Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Scaffold Registry

Veridise Inc.

September 26, 2025

► **Prepared For:**

AhaLabs
https://ahalabs.dev/

► **Prepared By:**

Aayushman Thapa Magar
Evgeniy Shishkin

► **Contact Us:**

contact@veridise.com

► **Version History:**

Nov. 17, 2025	V3
Nov. 17, 2025	V2
Sep. 26, 2025	V1
Sep. 25, 2025	Initial Draft

Contents

1 Executive Summary

From Sep. 26, 2025 to Sep. 22, 2025, AhaLabs engaged Veridise to conduct a security assessment of the Scaffold Registry - a smart contract that enables users to publish and verify versioned contract WASM binaries, deploy them as named instances, manage multiple contract versions, and reuse deployed contracts across decentralized applications. Veridise conducted the assessment over 6 person-days, with 2 security analysts reviewing the project over 3 days on commit `142ee6f`. The review strategy was a manual code review of the program source code performed by Veridise security analysts.

Project Summary. **Scaffold Registry** is an on-chain contract management system for Stellar. It manages versioning of smart contracts identified by unique names and semantic versions. Its core functionality includes:

- Publishing smart contracts under a specific name and version.
- Deploying instances of previously published contracts with user-specified parameters.
- Retrieving contracts by name and version.
- Retrieving deployed instances by name.
- Retrieving the latest version of a contract by name.
- Upgrading deployed instances to newer versions of the corresponding contracts.
- Performing custom upgrades of deployed instances, where supported by the contract.

The Registry enforces syntactic constraints on contract names. Version identifiers must comply with the semantic versioning (SemVer) specification, as implemented in Rust's Cargo package manager.

Code Assessment. The Scaffold Registry developers provided the contracts source code for the code review. The source code appears to be mostly original code written by the Scaffold Registry developers. It contains some documentation in the form of READMEs and documentation comments on functions. The READMEs also contained various information such as use cases, summary, and description, along with video references on how the project is meant to be used.

The source code included a test suite that, as Veridise security analysts noted, was fairly limited in scope and exercised only a small number of execution paths. During the subsequent fix-review phase, the developers significantly expanded the test suite, incorporating all of the recommended improvements.

Summary of Issues Detected. The security assessment uncovered 6 issues, 2 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, (V-AHA-VUL-001) and (V-AHA-VUL-002) where Soroban Maps' lexicographic ordering, rather than semantic versioning, causes "latest" version and hash lookups to select older versions, misleading users and enabling unintended behaviors. The Veridise analysts also identified 2 medium-severity issues, including (V-AHA-VUL-003), where the Registry contract doesn't extend storage TTLs, so entries expire and get archived until manually recovered, as well as 1 low-severity issue, and 1 warning.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Scaffold Registry.

Improve Testing. The Registry component currently includes only basic tests, covering isolated functions such as name validation, version validation, and a minimal workflow (e.g., publishing a single contract). While the Scaffold Registry is not a highly complex protocol, a more rigorous testing approach is recommended. In particular, all critical workflows should be enumerated and accompanied by dedicated test cases that capture both expected (positive) and erroneous (negative) behaviors.

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

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Scaffold Registry	`142ee6f`	Rust	Stellar

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 26–Sep. 22, 2025	Manual & Tools	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	1	1	1
Medium-Severity Issues	2	2	2
Low-Severity Issues	1	1	0
Warning-Severity Issues	1	1	1
TOTAL	6	6	5

Table 2.4: Category Breakdown.

Name	Number
Logic Error	2
Usability Issue	1
Logic Error,Data Validation	1
Access Control	1
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Methodology

The Security Assessment process consists of the following steps:

1. Gather an initial understanding of the protocol's business logic, users, and workflows by reviewing the provided documentation and consulting with the developers.
2. Identify all valuable assets in the protocol.
3. Identify the main workflows for managing these assets.
4. Identify the most significant security risks associated with these assets.
5. Systematically review the codebase for execution paths that could trigger the identified security risks, considering different assumptions.
6. Prioritize one finding over another by assigning a severity level to each.

