a user is asked to approve an update to a resource tracked by Orbit, they are presented with the result of the update, but they are not shown *what* has changed (TOB-ORBIT-14). A user must perform additional investigation to determine this. If the user makes a mistake, they risk approving an update they did not intend to.

On a similar note, the web UI allows request policy rules to be applied where they should not be. Specifically, metadata rules make sense only for transfers, but the web UI allows them to be used in other contexts as well (TOB-ORBIT-15). An administrator that uses a metadata policy rule to configure edit access to their account could in this way lock themselves out by mistake.

The Orbit wallet interacts with the control panel and station canisters. The station canister implements a lot of the back-end logic of Orbit. It handles access controls based on request policy rules and tracks Orbit-controlled resources like accounts and canisters. We found the access control mechanism to be robust, but there are few guard rails in place to prevent user mistakes. For example, as mentioned above, administrators could lock themselves out by applying a metadata-based policy rule restricting administrator access (TOB-ORBIT-15).

The station canister generally implements sufficient input validation, but we still found a number of cases where request input validation could be strengthened to prevent invalid data from entering the system. For example, in certain situations, the station canister allows invalid transfers (TOB-ORBIT-3), invalid assets (TOB-ORBIT-4), invalid address book entries (TOB-ORBIT-5), and invalid canister IDs (TOB-ORBIT-6).

The station canister interacts with the upgrader canister, which can be used to upgrade the station canister and the upgrader canister itself. The upgrader canister includes a "disaster recovery" mechanism that can be used to recover from unsuccessful canister upgrades. However, the disaster recovery mechanism has not been fully implemented in the web UI.

Finally, we found that the upgrader canister performs sufficient logging to support log triage during an incident response. However, the station canister does not log events in the same way, and it does not provide an API to obtain event logs. Since Orbit manages sensitive resources like canisters and user funds, comprehensive and detailed logs are vital in case an organization needs to respond to a security incident.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the DFINITY team take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation to strengthen the overall security posture of the system.
- **Have humans extensively test all elements of the web UI.** Unit and integration tests can help to identify logical flaws, but they are poor tools for identifying sources of confusion. Having humans test the web UI is the best way to identify the latter.
- **Invest in unit and property testing to improve measurable test coverage.** Given that there is currently no way to compute test coverage for Wasm code, it is impossible to determine the parts of the canister implementations that are covered by the integration test suite running under `pocket-ic`. We recommend that DFINITY invest in unit and property testing to improve measurable coverage of the Orbit canister implementations. In particular, we believe that adding property tests would greatly improve coverage on invalid and unexpected inputs. In the long term,

we also believe it would make sense to invest in a solution that adds coverage reporting functionality to `pocket-ic` or a similar tool.

- **Add logging functionality to the station.** Comprehensive and detailed logs should be produced by the station canister. Such logs will be needed should DFINITY ever have to respond to a security incident.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	2
Low	5
Informational	13
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	2
Data Validation	10
Denial of Service	1
Error Reporting	3
Patching	3
Session Management	2
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the DFINITY Orbit asset and canister custody solution. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are request state transitions and journaling handled correctly by the station canister?
- Are request inputs sufficiently validated?
- Are request policy rules evaluated correctly?
- Can users approve a request at most once?
- Is request approval restricted to the relevant users/groups?
- Is canister state managed correctly across asynchronous inter-canister calls?
- Is it possible for a user to lock themselves out by restricting resource access using an invalid policy rule?
- Does the implementation allow for effective triage if an incident occurs?
- Does the wallet front end manage local state in a secure manner?
- Are upgrade requests handled correctly by the upgrader canister?
- Is disaster recovery implemented in a secure manner to allow for secure state rollback in the event of an incident?

Project Targets

The engagement involved reviewing and testing the following target.

Orbit

Repository	https://github.com/dfinity/orbit
Version	`6f8dd70d4881a236b796bdeff0f52233915b4c3d`
Type	Rust, TypeScript
Platform	Internet Computer

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Static analysis:** We ran Clippy and Semgrep over the codebase and reviewed the results.
- **Dependency analysis:** We ran `cargo update` to look for compatible upgrades the project might be missing. We ran `cargo upgrade --incompatible` to look for incompatible upgrades the project might take advantage of. Finally, we ran `cargo audit` to look for dependencies with outstanding RustSec advisories.
- **Test coverage review:** We verified that all of the project's tests pass. We also computed the project's test coverage using `cargo-llvm-cov` to look for important conditions the tests might have missed.
- **Manual review:** We manually reviewed the components in the list below.
    - **Station:** We traced the path of each request type from ingress to execution to ensure that request inputs are sufficiently validated, that permissions and request policy rules are enforced by the implementation, and that request execution is implemented correctly for each request type. We also looked for time-of-check to time-of-use race conditions that can occur during inter-canister calls.
    - **Upgrader:** We checked that appropriate access controls are in place. We also checked that logging is implemented properly.
    - **Control panel:** We focused on the canister registry, as this was deemed the most security-critical component by DFINITY. We checked that the registry performs proper canister validation and that requests from the web UI are serviced faithfully.
    - **Wallet front end:** We checked that secrets are handled and stored properly. We also looked for situations where data presented by the wallet UI could be misinterpreted by the user.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **The `orbit-essentials` and `orbit-essential-macros` libraries:** We briefly reviewed parts of the `orbit-essentials` library and the corresponding macros library to ensure that we understood the canister implementations, but these libraries did not receive a comprehensive review.
- **Fund manager and the `canfund` library:** The cycle manager responsible for making sure that managed canisters have an adequate cycles balance is implemented using the `FundManager` type from the `canfund` library. These were not reviewed as part of the engagement.
- **Canister API types:** The canister API types (defined under the `api` directories) and corresponding conversions to canister internal types were not covered during the review.
- **Integration test coverage:** Since the integration tests run under the `pocket-ic` test runner, we were not able to obtain coverage data for the integration tests during the review.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the tools in the table below for the automated testing phase of this project. Instructions for installing the tools can be found in appendix C.

Tool	Description
`cargo-audit`	A Cargo subcommand that can be used to audit project dependencies for known vulnerabilities
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage
`cargo-upgrade`	A Cargo subcommand to upgrade dependencies in `Cargo.toml` to their latest versions
Clippy	A Rust linter used to catch common mistakes and unidiomatic Rust code
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
ShellCheck	A tool that gives warnings and suggestions for Bash/sh shell scripts
`zizmor`	A static analysis tool for GitHub actions

Areas of Focus

Our automated testing and verification work focused on the following:

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of `unwrap` and `expect`
- Poor unit and integration test coverage
- General issues with dependency management and known vulnerable dependencies

- Common mistakes and security issues in GitHub actions

Test Results

The results of this focused testing are detailed below.

`cargo-audit`

The `cargo-audit` tool identified several unmaintained dependencies and one dependency with an outstanding RustSec advisory (TOB-ORBIT-8).

`cargo-llvm-cov`

The coverage report obtained from `cargo-llvm-cov` shows test coverage only for the unit test suite. Since the integration tests run under the `pocket-ic` test runner, there is currently no way to collect test coverage reports for these tests.

The test coverage report for the unit test suite shows that test coverage could be improved across all canisters (TOB-ORBIT-23).

`cargo-upgrade`

Running `cargo upgrade --incompatible` revealed 14 packages that could be upgraded and that would not be updated simply by running `cargo update` (TOB-ORBIT-8).

Clippy

Running Clippy 1.78 produced no warnings. However, running a more recent Clippy version (1.84) produced several warnings (TOB-ORBIT-7).

Semgrep

We ran Semgrep Pro on the codebase using both the default configuration and our internal rulesets. This analysis did not identify any security issues in the codebase.

ShellCheck

We ran ShellCheck over the scripts in the `scripts` subdirectory. This produced one warning regarding an unused variable and nine warnings regarding unquoted variables (TOB-ORBIT-9).

`zizmor`

Running `zizmor` to analyze the GitHub workflows used by the project identified three low-severity issues with how GitHub actions are used (TOB-ORBIT-10, TOB-ORBIT-11, and TOB-ORBIT-12).

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Most of the arithmetic operations in the station codebase use saturating operations to avoid overflows and underflows. However, we found one potential integer overflow issue (TOB-ORBIT-1) that could cause issues with paginated lists returned from the station.	**Satisfactory**
Auditing	Logs are generated under important conditions and with pertinent information in the upgrader. However, the station has no similar functionality. Comprehensive and detailed logs will be needed should DFINITY ever have to respond to a security incident.	**Moderate**
Authentication / Access Controls	User authentication is implemented using Internet Identity, which provides a robust way to outsource Orbit user authentication to an already existing, vetted protocol. The delegated key is stored in the browser using IndexedDB and is not accessible to other tabs. Station resource access controls are implemented using the `authorize` middleware. This makes access control checks easy to review for correctness.	**Satisfactory**
Complexity Management	The codebase is generally easy to navigate. Canister implementations are structured similarly, which makes the codebase easier to maintain and review for correctness. Most functions and most types have a single clear purpose, which also reduces the overall complexity of the codebase. However, some functions and methods are quite long or use heavy nesting, which makes them difficult to read. There are multiple examples where one or more function calls are inlined in the definition of a new object, which in	**Moderate**

	turn is inlined as an argument to another function call. This coding practice needlessly makes the code more complex and difficult to review. We also found a number of issues related to dependency management. The project uses an old Rust toolchain (TOB-ORBIT-7) and has a number of outdated dependencies, one of which has an outstanding RustSec advisory (TOB-ORBIT-8).	
Cryptography and Key Management	Account keys are managed by the station. For chain-key assets like ckETH and ckBTC, keys are managed using the threshold chain-key implementation provided by the Internet Computer.	**Satisfactory**
Decentralization	The system provides a decentralized platform for asset and canister management. However, this presupposes that strict policy rules are in place to restrict privileged operations like editing accounts and assets. If a single user is able to edit station-controlled resources, it may be possible for them to trick other users into transferring funds by mistake (TOB-ORBIT-18).	**Moderate**
Documentation	The high-level documentation for the project was sufficient to understand the project functionality and scope. However, we sometimes found that the code could benefit from more code-level comments describing the use cases for particular types and traits defined by the implementation.	**Satisfactory**
Testing and Verification	The codebase defines an extensive test suite with both unit and integration tests. There are also negative tests that exercise the code on unexpected or invalid data to ensure that exceptional cases are handled correctly. However, the integration test suite uses the `pocket-ic` test runner to execute. This means that it is not possible to obtain accurate test coverage data for the integration tests, making it difficult to determine which components lack testing. There are also no property tests in the test suite. Implementing some form of randomized testing, like property testing, would improve test coverage and help ensure that important security invariants are upheld by the code.	**Satisfactory**

Transaction Ordering	Since requests are serviced asynchronously by the station, the ordering guarantees provided by the Internet Computer do not apply directly to Orbit, and we did identify one issue (TOB-ORBIT-13) where requests could be delayed if the load of incoming requests to the station is high. Apart from this, we did not find any issues related to the ordering of requests, transfers, or canister calls, and the underlying execution layer ensures that all incoming requests are serviced in the order that they arrive.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity. For more information on the issues that the DFINITY team has addressed and on the mitigations implemented, see appendix E.

