SEDA Chain Token Migration

Security Assessment

March 29, 2024

Prepared for:
Jasper de Gooijer
SEDA

Prepared by: **Benjamin Samuels and Michael Colburn**

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, as well as 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 Twitter 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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be 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 the controls and security properties of software. 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. Their use is also limited by the time and resource constraints of a project.

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 director was associated with this project:

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

The following consultants were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Project Timeline

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

Date	Event
February 8, 2024	Technical onboarding call
February 22, 2024	Pre-project kickoff call
March 4, 2024	Status update meeting #1
March 11, 2024	Delivery of report draft
March 11, 2024	Report readout meeting
March 29, 2024	Delivery of comprehensive report with fix review appendix

Executive Summary

Engagement Overview

SEDA engaged Trail of Bits to review the security of its token migration contracts and the SEDA chain's staking and vesting modules.

A team of two consultants conducted the review from February 26 to March 8, 2024, for a total of four engineer-weeks of effort. Our testing efforts focused on issues in the token migration process or in the staking or vesting operations in their respective modules. With full access to source code and documentation, we performed static and dynamic testing of the targets, using automated and manual processes.

Observations and Impact

The custom logic for the token migration is straightforward and presents a minimal attack surface in the smart contracts themselves. The migration process was designed to include multiple safeguards such as a relayer allowlist and the ability to pause the migration contracts on either side.

For the SEDA chain codebase itself, the current custom feature set is minimal and includes support only for staking and vesting of the migrated token. Rather than reimplementing this from scratch, much of the code has been adapted from an existing project. Aside from the modifications necessary to adapt the logic, the SEDA team also took a more comprehensive approach to error handling, though this includes some panic statements that seem out of place (TOB-SEDA-2).

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Continue work on simulations and add a fuzz testing suite.** While some work has been done for simulations in SEDA, we recommend continuing this work to help establish a fuzz testing suite for the SEDA chain. Once simulations are working, the native Go fuzzing engine can be added to randomly explore the entire chain lifecycle from genesis to generation of transactions to block creation. This may prove especially useful for testing edge cases with the non-deterministic Overlay network, but since these integrations can often be complex, planning such a test suite should be started early. Some guidelines for establishing such a test harness can be found on our blog: Improving the state of Cosmos fuzzing.

- **Consider adding Cosmos invariants.** As SEDA continues to implement elements of the Overlay network, it may be beneficial to create an invariant registry for the chain to increase its level of security assurance. Cosmos provides support for runtime invariant verification, which can halt the chain or perform other mitigation actions in an emergency if an invariant is violated. This may prove especially useful if the Overlay component has critical liveness considerations that could negatively impact SEDA if they are breached.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	1
Informational	1
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	1
Error Reporting	1

Project Goals

The engagement was scoped to provide a security assessment of the SEDA chain. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can the damage from a potential Wormhole hack be mitigated or limited?
- Can an attacker double-migrate tokens or replay migration requests?
- Does each of the system's privileged components (CosmWasm contract, Solidity migration contract, etc.) use adequate authorization?
- Were any arithmetic or logic issues introduced by the vesting module changes?
- Is it possible to halt the chain by causing a Cosmos module panic?

Project Targets

The engagement involved a review and testing of the targets listed below.

seda-chain

Repository	https://github.com/sedaprotocol/seda-chain/
Version	`8a3e964098f16b37582dfa14ca0d00652c525972`
Type	Go
Platform	Cosmos

token-migration

Repository	https://github.com/sedaprotocol/token-migration/
Version	`aa108d11921802e90faa62a45afd86e710e78e9e`
Type	Rust, Solidity
Platform	CosmWasm, EVM

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:

- **Cosmos staking module**. This module is forked from the mainline Cosmos staking module, with changes that allow token delegations and unbonding delegations to be transferred to other validators. We analyzed the changes to this module using SAST tooling and manually reviewed them for arithmetic and logic issues.
- **Cosmos vesting module**. This module is also forked from the mainline Cosmos staking module, with changes that allow the creator of a vesting contract to "claw back" vesting funds. This clawback code was adapted from code by Agoric to add the ability to claw back continuously vesting funds instead of only periodically vesting funds. We analyzed this module using SAST tooling and manually reviewed it for issues such as the use of vesting modules to bypass locked tokens, unauthorized creation of a vesting account for a target user, and other logic issues.
- **Solidity contract**. The `MigrateFLX` contract allows users to initiate a token migration by burning their tokens on the Ethereum side and signaling this to the Wormhole contract. We manually reviewed this contract to check whether tokens are burned properly and can be recovered, whether the appropriate parameters are passed to Wormhole, and whether the pausing functionality works as expected.
- **CosmWasm contract**. The `cw-seda-token-migration` contract receives messages relayed via the Wormhole protocol and delivers the migrated tokens to the end user. We manually reviewed this contract to check whether the migration logic performs proper validation of the Verified Action Approvals (VAAs) from Wormhole, whether there are any errors in bookkeeping, whether the relayer allowlist can be bypassed, and whether proper access controls are in place for managing the governance operations. We also checked for any issues related to address normalization or other standard CosmWasm issues.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- The SEDA team noted that the `randomness` and `wasm-storage` modules in the `seda-chain` repository were not used outside of tests, so they were considered out of scope for this review.

- The Wormhole Solidity contract, the CosmWasm contract, and the network that bridges messages between the two were out of scope. A compromise of the Wormhole network would lead to a system compromise whose damage would have to be mitigated in a timely manner by the SEDA team.
- SEDA's use of Verifiable Randomness Functions (VRF) was out of scope for this audit.

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 following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	N/A
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time.	Appendix D
Gosec	A static analysis utility that looks for various problems in Go codebases and will identify potentially stored credentials, unhandled errors, cryptographically troubling packages, and similar problems	Appendix D
Staticcheck	A static analysis utility that identifies both stylistic problems and implementation problems within a Go codebase	Appendix D
ineffassign	A static analysis utility that identifies ineffectual assignments, most notably ineffectual error assignments	Appendix D
errcheck	A static analysis utility that identifies situations where errors are not handled appropriately	Appendix D
looppointer	An analyzer that checks for pointers to enclosing loop variables	Appendix D
NilAway	A static analysis tool for detecting nil panics, which can be especially detrimental in a Cosmos application	Appendix D

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	The SEDA chain has relatively little arithmetic in its modified code, with most arithmetic risk limited to operations implemented in third-party components upstream of SEDA's changes.	**Not Applicable**
Auditing	The CosmWasm and Solidity contracts emit events where state mutations are created. The Cosmos modules all emit Prometheus metrics for important messages and state mutations.	**Satisfactory**
Authentication / Access Controls	Each migration contract has adequate access controls implemented for privileged functions, limiting access to a specific owner or governance address. The CosmWasm migration contract has additional authorization for the forwarding of VAA messages from the Wormhole network, creating a defense-in-depth measure that can protect SEDA's token migration in the event of a Wormhole compromise.	**Strong**
Complexity Management	SEDA reuses preexisting components for its vesting, staking, and clawback functionality, reducing the amount of code the team has to maintain and simplifying maintenance. SEDA also has excellent internal documentation and specification, boosting the system's maintainability. However, some functions from the Agoric codebase are excessively large and difficult to understand, with some running 200-plus lines of code. These functions either need to be refactored or have further documentation added to explain what they are doing and their side effects.	**Satisfactory**

Decentralization	SEDA's centralization risks are adequately highlighted in its supporting documentation, and defense-in-depth measures have been added for addressing centralization risks introduced by the Wormhole integration.	**Satisfactory**
Documentation	The token migration process is extensively documented. The specification outlines each of the key components in the system, their features and requirements, details about their implementation, and which party is responsible for their operation.	**Strong**
Low-Level Manipulation	This category was not applicable for this review.	**Not Applicable**
Memory Safety and Error Handling	For the most part, the codebase takes a very comprehensive and defensive approach to error handling. However, there are a few instances in the Cosmos modules where panics unexpectedly occur instead of an error being returned (TOB-SEDA-2).	**Satisfactory**
Testing and Verification	Overall the codebase has good test coverage. The Cosmos code would benefit from the addition of more advanced testing techniques, such as testing invariants and fuzzing. Additionally, the simulations should be completed.	**Satisfactory**
Transaction Ordering	Within the current scope of the system's capabilities, there are no vectors for transaction ordering risks. This may change as functionality becomes available in future versions.	**Strong**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Sensitive material stored in files with loose permissions	Configuration	Low
2	panic() is overused for error management	Error Reporting	Informational