ID	Title	Type	Severity
1	Paginated queries may return the wrong list of items	Data Validation	**Medium**
2	Transfer execution function may set an incorrect transfer status	Error Reporting	**Informational**
3	The station accepts invalid transfers	Data Validation	**Informational**
4	The station canister allows for the creation of invalid assets	Data Validation	**Informational**
5	Insufficient validation of address book addresses	Data Validation	**Informational**
6	Request specifier validation does not validate canister IDs	Data Validation	**Informational**
7	Use of old Rust toolchain	Patching	**Informational**
8	Outdated and vulnerable dependencies	Patching	**Informational**
9	ShellCheck warnings	Patching	**Informational**
10	Overly broad GitHub workflow permissions	Session Management	**Low**
11	Potential credential persistence through GitHub actions artifacts	Session Management	**Informational**
12	Unpinned external GitHub CI/CD action versions	Auditing and Logging	**Low**

13	New requests may arbitrarily delay earlier requests	Denial of Service	**Low**
14	Web UI shows an update's result, not its diff	Data Validation	**High**
15	Metadata rules allowed where they are inapplicable	Access Controls	**Medium**
16	request_recovery silently fails if caller is not a committee member	Error Reporting	**Low**
17	Asset edit requests can set conflicting or invalid asset metadata	Data Validation	**High**
18	The asset edit API endpoint ignores request expiration times	Data Validation	**Low**
19	Request approval submission API does not update modification timestamp	Auditing and Logging	**Informational**
20	Balance request API silently ignores invalid account IDs	Error Reporting	**Informational**
21	Ad hoc validation of request operation inputs	Data Validation	**Informational**
22	Measurable test coverage is low	Testing	**Undetermined**
23	validate_dependencies is not recursive	Data Validation	**Informational**

Detailed Findings

1. Paginated queries may return the wrong list of items	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ORBIT-1
Target: `core/station/impl/src/core/utils.rs`	

Description

The `offset` and `limit` parameters passed to the `paginated_items` function are converted to `usize`, which is 32 bits for `wasm32` build targets. The sum of `offset` and `limit` is used to determine the next offset and the paginated list of items returned by the function. If the result of `offset + limit` overflows, then it will be less than `offset`, which means that `next_offset` will be less than `offset` and the paginated list of items returned by the function will be empty.

```
let next_offset = match (offset + limit) < total {
    true => Some((offset + limit) as u64),
    false => None,
};

let items = args
    .items
    .get(offset..std::cmp::min(offset + limit, total))
    .unwrap_or(&[])
    .to_vec();

Ok(PaginatedData {
    items,
    next_offset,
    total: total as u64,
})
```

Figure 1.1: core/station/impl/src/core/utils.rs#56–71

This could lead to issues when processing lists containing a large amount of resources (on the order of 2^{32}). In particular, it could cause problems in situations where unexpected return values are not readily apparent, such as in scripted applications.

Exploit Scenario

An Orbit user queries the station canister for a list of resources. Because of an overflow in the `paginated_items` function, the returned list is empty. This causes the user to make the wrong decision based on the invalid list of resources returned by the canister.

Recommendations

Short term, use checked addition to ensure that overflows are handled correctly by the function.

Long term, consider enabling overflow checks in release mode to detect these types of issues by setting the `overflow-checks` option to true in the `[profile.release]` section in the `Cargo.toml` project file.

2. Transfer execution function may set an incorrect transfer status

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-ORBIT-2
Target: `core/station/impl/src/jobs/execute_created_transfers.rs`	

Description

The `Job::execute_created_transfers` function, which is used to execute transfers, has an indexing issue: it uses the index from a filtered list to index into the original, unfiltered list, so it may inadvertently mark completed transactions as failed.

The `Job::execute_created_transfers` function filters out transfers that correspond to an existing request in the request repository and then attempts to execute each individual transfer in this list. When each asynchronous call has completed, the function processes the list of returned results to update the status of each transfer.

```
// batch the transfers to be executed
let calls = transfers
    .clone()
    .into_iter()
    .filter(|transfer| requests.contains_key(&transfer.id))
    .map(|transfer| self.execute_transfer(transfer));

// wait for all the transfers to be executed
let results = future::join_all(calls).await;
```

Figure 2.1: The list of execution results may be strictly shorter than the list of transfers. (core/station/impl/src/jobs/execute_created_transfers.rs#101–110)

Since some of the original transfers may have been filtered out, the list of results can be shorter than the original list of transfers. This means that in general, the result at a given offset `pos` will not correspond to the transfer at offset `pos` in the original list. However, the function still uses this index to obtain the corresponding transfer when execution fails. This means that the wrong transfer could be marked as failed by mistake.

```
for (pos, result) in results.iter().enumerate() {
    match result {
        Ok((transfer, details)) => {
            // ...
        }
        Err(e) => {
            let mut transfer = transfers[pos].clone();
            transfer.status = TransferStatus::Failed {
```

```
                reason: e.to_string(),
            };
            let transfer_failed_time = next_time();
            transfer.last_modification_timestamp = transfer_failed_time;
            self.transfer_repository
                .insert(transfer.to_key(), transfer.to_owned());

            if let Some(request) = requests.get(&transfer.id) {
                let request = request.clone();
                self.request_service
                    .fail_request(request, e.to_string(), transfer_failed_time)
                    .await;
            } else {
                print(format!(
                    "Error: request not found for transfer {}",
                    Uuid::from_bytes(transfer.id).hyphenated()
                ));
            }
        }
    }
}
```

Figure 2.2: Because the function uses an index from the list of execution results to index into the original list of transfers, it could return the wrong transfer if some transfers were filtered out. (core/station/impl/src/jobs/execute_created_transfers.rs#112–179)

As an example, suppose that the original list consists of the transfers $[T_0, T_1, T_2, T_3]$ and that the first transaction T_0 is filtered out prior to execution. The three remaining transfers are executed, resulting in the results $[R_1, R_2, R_3]$, where result R_i corresponds to transaction T_i.

Now, if T_2 fails to execute and R_2 is an error result, the `for` loop highlighted in figure 2.2 will retrieve the transfer T_1 at offset 1 (since R_2 has index 1 in the list of execution results) and mark it and the corresponding request as failed. When this result is propagated back to the end user, they may erroneously believe that the transfer T_1 failed even though it succeeded, which could lead to the user double-spending funds by mistake.

Furthermore, since the list of results is strictly shorter than the list of transfers, the status of the final transaction T_3 will not be updated, even though it has been executed. This means that the final transaction T_3 will never be marked as completed. This could also lead to confusion for the end user as to whether the transfer was processed.

We note that a transfer will be filtered out when the `calls` iterator is defined (in figure 2.1) only if there is no request corresponding to the transfer request ID. Since the request ID is validated when the transfer is created and requests are never removed from the request repository, we do not currently see how this could occur in practice. In spite of this, we still believe that this issue represents a serious risk to the system, since a small and seemingly

unrelated change to the codebase would cause this issue to become exploitable and allow malicious users to steal funds through double spends.

The same code pattern is present in the `Job::execute_scheduled_requests` function, which is used to service scheduled requests. However, the list of requests to be executed is not filtered in this function, which means that it is not affected by the issue described above.

Exploit Scenario

The development team introduces a change causing old requests to be pruned from the request repository, which causes the issue above to become exploitable. A malicious user creates a large number of transfer requests. As these are executed, one of the requests has already been pruned from the request repository, causing a double spend when the final transfer in the batch is executed twice.

Recommendations

Short term, return the transfer (or transfer ID) of the failing transfer as part of the error from the `Job::execute_transfer` function, and use this to update the statuses of the failing transfer and corresponding request.

Long term, ensure that unit and integration tests cover failure paths to ensure that unexpected failures are handled correctly.

3. The station accepts invalid transfers

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ORBIT-3
Target: `core/station/impl/src/models/transfer.rs`	

Description

The `Transfer::validate` method is used to validate incoming transfers. However, the following properties are not checked by the function.

- That asset IDs correspond to assets registered with the system
- That the blockchain network corresponds to a supported blockchain
- That the recipient address is a valid address

```
impl ModelValidator<TransferError> for Transfer {
    fn validate(&self) -> ModelValidatorResult<TransferError> {
        self.metadata.validate()?;
        validate_to_address(&self.to_address)?;
        validate_network(&self.blockchain_network)?;

        EnsureUser::id_exists(&self.initiator_user).map_err(|err| match err {
            RecordValidationError::NotFound { id, .. } =>
                TransferError::ValidationError {
                    info: format!("The initiator_user {} does not exist", id),
                },
        })?;

        EnsureAccount::id_exists(&self.from_account).map_err(|err| match err {
            RecordValidationError::NotFound { id, .. } =>
                TransferError::ValidationError {
                    info: format!("The from_account {} does not exist", id),
                },
        })?;

        EnsureRequest::id_exists(&self.request_id).map_err(|err| match err {
            RecordValidationError::NotFound { id, .. } =>
                TransferError::ValidationError {
                    info: format!("The request_id {} does not exist", id),
                },
        })?;

        Ok(())
    }
```

```
}
```

Figure 3.1: core/station/impl/src/models/transfer.rs#182–208

The asset ID is not checked at all. The validation functions checking the blockchain network and recipient address ensure only that the string length is within a set of predefined bounds.

Recommendations

Short term, use the `EnsureAsset::id_exists` function to ensure that the asset ID corresponds to an asset registered with the asset repository.

Use the static vector `SUPPORTED_BLOCKCHAINS` to validate the blockchain network. Finally, use the `AccountAddress` type to represent the recipient address, and use the `AccountAddress::validate` method to check that it is correct.

Long term, consider implementing procedural macros that automatically generate validation code for all model fields based on their types.

4. The station canister allows for the creation of invalid assets

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ORBIT-4
Target: `core/station/impl/src/models/asset.rs`	

Description

The `Asset::validate` method used to validate registered assets does not validate the asset token standards against the asset blockchain. This means that users could inadvertently register invalid assets with the system.

```
impl ModelValidator<AssetError> for Asset {
    fn validate(&self) -> ModelValidatorResult<AssetError> {
        validate_symbol(&self.symbol)?;
        validate_name(&self.name)?;
        validate_decimals(self.decimals)?;
        validate_uniqueness(&self.id, &self.symbol, &self.blockchain)?;

        self.metadata.validate()?;

        Ok(())
    }
}
```

Figure 4.1: The blockchain and token standard fields are not validated. (core/station/impl/src/models/asset.rs#133–144)

Recommendations

Short term, check the asset token standards against the blockchain using the `Blockchain::supported_standards` method.

Long term, extend the test suite to ensure that invalid model values are rejected by the system.

5. Insufficient validation of address book addresses

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ORBIT-5
Target: `core/station/impl/src/models/address_book.rs`	

Description

The `AddressBookEntry::validate` method validates only the length of the address field. The address format field is not used, which increases the risk that a user could specify an invalid address by mistake.

```
fn validate_address(address: &str) -> ModelValidatorResult<AddressBookError> {
    if (address.len() < AddressBookEntry::ADDRESS_RANGE.0 as usize)
        || (address.len() > AddressBookEntry::ADDRESS_RANGE.1 as usize)
    {
        return Err(AddressBookError::InvalidAddressLength {
            min_length: AddressBookEntry::ADDRESS_RANGE.0,
            max_length: AddressBookEntry::ADDRESS_RANGE.1,
        });
    }

    Ok(())
}
```

Figure 5.1: The address is not validated against the expected address format. (core/station/impl/src/models/address_book.rs#65–76)

Recommendations

Short term, use the `AccountAddress` type to represent the address, and use `AccountAddress::validate` to validate it. Since the address format is part of the `AccountAddress` type, the `validate` function will validate the address against the address format.

Long term, implement unit tests to ensure that created objects are always internally consistent.

6. Request specifier validation does not validate canister IDs

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ORBIT-6
Target: `core/station/impl/src/models/request_specifier.rs`	

Description

The `RequestSpecifier::validate` method does not validate the canister ID fields that are part of the `ChangeExternalCanister` and `FundExternalCanister` variants.

```
impl ModelValidator<ValidationError> for RequestSpecifier {
    fn validate(&self) -> ModelValidatorResult<ValidationError> {
        match self {
            RequestSpecifier::AddAccount
            | RequestSpecifier::AddUser
            | RequestSpecifier::AddAddressBookEntry
            | RequestSpecifier::SystemUpgrade
            | RequestSpecifier::ChangeExternalCanister(_)
            | RequestSpecifier::FundExternalCanister(_)
            | RequestSpecifier::CreateExternalCanister
            | RequestSpecifier::AddRequestPolicy
            | RequestSpecifier::ManageSystemInfo
            | RequestSpecifier::SetDisasterRecovery
            | RequestSpecifier::AddUserGroup
            | RequestSpecifier::AddAsset
            | RequestSpecifier::AddNamedRule => (),

            // ...
        }
    }
}
```

Figure 6.1: Canister IDs are not validated.
(core/station/impl/src/models/request_specifier.rs#90–107)

Recommendations

Short term, use `EnsureExternalCanister::is_external_canister` to validate the canister IDs.

Long term, extend the unit test suite to ensure that invalid model values are rejected by the corresponding validation method.

7. Use of old Rust toolchain

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Patching	Finding ID: TOB-ORBIT-7
Target: `rust-toolchain.toml`	

Description

The project uses a Rust toolchain (figure 7.1) that is just under nine months old and seven releases prior to the current Rust version (1.85.0). Using an old toolchain implies the code does not benefit from more recently introduced error checks.

```
1    [toolchain]
2    channel = "1.78.0"
```

Figure 7.1: Toolchain used by Orbit (`orbit/rust-toolchain.toml#1–2`)

The following example demonstrates how Orbit would benefit from more recently introduced error checks. Running Clippy 1.84 over the codebase produces 17 warnings not produced by Clippy 1.78 (see table 7.2). Thus, while Clippy 1.78 does not warn about potential problems in the codebase, Clippy 1.84 does.

Lint Name	Number of Warnings
`clippy::legacy_numeric_constants`	3
`clippy::manual_inspect`	2
`clippy::manual_pattern_char_comparison`	1
`clippy::needless_borrows_for_generic_args`	1
`clippy::needless_lifetimes`	6
`clippy::needless_return`	1
`clippy::unnecessary_map_or`	3

Table 7.2: Warnings produced by running Clippy 1.84 over the codebase

Exploit Scenario

The Orbit wallet is enabled for production use, though it contains a bug. The bug would have been flagged by a more recent Rust toolchain.

Recommendations

Short term, upgrade the project's `rust-toolchain.toml` to use a more recent Rust toolchain, such as 1.85.0 (current as of this writing). Doing so will allow the project to benefit from more recent error checks, such as those provided by Clippy 1.85.

Long term, regularly check for new versions of Rust. Consider following the Rust blog (where new Rust versions are announced) or running `rustup` in a GitHub workflow to determine the latest stable toolchain. Taking these steps will help ensure the project uses the most up-to-date toolchain.

8. Outdated and vulnerable dependencies

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Patching	Finding ID: TOB-ORBIT-8
Target: `Cargo.toml`, `Cargo.lock`	

Description

The project relies on several packages for which more recent versions exist. Additionally, the project relies on at least one package with a known vulnerability. Since silent bug fixes are common, the project should rely on the most recent versions of its dependencies whenever possible.

Table 8.1 lists 14 packages that Orbit relies on that could be upgraded. Note that Cargo considers the newer version of each package to be incompatible with the one currently used. Thus, the upgrades must be performed manually or with `cargo upgrade --incompatible`; they cannot simply be performed with `cargo update`.

Package	Version Currently Used	Latest Version Available
`convert_case`	0.6.0	0.7.1
`getrandom`	0.2.15	0.3.1
`ic-agent`	0.38.2	0.39.3
`ic-cdk`	0.16.0	0.17.1
`ic-cdk-macros`	0.16.0	0.17.1
`ic-utils`	0.38.2	0.39.3
`itertools`	0.13.0	0.14.0
`mockall`	0.12.1	0.13.1
`rand`	0.8.5	0.9.0
`rand_chacha`	0.3.1	0.9.0
`rstest`	0.18.2	0.24.0
`strum`	0.26.3	0.27.0

tabled	0.16.0	0.18.0
thiserror	1.0.69	2.0.11

Table 8.1: Packages that can be upgraded

Additionally, running `cargo audit` produces the error in figure 8.2. Running `cargo update -p url` updates the package to version 2.5.4 and makes the error go away.

```
Crate:     idna
Version:   0.5.0
Title:     `idna` accepts Punycode labels that do not produce any non-ASCII when
decoded
Date:      2024-12-09
ID:        RUSTSEC-2024-0421
URL:       https://rustsec.org/advisories/RUSTSEC-2024-0421
Solution:  Upgrade to >=1.0.0
Dependency tree:
idna 0.5.0
└── url 2.5.2
    ├── reqwest 0.12.9
    │   ├── pocket-ic 6.0.0
    │   │   └── integration-tests 0.0.1
    │   ├── integration-tests 0.0.1
    │   ├── ic-agent 0.38.2
    │   │   ├── ic-utils 0.38.2
    │   │   │   ├── ic-asset 0.21.0
    │   │   │   │   └── dfx-orbit 0.9.0
    │   │   │   │       └── integration-tests 0.0.1
    │   │   │   ├── dfx-orbit 0.9.0
    │   │   │   └── dfx-core 0.1.0
    │   │   │       ├── ic-asset 0.21.0
    │   │   │       └── dfx-orbit 0.9.0
    │   │   ├── ic-identity-hsm 0.38.2
    │   │   │   └── dfx-core 0.1.0
    │   │   ├── ic-asset 0.21.0
    │   │   ├── dfx-orbit 0.9.0
    │   │   └── dfx-core 0.1.0
    │   └── dfx-core 0.1.0
    ├── ic-agent 0.38.2
    └── dfx-core 0.1.0
```

Figure 8.2: Error produced by running `cargo audit` over the codebase

Exploit Scenario

Eve discovers a vulnerability in an Orbit dependency. Eve exploits the vulnerability knowing that Orbit relies on the outdated, vulnerable version.

Recommendations

Short term, regularly take the following steps:

- Run `cargo update` to obtain the latest compatible version of each package that Orbits relies on.
- Run `cargo upgrade --incompatible --dry-run` to see incompatible upgrades that are available.
- Run `cargo audit` to determine whether Orbit relies on packages with known vulnerabilities.

Taking these steps will help to ensure that Orbit relies on the most up-to-date versions of each of its dependencies and that it relies on dependencies free from known vulnerabilities.

Long term, automate the above steps with a GitHub workflow or with Dependabot to ensure that they are performed regularly.

References

- `cargo edit` (provides the `cargo upgrade` command)
- RustSec: `cargo audit`

9. ShellCheck warnings

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-ORBIT-9
Target: `orbit` (shell script)	

Description

The project uses a Bash shell script as part of its production code. Running `shellcheck` against the script produces several warnings, including unquoted variables. The warnings should be addressed if the script will continue to be used for production.

```
 108    Then the following steps can be used to setup the Orbit canister ecosystem
for local development.
 109
 110    ```bash
 111    ./orbit --init
 112    ```
 113
 114    This will build the canisters, install the required node modules and deploy
the canisters to your local replica. All the canisters will be deployed to the
`local` network with their fixed canister ids.
```

Figure 9.1: Excerpt of `README.md` referring to the `orbit` shell (audit-dfinity/README.md#108–114)

When run against the script, `shellcheck` warns about the following:

- One unused variable (`CANISTER_ID_WALLET`)
- Nine unquoted variables

An example of the latter appears in figure 9.2. Note that using an unquoted variable would enable command execution if an attacker is able to influence the variable's contents.

```
In orbit line 91:
    ./scripts/generate-wasm.sh $canister_name
                               ^------------^ SC2086 (info): Double quote to prevent
globbing and word splitting.
```

Figure 9.2: Example warning regarding an unquoted variable

Exploit Scenario

Eve finds a way to influence the contents of one of the unquoted variables the `orbit` shell script uses. Eve uses this capability to execute commands on Alice's machine.

Recommendations

Short term, run `shellcheck` against the `orbit` shell script and address each of the warnings produced. Doing so will help ensure that running the script produces only its intended effects.

Long term, seek a solution for building and installing canisters that does not use shell scripts. Writing robust shell scripts can be difficult. Hence, modern programming languages built for application development should be considered.

References

- ShellCheck: A shell script static analysis tool

10. Overly broad GitHub workflow permissions

Severity: **Low**	Difficulty: **High**
Type: Session Management	Finding ID: TOB-ORBIT-10
Target: `.github/workflows`	

Description

All of the jobs that are executed as part of the workflows in the Orbit repository run with default permissions. If the default permissions defined by the repository or DFINITY GitHub organization are too permissive, an attacker who is able to execute code during workflow execution could use this to compromise the repository.

In general, workflow permissions should be declared as minimally as possible and as close to their usage site as possible.

In practice, this means that workflows should almost always set `permissions: {}` at the workflow level to disable all permissions by default, and then set specific job-level permissions as needed. However, this is currently not done in any of the workflows used in the Orbit repository.

```
warning[excessive-permissions]: overly broad permissions
  --> .github/workflows/tests.yaml:30:3
   |
30 | /   test-rust:
31 | |     name: 'test-rust:required'
...  |
38 | |       - name: 'Test cargo crates'
39 | |         run: cargo test --locked --workspace --exclude integration-tests
   | |                                                                        -
   | |________________________________________________________________________|
   |                                                                      this job
   |                       default permissions used due to no permissions: block
   |
   = note: audit confidence → Medium
```

Figure 10.1: Running `zizmor` shows that all of the workflow jobs run with default permissions. (`.github/workflows/tests.yaml`)

Exploit Scenario

Eve is able to compromise a GitHub action used by the Orbit repository. This allows her to execute code when the action runs. Since the action runs with default permissions, the action has write permissions to the GitHub artifact cache. This allows Eve to poison the repository cache, allowing her to infect build artifacts with malicious code.

Recommendations

Short term, assign permissions per job in each of the workflows used by the Orbit repository.

Long term, add `actionlint`, `poutine`, and `zizmor` to the repository CI/CD pipeline.