Detailed Findings

1. Sensitive material stored in files with loose permissions	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-SEDA-1
Target: `seda-chain/app/utils/vrf_key.go`, `seda-chain/cmd/sedad/cmd/init.go`, `seda-chain/e2e/validator.go`	

Description

The SEDA node client stores sensitive material using overly loose file or folder permissions of 755 or higher. This includes the VRF key file in figures 1.1 and 1.2, the validator state file in figure 1.3, and the validator configuration file in figure 1.4. The final example is in a file that instruments end-to-end testing and is included for thoroughness.

```
pvKeyFile := config.PrivValidatorKeyFile()
if err := os.MkdirAll(filepath.Dir(pvKeyFile), 0o755); err != nil {
	return nil, fmt.Errorf("could not create directory %q: %w",
filepath.Dir(pvKeyFile), err)
}
```

Figure 1.1: The validator key file is stored in a directory with permission code 755, allowing every user on the system to view the file. (`seda-chain/app/utils/vrf_key.go#229–232`)

```
keyFilePath := config.PrivValidatorKeyFile()
if err := os.MkdirAll(filepath.Dir(keyFilePath), 0o777); err != nil {
	return fmt.Errorf("could not create directory %q: %w",
filepath.Dir(keyFilePath), err)
}
```

Figure 1.2: The validator key file is stored in a directory with permission code 777, allowing every user on the system to view and modify the file. (`seda-chain/cmd/sedad/cmd/init.go#40–43`)

```
stateFilePath := config.PrivValidatorStateFile()
if err := os.MkdirAll(filepath.Dir(stateFilePath), 0o777); err != nil {
	return fmt.Errorf("could not create directory %q: %w",
filepath.Dir(stateFilePath), err)
}
```

Figure 1.3: The validator state file is stored in a directory with permission code 777, allowing every user on the system to view and modify the file. (`seda-chain/cmd/sedad/cmd/init.go#44–47`)

```go
func (v *validator) createConfig() error {
	p := path.Join(v.configDir(), "config")
	return os.MkdirAll(p, 0o755)
}
```

Figure 1.4: The node configuration file is stored in a directory with permission code 755, allowing every user on the system to view the file. (`seda-chain/e2e/validator.go#64–67`)

The code path in figure 1.4 is part of the test suite, not production code.

Exploit Scenario

A validator operator runs a node on a major cloud platform that includes a low-permission logging and metrics daemon on each compute instance used by its clients. Since SEDA's key material is stored in files permissioned so that any user on the system may view their contents, compromise of the logging daemon leads to compromise of the private key.

Recommendations

Short term, modify the created directory permissions to use permission code `0750`, or a more restrictive code if possible. This modification should include less sensitive files such as the configuration and state files because enforcing rules around file permissions is easier if the rules are the same regardless of file content.

Long term, establish internal code quality guidelines that establish the proper methods through which files can be created. This finding was discovered using Gosec, so the SEDA repository may benefit from more regular use of Gosec or from making it a requirement in the repository's CI pipeline.

2. panic() is overused for error management

Severity: **Informational**	Difficulty: **N/A**
Type: Error Reporting	Finding ID: TOB-SEDA-2
Target: Staking module, vesting module	

Description

The SEDA Cosmos modules for staking and vesting use `panic()` statements for certain error cases, as shown in figures 2.1 and 2.2. Go's panic mechanism is not recommended for use in Go applications except for the most fatal errors that require an immediate halt of the application's runtime.

```
maxEntries, err := k.MaxEntries(ctx)
if err != nil {
	panic(err)
}

valSrcAddr, err := sdk.ValAddressFromBech32(toRedelegation.ValidatorSrcAddress)
if err != nil {
	panic(err)
}
valDstAddr, err := sdk.ValAddressFromBech32(toRedelegation.ValidatorDstAddress)
if err != nil {
	panic(err)
}
```

Figure 2.1: The `panic()` statement is used to immediately terminate the application in the staking module. (`seda-chain/x/staking/keeper/keeper.go#62–74`)

```
if !toClawBackStaking.IsZero() {
	panic("failed to claw back full amount")
}
```

Figure 2.2: The `panic()` statement is used to immediately terminate the application in the vesting module. (`seda-chain/x/vesting/keeper/msg_server.go#257–259`)

Exploit Scenario

The code containing `panic()` statements is called by out-of-block code (e.g., during the `BeginBlock` and `EndBlock` functions), and the panic causes the chain to halt.