11. Potential credential persistence through GitHub actions artifacts

Severity: **Informational**	Difficulty: **High**
Type: Session Management	Finding ID: TOB-ORBIT-11
Target: `.github/workflows`	

Description

The default behavior of the `actions/checkout` GitHub action is to persist credentials, which means that the GitHub token is written to the local repository directory. This allows the action to execute Git commands that require authentication. The credentials are stored in the `.git/config` file in the local repository directory, where they could be inadvertently included in workflow artifacts or be accessed by subsequent workflow steps.

```
warning[artipacked]: credential persistence through GitHub Actions artifacts
   --> .github/workflows/tests.yaml:205:9
    |
205 |         - name: 'Checkout'
    |  _________-
206 | |         uses: actions/checkout@v4
    | |_____________________________________- does not set persist-credentials: false
    |
    = note: audit confidence → Low
```

Figure 11.1: `zizmor` identifies multiple locations where GitHub credentials are persisted by the `actions/checkout` action. (`.github/workflows/tests.yaml`)

If this is not required, it is recommended to set the `persist-credentials` property to `false` for the `actions/checkout` action.

Recommendations

Short term, add `persist-credentials: false` to checkout actions that do not require privileged Git operations.

Long term, add `actionlint`, Poutine, and Zizmor to the CI/CD pipeline.

References

- ArtiPACKED: Hacking Giants through a Race Condition in GitHub Actions Artifacts
- `zizmor` ArtiPACKED documentation

12. Unpinned external GitHub CI/CD action versions

Severity: **Low**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-ORBIT-12
Target: `.github/workflows`	

Description

Several GitHub workflows in the Orbit repository use third-party actions pinned to a tag or branch name instead of a full-length commit SHA, as recommended by GitHub. This configuration enables repository owners to silently modify the actions. A malicious actor could use this ability to tamper with an application release or leak secrets.

The following actions are owned by GitHub organizations or individuals that are not affiliated directly with DFINITY.

- `docker-practice/actions-setup-docker@master`
- `mozilla-actions/sccache-action@v0.0.4`
- `pnpm/action-setup@v4`

```
- uses: docker-practice/actions-setup-docker@master
  timeout-minutes: 12
```

Figure 12.1: GitHub workflows depend on a number of third-party actions that are pinned only to a tag. (`.github/workflows/tests.yaml#212–213`)

Exploit Scenario

An attacker gains unauthorized access to the account of a GitHub action owner. The attacker manipulates the action's code to steal GitHub credentials and compromise the Orbit repository.

Recommendations

Short term, pin each third-party action to a specific full-length commit SHA, as recommended by GitHub. Additionally, configure Dependabot to keep GitHub actions up to date and to update the action commit SHAs after reviewing their updates.

Long term, add `actionlint`, `poutine`, and `zizmor` to the CI/CD pipeline.

13. New requests may arbitrarily delay earlier requests

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-ORBIT-13
Target: `core/station/impl/src/jobs/execute_scheduled_requests.rs`	

Description

Scheduled requests are serviced in batches by the `Job::execute_scheduled_requests` function. If more than 20 (`Job::MAX_BATCH_SIZE`) requests are scheduled to execute, the function truncates the list of requests returned by `RequestReport::find_scheduled`.

```
let mut requests = self
    .request_repository
    .find_scheduled(None, Some(current_time));

let num_processing_requests = self.request_repository.get_num_processing();
let batch_size = std::cmp::min(
    Self::MAX_PROCESSING_REQUESTS.saturating_sub(num_processing_requests),
    Self::MAX_BATCH_SIZE,
);

let processing_all_requests = requests.len() <= batch_size;

// truncate the list to avoid processing too many requests at once
requests.truncate(batch_size);
```

Figure 13.1: The list of scheduled requests is truncated, and only the first 20 are serviced. (core/station/impl/src/jobs/execute_scheduled_requests.rs#47–60)

However, the request repository APIs do not provide any ordering guarantees for the list of requests returned to the caller. Since this list is truncated and only the first 20 requests are serviced by the request executor, this means that requests can be arbitrarily delayed during times when the request load is high.

```
#[test]
fn test_scheduled_request_order() {
    let repository = RequestRepository::default();
    // Create and schedule requests.
    for i in 0..20 {
        let mut request = request_test_utils::mock_request();
        request.id = *Uuid::new_v4().as_bytes();
        request.created_timestamp = i;
        request.status = RequestStatus::Scheduled {
            scheduled_at: i + 10,
```

```
        };
        request.expiration_dt = i + 20;
        repository.insert(request.to_key(), request.clone());
    }
    let requests = repository.find_scheduled(None, Some(100));
    // Print creation timestamps.
    println!(
        "[{}]",
        requests
            .iter()
            .map(|r| r.created_timestamp.to_string())
            .collect::<Vec<_>>()
            .join(", ")
    );
    // Print scheduled timestamps.
    println!(
        "[{}]",
        requests
            .iter()
            .map(
                |r| if let RequestStatus::Scheduled { scheduled_at } = r.status {
                    scheduled_at.to_string()
                } else {
                    "".to_string()
                }
            )
            .collect::<Vec<_>>()
            .join(", ")
    );
    // Print expiration timestamps.
    println!(
        "[{}]",
        requests
            .iter()
            .map(|r| r.expiration_dt.to_string())
            .collect::<Vec<_>>()
            .join(", ")
    );
}
// running 1 test
//
// ...
//
// ---- repositories::request::tests::find_by_status_and_expiration_dt stdout ----
//
// [9, 19, 13, 11, 15, 4, 8, 16, 18, 7, 3, 5, 12, 10, 14, 0, 17, 1, 6, 2]
// [29, 39, 33, 31, 35, 24, 28, 36, 38, 27, 23, 25, 32, 30, 34, 20, 37, 21, 26, 22]
// [19, 29, 23, 21, 25, 14, 18, 26, 28, 17, 13, 15, 22, 20, 24, 10, 27, 11, 16, 12]
```

Figure 13.2: Printing the output from `RequestRepository::find_scheduled` shows that the output is not ordered by either creation time, scheduled time, or expiration time.

The same issue is present in the `Job::execute_created_transfers` function (defined in `core/station/impl/src/jobs/execute_created_transfers.rs`), which truncates the list of transfers and services only the first 20 (`Job::MAX_BATCH_SIZE`).

Exploit Scenario

Eve, a malicious user, wants to delay a particular transfer from completing. She sends a large number of requests to the station. Since requests are serviced in a random order, this ends up delaying the targeted transfer request.

Recommendations

Short term, have `Job::execute_scheduled_requests` sort the list of requests on creation time before truncating it. Similarly, have `Job::execute_created_transfers` sort transfers on creation time before truncating the list.

14. Web UI shows an update's result, not its diff	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ORBIT-14
Target: Web UI	

Description

When a resource update must be approved, only the final result of the update is presented to the approving party. In particular, *what* is changed is not presented to the approving party. This could allow an update to be approved when it should not be.

Two examples follow. First, consider a user that requests their name be "John Doe." A screenshot of such a change appears in figure 14.1. If the user's change additionally added them to the Admin group, the change would appear as in figure 14.2. Note that the difference between figure 14.1 and 14.2 is subtle and could be easily overlooked.

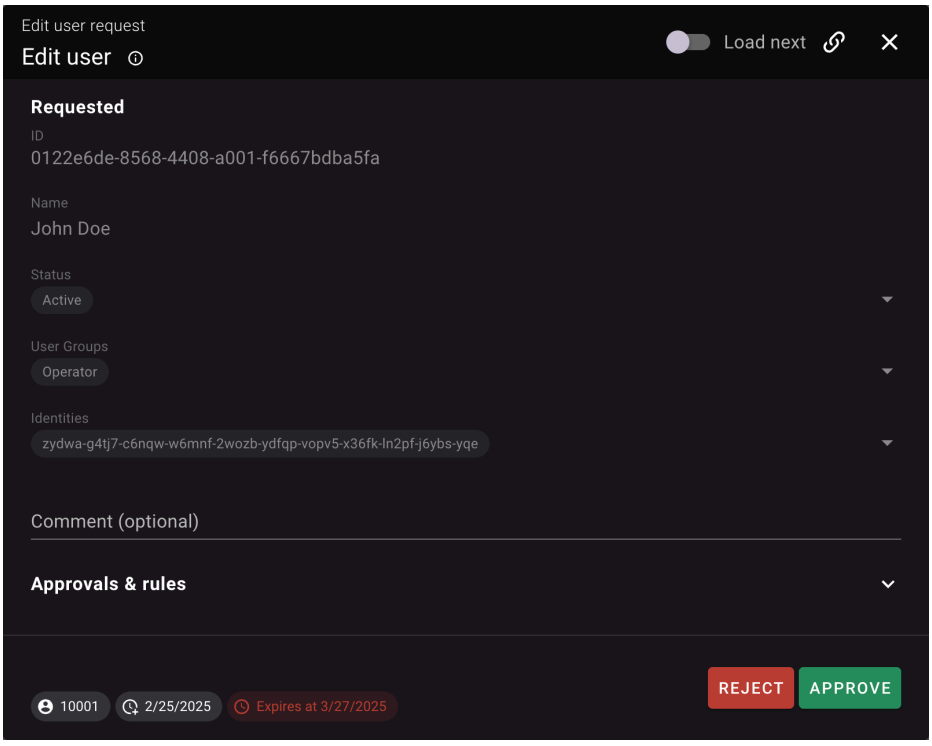

Figure 14.1: Screenshot of change that makes a user's name "John Doe"

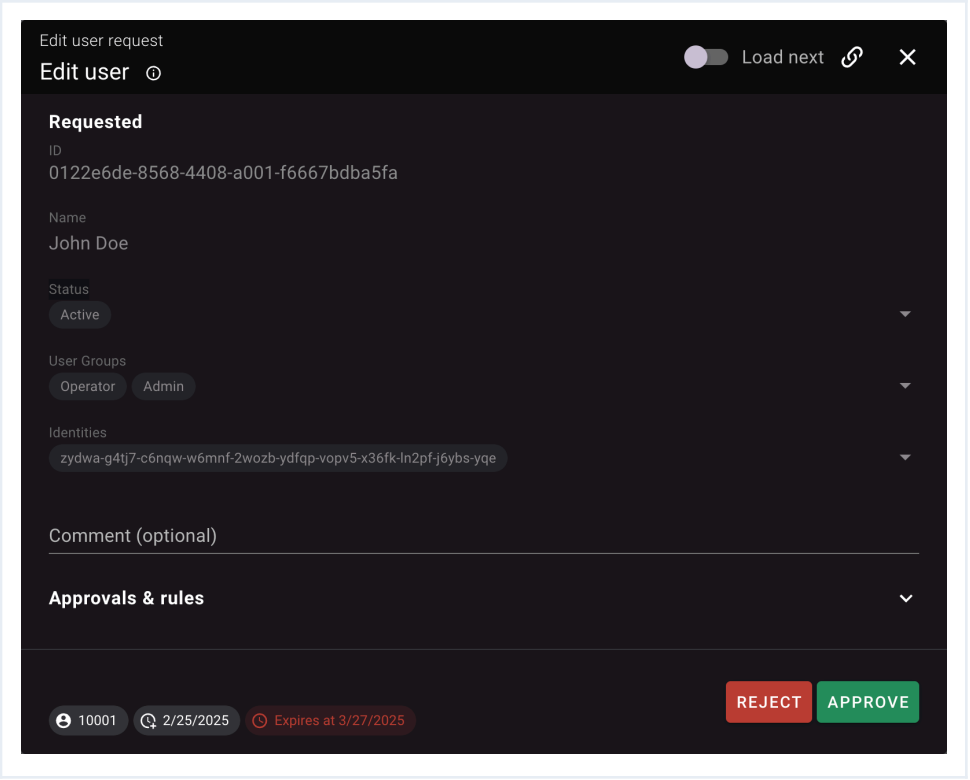

Figure 14.2: Screenshot of change that makes a user's name "John Doe" and that adds them to the `Admin` group

Next, consider a request to change an approval policy. The screenshot in figure 14.3 changes the party whose approval is required from `Admin` to `Operator`. The screenshot in figure 14.4 does the same but also changes the rule's name from "Admin approval" to "Operator approval." The reviewing party might not realize they were looking at what was originally the "Admin approval" rule and think that nothing was amiss.

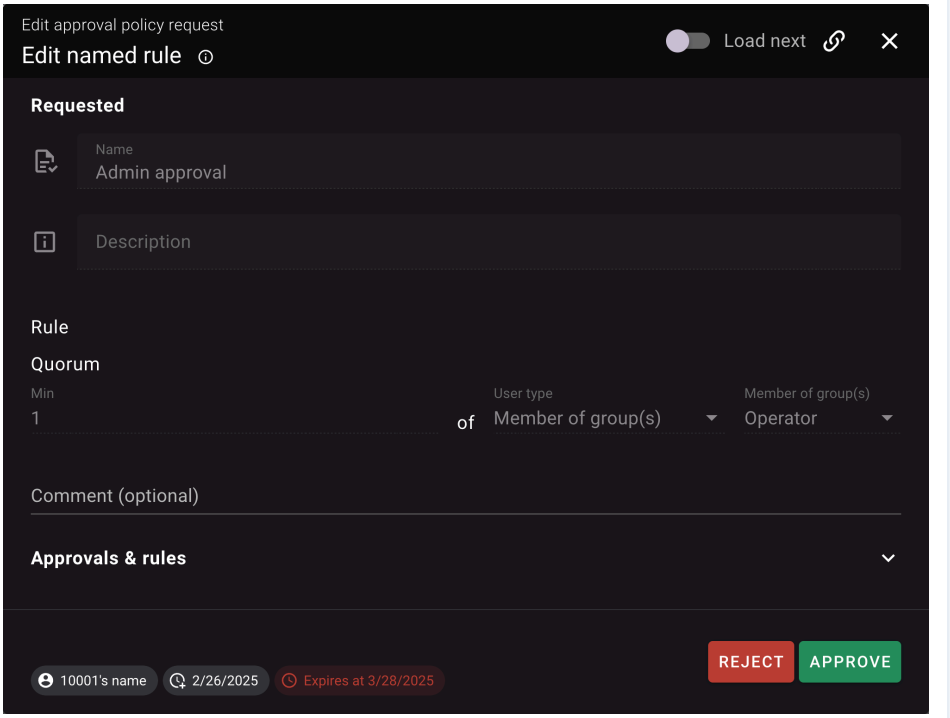

Figure 14.3: Screenshot of change that make a rule's approving group `Operator` instead of `Admin`

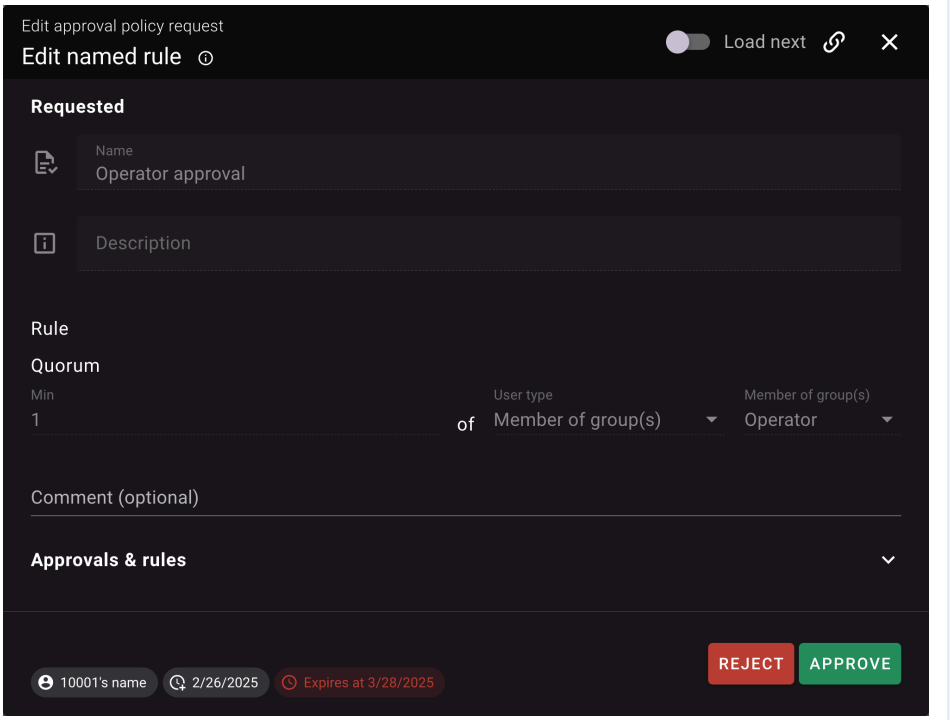

Figure 14.4: Screenshot of change that does the same as figure 14.3 but also changes the rule's name to "Operator approval"

Exploit Scenario

Eve is a malicious `Operator`. Eve tells Alice, the `Admin`, that she is submitting a change to her user record. The change adds Eve to the `Admin` group, which Eve hopes Alice will overlook. Alice does and approves the change, adding Eve to the `Admin` group.

Recommendations

Short term, adjust the web UI so that each update's diff is shown. Doing so will make it less likely that the approving party will overlook an aspect of an update.

Long term, ensure that all elements of the web UI are evaluated by humans. Having standard procedures can help to eliminate some confusing elements, but it cannot eliminate them all. Human evaluation is the most reliable way to identify and eliminate confusing elements.

15. Metadata rules allowed where they are inapplicable

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ORBIT-15
Target: `core/station/impl/src/models/{request_policy_rule.rs, request_specifier.rs}`	

Description

The `RequestPolicyRule` type includes a variant `AllowListedByMetadata` (figure 15.1), which checks the metadata associated with a transfer's target account. When evaluated against an operation that is not a transfer, the rule evaluates to `false`. A misapplied `AllowListedByMetadata` rule could render a wallet unusable. A similar problem seems to exist with `AllowListed` rules.

```
31    #[storable]
32    #[derive(Clone, Debug, PartialEq, Eq, Hash, PartialOrd, Ord)]
33    pub enum RequestPolicyRule {
34        AutoApproved,
35        QuorumPercentage(UserSpecifier, Percentage),
36        Quorum(UserSpecifier, u16),
37        AllowListedByMetadata(MetadataItem),
38        AllowListed,
39        // Logical operators
40        Or(Vec<RequestPolicyRule>),
41        And(Vec<RequestPolicyRule>),
42        Not(Box<RequestPolicyRule>),
43        // Named rule
44        NamedRule(NamedRuleId),
45    }
```

Figure 15.1: The `RequestPolicyRule` enum (orbit/core/station/impl/src/models/request_policy_rule.rs#31–45)

The code to evaluate an `AllowListedByMetadata` rule appears in figures 15.2 and 15.3. As can be seen in the latter figure, when the operation is not a transfer, the rule evaluates to `false`.

```
401    RequestPolicyRule::AllowListedByMetadata(metadata) => {
402        let is_match = self
403            .address_book_metadata_matcher
404            .is_match((request.as_ref().to_owned(), metadata.clone()))?;
405
406        Ok(RequestPolicyRuleResult {
```

```
407             status: if is_match {
408                 EvaluationStatus::Approved
409             } else {
410                 EvaluationStatus::Rejected
411             },
412             evaluated_rule: EvaluatedRequestPolicyRule::AllowListedByMetadata {
413                 metadata: metadata.clone(),
414             },
415         })
416     }
```

Figure 15.2: Excerpt of the `EvaluateRequestPolicyRule::evaluate` function, which evaluates `AllowListedByMetadata` rules (orbit/core/station/impl/src/models/request_policy_rule.rs#401–416)

```
259     impl Match<RequestHasMetadata> for AddressBookMetadataMatcher {
260         fn is_match(&self, v: RequestHasMetadata) -> Result<bool, MatchError> {
261             let (request, metadata) = v;
262
263             Ok(match request.operation.to_owned() {
264                 RequestOperation::Transfer(transfer) => {
                        ...
293                 }
294                 _ => false,
295             })
296         }
297     }
```

Figure 15.3: Excerpt of the definition of `Match::is_match` for `AddressBookMetadataMatcher`, showing that if the operation is not a transfer, the function simply returns `false` (orbit/core/station/impl/src/models/request_specifier.rs#259–297)

To see how such a rule could render a wallet unusable, consider the “Admin approval” policy (figure 15.4). If the policy involved an `AllowListedByMetadata` rule, it would require “disaster recovery” to undo it.

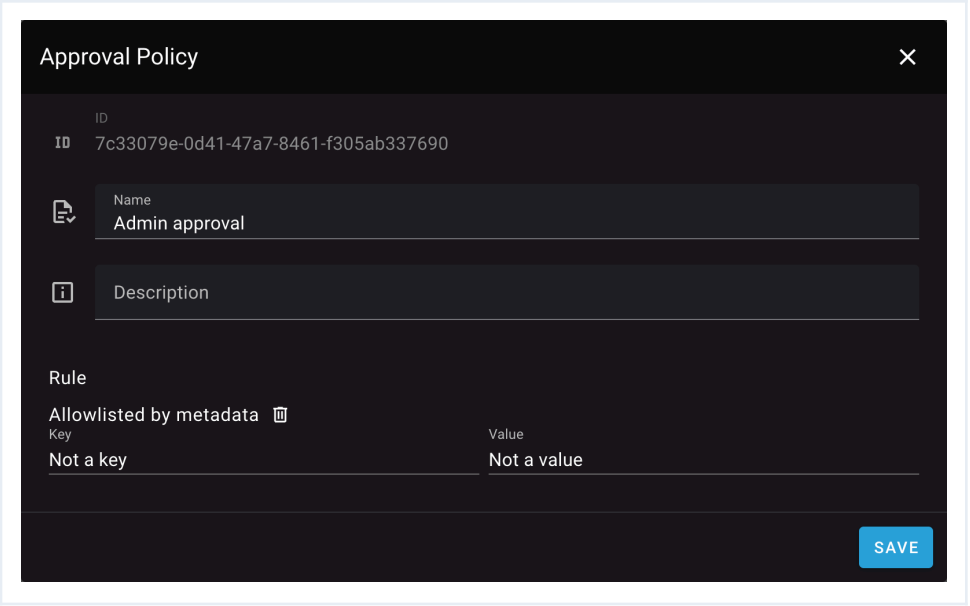

Figure 15.4: Screenshot of the "Admin approval" policy

Exploit Scenario

Alice, an Orbit wallet admin, accidentally changes her wallet's "Admin approval" policy to use an `AllowListedByMetadata` rule. The rule perpetually evaluates to `false` because the relevant operations do not involve transfers. Alice is unable to edit the policy and must perform a "disaster recovery" to correct the error.

Recommendations

Short term, review all uses of the `AllowListedByMetadata` and `AllowListed` rules. Disallow the rules in contexts where they do not make sense (e.g., in "Admin approval" policies). Taking these steps will help prevent users from rendering their wallets unusable.