In the chain's current state, this scenario is not possible, but given SEDA's planned changes, consider avoiding these panic-induced chain halts.

Recommendations

Short term, change uses of `panic()` to normal Go errors where applicable.

Long term, consider banning the use of `panic()` in SEDA's code-contributing guidelines.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, 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
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
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 exceeds 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 is not applicable 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. Code Quality Recommendations

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

- The `seda-chain` codebase contains several "TODO" or "TO-DO" notes. Ideally these should be in an actual issue tracker instead of left in the code itself. This would allow additional context to be included that can explain what exactly needs to be done, as well as what was preventing it from being implemented at the time.

- Split the CosmWasm contract's `execute_transfer_gov` function into two separate functions: `execute_propose_gov` and `execute_accept_gov`. This pattern, similar to what is done in the `Ownable2Step` contract from OpenZeppelin's Solidity library, could help prevent mistakenly transferring governance of the contract to the wrong address.

D. Automated Static Analysis

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

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings. Some static analysis tools provide a high enough signal to be worth integrating into the project's CI/CD pipeline; however, the ideal set of tools tends to vary from project to project.

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`. We used version 1.64.0 of Semgrep. To run Semgrep on the codebase, we ran the following in the root directory of the project:

```
semgrep --config "p/trailofbits" --sarif  --metrics=off --output
semgrep.sarif
```

We also ran the tool with the following rules (configurations):

- `p/ci`
- `p/security-audit`
- `r/go.lang`
- `p/semgrep-go-correctness`

We recommend integrating Semgrep into the project's CI/CD pipeline. Integrate at least the rules with HIGH confidence and with MEDIUM confidence and HIGH impact.

In addition to the four configurations listed above, we recommend using Trail of Bits' set of Semgrep rules (from the repository or less preferably from the registry).

Other Static Analysis Tools

We recommend using the following tools, either in an ad hoc manner or by integrating them into the CI/CD pipeline:

- `golangci-lint`: This tool is a wrapper around various other tools (some but not all of which are listed below). It offers the most effective path to integrating some of the following tools into your CI/CD pipeline because it provides centralized SAST configuration.

- Gosec is a static analysis utility that looks for various problems in Go codebases. Notably, Gosec will identify potentially stored credentials, unhandled errors, cryptographically troubling packages, and similar problems.
- `Go-vet` is a very popular static analysis utility that searches for more Go-specific problems within a codebase, such as mistakes pertaining to closures, marshaling, and unsafe pointers. `Go-vet` is integrated within the `go` command itself, with support for other tools through the `vettool` command line flag.
- Staticcheck is a static analysis utility that identifies both stylistic problems and implementation problems within a Go codebase. Many of the stylistic problems Staticcheck identifies are also indicative of potential problem areas in a project.
- ineffassign is a static analysis utility that identifies ineffectual assignments. These ineffectual assignments often identify situations where errors go unchecked, which could lead to undefined behavior of the program due to execution in an invalid program state.
- errcheck is a static analysis utility that identifies situations where errors are not handled appropriately.
- looppointer is an analyzer that checks for pointers to enclosing loop variables.
- exportloopref is another analyzer that checks for pointers to enclosing loop variables with more accurate results than looppointer but produces false-negatives for certain cases.
- NilAway is a static analysis tool for detecting nil panics.

Please also refer to our blog post on Security assessment techniques for Go projects for further discussion of the Go-related analysis tools.

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.

On March 21, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the SEDA 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 two issues described in this report, SEDA has resolved both of them. For additional information, please see the Detailed Fix Review Results below. The SEDA team also addressed the code quality recommendations in appendix C.

ID	Title	Status
1	Sensitive material stored in files with loose permissions	**Resolved**
2	panic() is overused for error management	**Resolved**

Detailed Fix Review Results

TOB-SEDA-1: Sensitive material stored in files with loose permissions
Resolved in PR #212. The directories are now being created with a more restrictive permission code, `0700`.

TOB-SEDA-2: panic() is overused for error management
Resolved in PR #212. The highlighted panics have been replaced with more descriptive error messages.

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.