Long term, develop a programmatic solution, such as the following, to ensure that the `AllowListedByMetadata` and `AllowListed` rules are used only where they are applicable.

- A rule scanner that checks whether a rule is applicable to operations of a certain type at runtime
- Parameterize `RequestPolicyRule` so that transfer-specific rules are allowed only in cases where they are applicable

Implementing such a solution will help to ensure this problem does not reappear.

16. request_recovery silently fails if caller is not a committee member

Severity: **Low**	Difficulty: **Medium**
Type: Error Reporting	Finding ID: TOB-ORBIT-16
Target: `core/upgrader/impl/src/services/disaster_recovery.rs`	

Description

If `request_recovery` is called by someone who is not a committee member, the call will silently fail (figure 16.1). If a committee member issues a call from the wrong account, they may not notice their error.

```
334    pub fn request_recovery(
335        &self,
336        caller: Principal,
337        request: upgrader_api::RequestDisasterRecoveryInput,
338    ) {
339        let mut value = self.storage.get();
340
341        if let Some(committee_member) = self.get_committee_member(caller) {
               ...
369        }
370    }
```

Figure 16.1: Definition of `request_recovery`, showing that if the caller is not a committee member, the call returns without error (orbit/core/upgrader/impl/src/services/disaster_recovery.rs#334–370)

Exploit Scenario

Alice is a member of the disaster recovery committee. Alice calls `request_recovery` but from the wrong account. Because the call succeeds, Alice does not notice her error.

Recommendations

Short term, have `request_recovery` return an error when the caller is not a committee member. Doing so will help unsuccessful recovery requests to be noticed.

Long term, have each function that can be called by a user return a `Result`. Doing so will encourage users to check the results of calls and developers to return errors when operations fail.

17. Asset edit requests can set conflicting or invalid asset metadata

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ORBIT-18
Target: `core/station/impl/src/services/asset.rs`	

Description

Asset edit requests allow users to change asset data such as the asset name, symbol, token standards, and other metadata. This allows privileged users to arbitrarily rename asset types. For example, a user who can edit asset data can swap the asset names for ckBTC to ckETH, which could cause other users to confuse the two assets with each other.

```
if let Some(name) = input.name {
    asset.name = name;
}

if let Some(symbol) = input.symbol {
    asset.symbol = symbol;
}
```

Figure 17.1: Asset names and symbols can be updated without validation. (core/station/impl/src/services/asset.rs#79–85)

The implementation of `ModelValidator::validate` for the `Asset` type will ensure that the symbol is unique among all registered assets on the corresponding blockchain, but it is still possible to swap asset names in multiple steps.

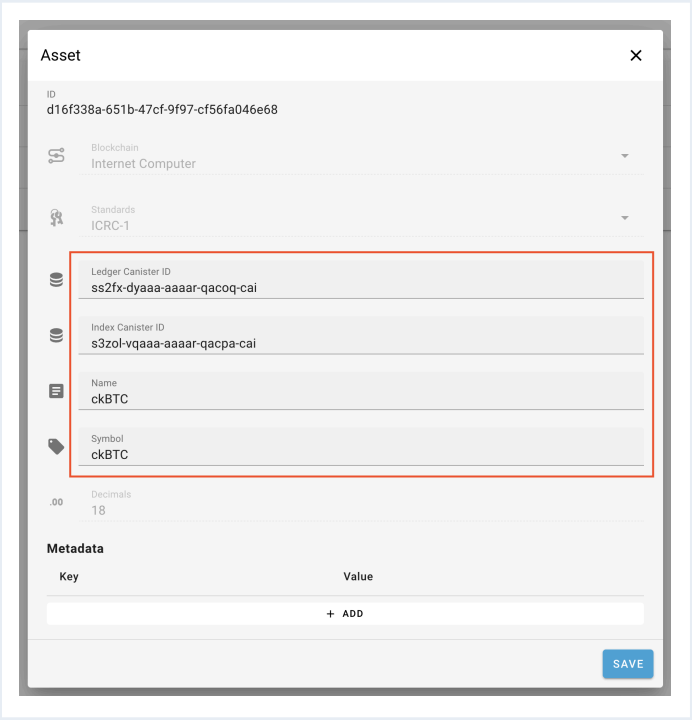

Figure 17.2: Screenshot of the asset UI for ckBTC before a malicious edit

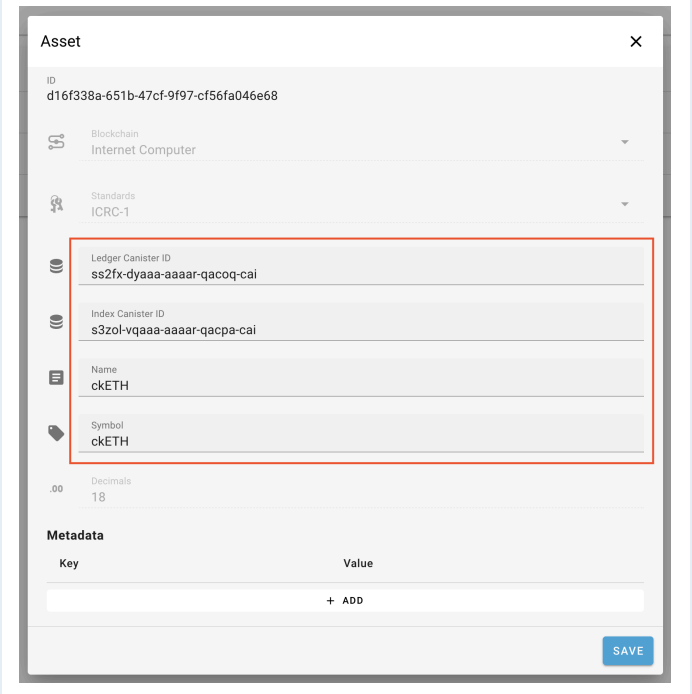

Figure 17.3: Screenshot of the asset UI for ckBTC after a malicious edit

The severity of this issue is rated as high because it could lead to significant financial loss for affected users. On the other hand, the difficulty is also rated as high because the ability to edit asset metadata assumes a high level of privilege within the system.

Exploit Scenario

Eve, a malicious Orbit user, tricks an administrator into swapping the names and symbols of ckBTC and ckETH using the issue described in TOB-ORBIT-14. She then asks Bob, another Orbit user, to transfer one ckETH to an address controlled by Eve. Since the two assets' names and symbols have been swapped, Bob transfers one ckBTC instead of one ckETH to Eve by mistake, losing 91,000 USD in the process.

Recommendations

Short term, prevent all users from editing the assets from the initial list of known assets, and prevent users from introducing custom assets with names or symbols that correspond to known assets. This would protect metadata associated with well-known assets from being manipulated by malicious users.

Long term, consider validating ICRC-1 token metadata against the corresponding ledger to ensure it matches what is reported by the ledger before allowing users to add the asset type to their Orbit accounts. Care must be taken when checking token metadata against third-party ledgers since these can return unexpected or invalid responses.

18. The asset edit API endpoint ignores request expiration times

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ORBIT-19
Target: `core/station/impl/src/factories/requests/edit_asset.rs`	

Description

When a new asset edit request is created in the `EditAssetRequestCreate::create` method, it is automatically populated with the default expiration time (30 days), and the `expiration_dt` field on the `CreateRequestInput` argument is ignored.

```
impl Create<station_api::EditAssetOperationInput> for EditAssetRequestCreate {
    async fn create(
        &self,
        request_id: UUID,
        requested_by_user: UUID,
        input: station_api::CreateRequestInput,
        operation_input: station_api::EditAssetOperationInput,
    ) -> Result<Request, RequestError> {
        let request = Request::new(
            request_id,
            requested_by_user,
            Request::default_expiration_dt_ns(),
            RequestOperation::EditAsset(EditAssetOperation {
                input: operation_input.into(),
            }),
            input
                .execution_plan
                .map(Into::into)
                .unwrap_or(RequestExecutionPlan::Immediate),
            input.title.unwrap_or_else(|| "Edit asset".to_string()),
            input.summary,
        );

        Ok(request)
    }
}
```

Figure 18.1: The request expiration time is ignored when a new asset edit request is created. (core/station/impl/src/factories/requests/edit_asset.rs#13–38)

The same issue is present in the implementation of `RemoveAssetRequestCreate::create`.

Exploit Scenario

Alice sets up an automated script to regularly update asset metadata. She uses a short expiration time to ensure that the updated values are always valid. However, because the request load on the station is high, one asset edit request takes longer to be serviced by the system than expected. Because the request expiration time is set to the default 30 days, when the metadata value is finally updated, it is no longer valid. This causes unexpected issues in Alice's automations built on Orbit.

Recommendations

Short term, use the `Request::from_request_creation_input` method to create requests. This will ensure that the expiration time is set correctly.

Long term, add unit tests to ensure that request inputs are propagated correctly throughout the system.

19. Request approval submission API does not update modification timestamp

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Auditing and Logging	Finding ID: TOB-ORBIT-20
Target: `core/station/impl/src/services/request.rs`	

Description

Submitting a (valid) approval for a request modifies the request and should therefore also update the `last_modification_timestamp` field on the request object. However, this is not done by the `RequestService::submit_request_approval` method handling the request.

```
request.add_approval(approver.id, approval_decision, input.reason)?;

// Must happen after the approval is added to the request to ensure the approval is
counted.
let maybe_evaluation = request.reevaluate().await?;

self.request_repository
    .insert(request.to_key(), request.to_owned());
```

Figure 19.1: The request approval is added but the modification timestamp is not updated. (core/station/impl/src/services/request.rs#462–468)

The request modification timestamp is useful for auditing purposes and should always reflect the last time the request was updated. However, we note that the approval object itself contains a timestamp that can be used to check when the approval was submitted.

Recommendations

Short term, have the `RequestService::submit_request_approval` method update the request modification time after the approval has been added.

Long term, add unit tests to ensure that objects are updated correctly by each API handler.

20. Balance request API silently ignores invalid account IDs

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Error Reporting	Finding ID: TOB-ORBIT-21
Target: `core/station/impl/src/services/account.rs` and `core/station/impl/src/repositories/account.rs`	

Description

The `fetch_account_balances` API takes a list of account IDs as input and returns the balances for each corresponding account. The relevant accounts are fetched from the account repository using the method `AccountRepository::find_by_ids`. This method silently filters out invalid account IDs and returns only the accounts that match the subset of valid account IDs.

```
pub fn find_by_ids(&self, ids: Vec<AccountId>) -> Vec<Account> {
    ids.iter()
        .filter_map(|id| self.get(&Account::key(*id)))
        .collect::<Vec<_>>()
}
```

Figure 20.1: Invalid accounts are silently filtered out in the call to `filter_map`. (`core/station/impl/src/repositories/account.rs#141–145`)

This behavior is inherited by the `fetch_account_balances` API endpoint (implemented by the `AccountService::fetch_account_balances` method), which will silently ignore invalid account IDs. In the extreme case, if a single invalid account ID is passed to the API, it will return an empty list without any error indicating that the given input was invalid.

The same type of issue is present in the `list_users` API endpoint (implemented by the `UserService::list_users` method), which silently ignores invalid group IDs.

Recommendations

Short term, return an error from the `AccountRepository::find_by_ids` method if one of the given account IDs is invalid.

Long term, review all uses of `filter_map` to ensure that invalid inputs are not silently ignored by the Orbit canisters. Add a comment at each call site explaining why filtering out `None` values represents expected behavior.

21. Ad hoc validation of request operation inputs

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-ORBIT-22
Target: `core/station/impl/src/models/request.rs`	

Description

When a request is created, the request operation inputs are validated by the `validate_request_operation_foreign_keys` function. This function validates some but not all of the input fields. As an example, for `RequestOperation::AddAccount` operations, the asset IDs identifying the asset types supported by the account are not checked to ensure that they correspond to assets registered with the system.

```
RequestOperation::AddAccount(op) => {
    op.input.read_permission.validate()?;
    op.input.configs_permission.validate()?;
    op.input.transfer_permission.validate()?;

    if let Some(policy_rule) = &op.input.transfer_request_policy {
        policy_rule.validate()?;
    }

    if let Some(policy_rule) = &op.input.configs_request_policy {
        policy_rule.validate()?;
    }
}
```

Figure 21.1: Request operation input validation checks some fields but disregards others. (core/station/impl/src/models/request.rs#203–215)

Most of these fields will be validated implicitly when the request is executed. However, performing early validation when the request is created allows the system to return an error immediately and ensures that no validation step is skipped by mistake.

Recommendations

Short term, implement the `ModelValidator` trait for each request operation variant and use it to validate request operation inputs.

Long term, prefer early validation to ensure that objects that the system operates on are internally consistent. Implement property tests to ensure that invalid inputs are rejected by the system.

22. Measurable test coverage is low

Severity: **Undetermined**	Difficulty: **High**
Type: Testing	Finding ID: TOB-ORBIT-23
Target: `core`	

Description

Running the test suite using `cargo-llvm-cov` provides coverage statistics for each individual source file in the project. However, because the integration tests run under the `pocket-ic` test runner, there is currently no way to automatically measure test coverage for the integration test suite.

The statistics provided by `cargo-llvm-cov` for the unit test suite indicate that the unit test coverage is generally low and that some files are not covered by the unit tests at all.

core/station/impl/src/repositories/external_canister.rs	62.86% (22/35)	58.92% (142/241)	54.55% (54/99)
core/station/impl/src/repositories/indexes/notification_user_index.rs	85.71% (12/14)	94.94% (75/79)	88.46% (23/26)
core/station/impl/src/repositories/indexes/request_index.rs	100.00% (9/9)	100.00% (77/77)	100.00% (12/12)
core/station/impl/src/repositories/indexes/request_policy_resource_index.rs	80.00% (32/40)	96.55% (448/464)	88.24% (105/119)
core/station/impl/src/repositories/indexes/request_resource_index.rs	100.00% (14/14)	100.00% (73/73)	100.00% (26/26)
core/station/impl/src/repositories/indexes/transfer_account_index.rs	100.00% (14/14)	96.08% (98/102)	91.67% (33/36)
core/station/impl/src/repositories/indexes/transfer_status_index.rs	100.00% (14/14)	100.00% (84/84)	100.00% (26/26)
core/station/impl/src/repositories/indexes/unique_index.rs	73.33% (11/15)	73.40% (69/94)	71.79% (28/39)
core/station/impl/src/repositories/indexes/user_status_group_index.rs	85.71% (12/14)	94.87% (74/78)	91.18% (31/34)
core/station/impl/src/repositories/named_rule.rs	81.25% (26/32)	85.09% (137/161)	76.92% (60/78)
core/station/impl/src/repositories/notification.rs	72.22% (13/18)	54.68% (76/139)	47.27% (26/55)
core/station/impl/src/repositories/permission.rs	81.82% (36/44)	90.74% (294/324)	87.69% (114/130)
core/station/impl/src/repositories/request.rs	92.16% (47/51)	93.10% (648/696)	83.25% (164/197)
core/station/impl/src/repositories/request_evaluation_result.rs	100.00% (4/4)	100.00% (28/28)	100.00% (9/9)
core/station/impl/src/repositories/request_policy.rs	95.65% (22/23)	98.57% (207/210)	96.36% (53/55)
core/station/impl/src/repositories/transfer.rs	100.00% (19/19)	96.82% (152/157)	86.96% (40/46)
core/station/impl/src/repositories/user.rs	80.00% (40/50)	83.52% (223/267)	74.62% (97/130)
core/station/impl/src/repositories/user_group.rs	74.36% (29/39)	76.56% (147/192)	71.58% (68/95)
core/station/impl/src/services/account.rs	63.46% (33/52)	71.70% (1016/1417)	60.81% (239/393)
core/station/impl/src/services/address_book.rs	66.67% (14/21)	85.75% (337/393)	68.00% (68/100)
core/station/impl/src/services/asset.rs	80.77% (21/26)	92.00% (368/400)	75.28% (67/89)
core/station/impl/src/services/change_canister.rs	4.35% (1/23)	1.92% (3/156)	1.06% (1/94)
core/station/impl/src/services/cycle_manager.rs	14.29% (2/14)	11.76% (14/119)	7.89% (3/38)
core/station/impl/src/services/disaster_recovery.rs	58.33% (21/36)	57.19% (187/327)	44.79% (73/163)
core/station/impl/src/services/external_canister.rs	62.89% (61/97)	85.79% (2621/3055)	60.21% (339/563)
core/station/impl/src/services/named_rule.rs	74.07% (20/27)	84.27% (284/337)	73.39% (80/109)
core/station/impl/src/services/notification.rs	70.59% (12/17)	84.02% (142/169)	64.41% (38/59)
core/station/impl/src/services/permission.rs	91.30% (21/23)	97.85% (273/279)	89.47% (85/95)
core/station/impl/src/services/request.rs	68.00% (51/75)	93.18% (1898/2037)	69.47% (248/357)
core/station/impl/src/services/request_policy.rs	92.86% (13/14)	85.51% (183/214)	86.59% (71/82)
core/station/impl/src/services/system.rs	64.86% (24/37)	68.42% (273/399)	59.51% (97/163)
core/station/impl/src/services/transfer.rs	80.95% (17/21)	88.89% (272/306)	61.29% (57/93)
core/station/impl/src/services/user.rs	77.55% (38/49)	87.83% (570/649)	80.00% (172/215)
core/station/impl/src/services/user_group.rs	43.75% (7/16)	57.14% (84/147)	45.59% (31/68)

Figure 22.1: Measurable test coverage is low for many files in the codebase.

The global test coverage across the entire test suite is definitely higher than indicated by the report produced by `cargo-llvm-cov`. However, due to the inability to accurately measure integration test coverage, it is unclear which components lack test coverage and where to invest in more testing. Additionally, ensuring that all the important functions and critical code paths are covered by the test suite would require extensive manual work.

Exploit Scenario

An update to the upgrader inadvertently introduces a bug in a critical function. Since the implementation is not exercised by the unit test suite, the code remains untested and the issue remains undetected. When the new version of the upgrader is deployed across the ecosystem, it breaks the disaster recovery functionality, causing some end users to be permanently locked out of the system. This leads to financial losses for end users and reputational loss for the project.

Recommendations

Short term, work toward improving measurable test coverage for the project.

Long term, introduce randomized testing to further exercise the implementation on unexpected and invalid inputs. This can be done using a mature property testing framework like proptest.

23. validate_dependencies is not recursive

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ORBIT-24
Target: `core/control-panel/impl/src/models/registry_entry.rs`	

Description

The control panel registry has a `validate_dependencies` function, which verifies that a package's dependencies are present in the registry (figure 23.1). However, the function checks only the package's immediate dependencies, not its transitive dependencies. A package could pass validation even though one of its transitive dependencies is not in the registry.

```
331    fn validate_dependencies(entry: &RegistryEntry) ->
ModelValidatorResult<RegistryError> {
332        match &entry.value {
333            RegistryValue::WasmModule(value) => {
                   ...
344                for dependency in value.dependencies.iter() {
345                    validate_wasm_module_version(&dependency.version)?;
346
347                    let found = REGISTRY_REPOSITORY.find_ids_where(
                           ...
353                    );
354
355                    if found.is_empty() {
356                        return Err(RegistryError::ValidationError {
357                            info: format!("The dependency {} is not valid",
dependency.name),
358                        });
359                    }
360                }
361            }
362        }
363
364        // Check for circular dependencies
365        dfs_check_entry_dependencies(entry, &mut HashSet::new())?;
366
367        Ok(())
368    }
```

Figure 23.1: Excerpt of the definition of `validate_dependencies` (orbit/core/control-panel/impl/src/models/registry_entry.rs#331–368)

For comparison, consider the function `dfs_check_entry_dependencies`, appearing near the bottom of figure 23.1. This function checks for circular references among dependencies. Unlike `validate_dependencies`, `dfs_check_entry_dependencies` does check the package's transitive dependencies.

Recommendations

Short term, have `validate_dependencies` verify the existence of a package's transitive dependencies. Doing so will reduce the likelihood that a package is considered valid even though it has missing transitive dependencies.

Long term, try to ensure consistency among APIs, as it can help to expose problems like this. As alluded to above, `dfs_check_entry_dependencies` is called by `validate_dependencies`. However, `dfs_check_entry_dependencies` checks transitive dependencies, whereas `validate_dependencies` does not.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeded industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the repository's root directory.

cargo-llvm-cov

The `cargo-llvm-cov` Cargo plugin is used to generate LLVM source–based code coverage data. The plugin can be installed via the command `cargo install cargo-llvm-cov`. To run the tool, run the command `cargo llvm-cov` in the repository's root directory.

cargo-upgrade

The `cargo-upgrade` Cargo plugin identifies upgradeable dependencies in a project's `Cargo.toml` file. The plugin is part of the `cargo-edit` tool suite and can be installed via the command `cargo install cargo-edit`. To identify incompatible upgrades (i.e., upgrades that `cargo update` would not perform), run the command `cargo upgrade --incompatible` in the repository's root directory.

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a ruleset hosted on the Semgrep registry. To use the default configuration, use the command-line argument `--config auto`. Trail of Bits' public ruleset can be used by running `semgrep --config p/trailofbits`.

ShellCheck

ShellCheck can be installed with common system-wide package installers. For example, `brew install shellcheck` works on macOS, and `sudo apt install shellcheck` works on Ubuntu. To reproduce the results in TOB-ORBIT-9, run `shellcheck scripts/*.sh`.

zizmor

`zizmor` can be installed using Cargo by running `cargo install zizmor --locked`. To analyze the GitHub workflows in the `orbit` repository, navigate to the root directory of the repository and run `zizmor .`. To enable pedantic output, run the tool with the `-p` flag.

D. Non-Security-Related Recommendations

The following recommendations are not associated with any specific vulnerabilities. However, they will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Avoid using shell scripts to run tests.** Currently, the integration tests are run with `scripts/run-integration-tests.sh`, an excerpt of which appears in figure D.1. Ideally, all tests would be run with `cargo test`.

```
31    cd tests/integration
32    echo "PocketIC download starting"
33    curl -sLO https://github.com/dfinity/pocketic/releases/download/7.0.0/pocket-ic-x86_64-$PLATFORM.gz || exit 1
34    gzip -df pocket-ic-x86_64-$PLATFORM.gz
35    mv pocket-ic-x86_64-$PLATFORM pocket-ic
36    export POCKET_IC_BIN="$(pwd)/pocket-ic"
37    chmod +x pocket-ic
38    echo "PocketIC download completed"
39    cd ../..
```

Figure D.1: Excerpt of the script used to run integration tests (`audit-dfinity/scripts/run-integration-tests.sh#31–39`)

- **Regularly run Clippy's `or_fun_call` lint over the codebase.** Currently, the lint produces many warnings, an example of which appears in figure D.2.

```
warning: use of `unwrap_or` followed by a function call
   --> core/station/impl/src/migration.rs:344:30
    |
344 | ...
.unwrap_or(blockchain.generate_account_identifier(&seed)),
    |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ help: try:
`unwrap_or_else(|| blockchain.generate_account_identifier(&seed))`
    |
    = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#or_fun_call
```

Figure D.2: Example warning produced by Clippy's `or_fun_call` lint

- **Fix the inner notification type in `RequestService::failed_request_hook`.** The `failed_request_hook` method sends a notification with inner notification type `RequestRejectedNotification` instead of `RequestFailedNotification`. (However, these both resolve to the same underlying `RequestNotification` type.)

```
pub async fn failed_request_hook(&self, request: &Request) {
```

```
    self.notification_service
        .send_notification(
            request.requested_by,
            NotificationType::RequestFailed(RequestRejectedNotification {
                request_id: request.id,
            }),
            request.title.to_owned(),
            request.summary.to_owned(),
        )
        .await;
}
```

Figure D.3: core/station/impl/src/services/request.rs#374–385

- **Update shell scripts to support older versions of Bash.** The test scripts in the scripts directory use the syntax `OSTYPE = ${OSTYPE,,}` to convert the `OSTYPE` variable to lowercase. This is not supported by earlier versions of Bash. In particular, it is not supported by version 3.2, which ships with macOS. Prefer using `OSTYPE=$(echo "$OSTYPE" | tr '[:upper:]' '[:lower:]')`, which is more portable across different shell versions.

- **Validate metadata keys to ensure they are non-empty.** The `Metadata::validate` method allows keys to be empty. This should never happen, and we recommend checking for this in `Metadata::validate` to prevent user errors.

- **Unify label validation to ensure it is consistent.** The two validation methods `ExternalCanister::validate` and `AddressBookEntry::validate` validate labels differently. The `ExternalCanister` method allows empty labels, but the other does not. On the other hand, the `AddressBookEntry` allows duplicate labels, which the first does not. This logic should be unified to prevent inconsistencies.

- **Remove unused code.** The `Configuration` type defined in `core/station/impl/src/models/configuration.rs` is unused and can be removed.

- **Refactor external canister request policy management.** The code that handles request policies for external canisters is too complex and difficult to review. It would benefit from being refactored into smaller, well-documented components.

- **Regularly run Clippy's `unused_async` lint over the codebase.** There appear to be many functions from which `async` could be removed. An example appears in figure D.4.

```
132     async fn health_status(&self) -> HealthStatus {
133         self.system_service.health_status()
134     }
```

Figure D.4: Example function from which async could be removed (orbit/core/station/impl/src/controllers/system.rs#132–134)

- **Use the imported `upgrader_ic_cdk::caller` function, rather than `ic_cdk::caller`, in the code in figure D.5.** Using `upgrader_ic_cdk::caller` allows the caller to be mocked.

```
137     fn is_committee_member(&self) ->
ApiResult<upgrader_api::IsCommitteeMemberResponse> {
138         let caller = ic_cdk::caller();
            ...
147     }
```

Figure D.5: Use of `ic_cdk::caller` rather than `upgrader_ic_cdk::caller` in `is_committee_member` (orbit/core/upgrader/impl/src/controllers/disaster_recovery.rs#137–147)

- **In the test in figure D.6, check for a specific error rather than the result of `is_err`.** Doing so will help catch unexpected errors.

```
264     #[test]
265     fn unauthorized_callers_cannot_query_state() {
266         DISASTER_RECOVERY_SERVICE
267             .set_committee(mock_committee())
268             .expect("set committee should succeed");
269
270         set_caller(mock_non_committee_member());
271         assert!(CONTROLLER.get_disaster_recovery_accounts().is_err());
272     }
```

Figure D.6: Use of `is_err` in the `unauthorized_callers_cannot_query_state` test (orbit/core/upgrader/impl/src/controllers/disaster_recovery.rs#264–272)

- **Fix the misspelled type name.** The name of the `RequesApprovalRightsRequestPolicyRuleEvaluator` type (defined in `core/station/impl/src/core/request.rs`) is misspelled and should be fixed.
- **Update the following code comment to reflect the correct default expiration time.** The comment should be updated as the default expiration time is 30 days from the creation time.

```
/// Creates a new request adding the caller user as the requester.
///
/// By default, the request has an expiration date of 7 days from the
/// creation date.
```

Figure D.7: core/station/impl/src/services/request.rs#317–319

- **Avoid using an empty string to represent None.** The `SystemInfo::set_name` method uses the input string as an optional value, and if the string is empty, the name is reset to the default value ("Station"). The method should take an argument of type `Option<&str>` instead to make this behavior explicit. Additionally, if an empty name is invalid, the method should return an error on an empty name.

```
pub fn set_name(&mut self, name: String) {
    let mut name = name.trim().to_string();
    if name.is_empty() {
        name = "Station".to_string();
    }

    if name.len() > Self::MAX_NAME_LENGTH {
        name = name.chars().take(Self::MAX_NAME_LENGTH).collect();
    }

    self.name = name;
}
```

Figure D.8: core/station/impl/src/models/system.rs#122–133

- **In the test in figure D.9, `await` the result of the call to `trigger`, as the function is asynchronous.**

```
44    it('wont submit empty form which is invalid', async () => {
45      const wrapper = mount(JoinStation);
46      const sessionStore = useSessionStore();
47
48      const submit = wrapper.find('button[type="submit"]');
49
50      submit.trigger('click');
51      await flushPromises();
52
53      expect(sessionStore.addUserStation).not.toHaveBeenCalled();
54    });
```

Figure D.9: Test with a call that should be awaited (orbit/apps/wallet/src/components/add-station/JoinStation.spec.ts#44–54)

E. Fix Review Results

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

From September 3 to September 5, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the DFINITY team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 23 issues described in this report, DFINITY has resolved eight issues, has partially resolved one issue, and has not resolved the remaining 14 issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Paginated queries may return the wrong list of items	Medium	Resolved
2	Transfer execution function may set an incorrect transfer status	Informational	Unresolved
3	The station accepts invalid transfers	Informational	Unresolved
4	The station canister allows for the creation of invalid assets	Informational	Unresolved
5	Insufficient validation of address book addresses	Informational	Unresolved
6	Request specifier validation does not validate canister IDs	Informational	Resolved
7	Use of old Rust toolchain	Informational	Resolved
8	Outdated and vulnerable dependencies	Informational	Unresolved
9	ShellCheck warnings	Informational	Unresolved
10	Overly broad GitHub workflow permissions	Low	Unresolved

11	Potential credential persistence through GitHub actions artifacts	Informational	Unresolved
12	Unpinned external GitHub CI/CD action versions	Low	Unresolved
13	New requests may arbitrarily delay earlier requests	Low	Unresolved
14	Web UI shows an update's result, not its diff	High	Resolved
15	Metadata rules allowed where they are inapplicable	Medium	Resolved
16	request_recovery silently fails if caller is not a committee member	Low	Resolved
17	Asset edit requests can set conflicting or invalid asset metadata	High	Partially Resolved
18	The asset edit API endpoint ignores request expiration times	Low	Resolved
19	Request approval submission API does not update modification timestamp	Informational	Resolved
20	Balance request API silently ignores invalid account IDs	Informational	Unresolved
21	Ad hoc validation of request operation inputs	Informational	Unresolved
22	Measurable test coverage is low	Undetermined	Unresolved
23	validate_dependencies is not recursive	Informational	Unresolved

Detailed Fix Review Results

TOB-ORBIT-1: Paginated queries may return the wrong list of items
Resolved in PR #571. Overflow checks are now enabled in release builds.

TOB-ORBIT-2: Transfer execution function may set an incorrect transfer status
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-3: The station accepts invalid transfers
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-4: The station canister allows for the creation of invalid assets
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-5: Insufficient validation of address book addresses
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-6: Request specifier validation does not validate canister IDs
Resolved in PR #551. The implementation of the `ModelValidator` trait for `RequestOperation` now ensures that canister IDs correspond to a known external canister for all operations referencing external canisters. The `ModelValidator` trait implementation for `CreateExternalCanisterOperationInput` has also been updated to ensure that canister IDs correspond to known canisters. This PR also adds unit tests to ensure that invalid canister IDs are rejected by the implementation.

TOB-ORBIT-7: Use of old Rust toolchain
Resolved in PR #577. This PR updates the Rust toolchain used to 1.87.

TOB-ORBIT-8: Outdated and vulnerable dependencies
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-9: ShellCheck warnings
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-10: Overly broad GitHub workflow permissions
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-11: Potential credential persistence through GitHub actions artifacts
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-12: Unpinned external GitHub CI/CD action versions
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-13: New requests may arbitrarily delay earlier requests
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-14: Web UI shows an update's result, not its diff
Resolved in PR #566. The UI now shows a diff view that highlights requested changes for edit requests.

TOB-ORBIT-15: Metadata rules allowed where they are inapplicable
Resolved in PR #570. The implementation of the `ModelValidator` trait for the `RequestPolicy` type now ensures that the provided rule is applicable to the given request type. This is done by calling the `validate_rule_for_specifier` function, which (recursively) ensures that the rule (and all sub-rules) is applicable to the given request type.

TOB-ORBIT-16: request_recovery silently fails if caller is not a committee member
Resolved. This is not a security issue since `request_recovery` is an internal method. The API exposed by the canister returns an error as appropriate if the caller is not a member of the disaster recovery committee.

TOB-ORBIT-18: Asset edit requests can set conflicting or invalid asset metadata
Partially resolved. The DFINITY team has decided to accept this risk. The team provided the additional context that exploiting this issue either requires TOB-ORBIT-14 in order to approve an edit request, or some mistake in setting up approval policies. This means that the mitigation for TOB-ORBIT-14 indirectly also mitigates this issue.

TOB-ORBIT-19: The asset edit API endpoint ignores request expiration times
Resolved in PR #575. The `Request::from_request_creation_input` method is now used to create requests to add, edit, and remove assets. This ensures that the request expiration time is set correctly.

TOB-ORBIT-20: Request approval submission API does not update modification timestamp
Resolved in PR #576. The request repository now implements a new method called `save_modified`, which allows the caller to persist a request with an updated modification timestamp. This method is used whenever a request's status is updated by the system.

TOB-ORBIT-21: Balance request API silently ignores invalid account IDs
Unresolved. According to the team, this is done on purpose to provide a better user experience.

TOB-ORBIT-22: Ad hoc validation of request operation inputs
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

TOB-ORBIT-23: Measurable test coverage is low
Unresolved. The DFINITY team comments that this issue is not actionable at the moment since there is no way to obtain test coverage data for PocketIC integration tests.

TOB-ORBIT-24: validate_dependencies is not recursive
Unresolved. The DFINITY team temporarily accepts the risk, but may still fix the issue at a later point.

F. Fix Review Status Categories

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

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

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on X and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.