3.2 Identified Security Risks

After the initial phase of the security assessment was completed, a list of potential security risks was generated. Security analysts used this list during the code review as a starting point to identify potential attack vectors. A few of these risks, when expressed as questions, include the following:

- Can the data of a previously published contract (e.g., name, version, or WASM binary) be altered?
- Can a new version of a contract be published by someone other than its original publisher?
- Can the same WASM binary be published under different versions?
- Can a new version be published with a version number lower than the most recently published one?
- Are there front-running risks associated with contract provisioning, such as name hijacking or initialization?
- Are all access control mechanisms correctly enforced?
- Are there any Soroban-specific implementation issues, such as mismanagement of TTLs or inappropriate use of storage types for variables?

3.3 Scope

The scope of this security assessment is limited to a specific set of source files, as agreed upon with the Scaffold Registry developers.

- `contracts/registry/src/registry/wasm.rs`
- `contracts/registry/src/registry/contract.rs`
- `contracts/registry/src/version.rs`
- `contracts/registry/src/util.rs`
- `contracts/registry/src/registry.rs`
- `contracts/registry/src/name.rs`

- `contracts/registry/src/lib.rs`
- `contracts/registry/src/error.rs`
- `contracts/registry/src/alloc.rs`

3.4 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

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

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

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

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

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

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-AHA-VUL-001	Improper handling of latest WASM version ...	Critical	Fixed
V-AHA-VUL-002	Hash lookup without version parameter ...	High	Fixed
V-AHA-VUL-003	Storage items TTLs not updated in Registry ...	Medium	Fixed
V-AHA-VUL-004	Duplicate wasm binary publication may ...	Medium	Fixed
V-AHA-VUL-005	Potential front-running in Registry ...	Low	Acknowledged
V-AHA-VUL-006	Maintainability issues	Warning	Fixed

4.1 Detailed Description of Issues

4.1.1 V-AHA-VUL-001: Improper handling of latest WASM version lookup

Severity	Critical	**Commit**	142ee6f
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/registry/src/registry/wasm.rs:32-37`		
Confirmed Fix At	56e2035		

Description

The `wasm` module implements the `most_recent_version()` function, intended to return the latest version of a module with a given name. This function is used both to validate WASM updates - ensuring the update has a higher version than the current one - and to provide users with accurate version information.

The implementation is as follows:

```
pub fn most_recent_version(&self, name: &String) -> Result<String, Error> {
    self.registry(name)?
        .keys()
        .first()
        .ok_or(Error::NoSuchVersion)
}
```

The issue lies in how version keys are stored. When inserted into a `Map`, keys are sorted lexicographically. As a result, calling `.first()` does not return the most recent version; instead, it returns the oldest one.

Impact

This behavior causes two problems:

1. **Improper version validation** - An administrator can overwrite an existing WASM module with an older or even malicious one, bypassing update checks.
2. **Incorrect version reporting** - Users may be shown an outdated version as the "latest," leading them to run obsolete code.

Given the central role of versioning in the Registry, this issue has been assigned a Critical severity rating.

Recommendation

Versions must be fetched using the semver order instead of relying on the default ordering of maps.

Developers Response

Developers have fixed the issue at commit `56e2035`.

4.1.2 V-AHA-VUL-002: Hash lookup without version parameter yields non-latest result

Severity	High	**Commit**	142ee6f
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/registry/src/registry/wasm.rs:39-46, 117`		
Confirmed Fix At	56e2035		

Description In Soroban, map keys are ordered lexicographically for strings, which doesn't match semver ordering. For example, under semver `1.10.0` is newer than `1.9.0`, but a Soroban map orders them the other way around, causing hash associated with `1.9.0` to be the last element in the `values` vector.

If no version is provided, the `fetch_hash()` function returns the last value from the Map's values vector. However, this does not always represent the actual latest version, since the values are ordered lexicographically, which does not always match semver ordering.

Impact Users may be shown an outdated version as the "latest," leading them to run or deploy obsolete code, which might contain security vulnerabilities.

Recommendation Fetching versions must follow semver rules instead of the default lexicographic ordering of strings in Maps.

Developers Response Developers have fixed the issue at commit `56e2035`.

4.1.3 V-AHA-VUL-003: Storage items TTLs not updated in Registry contract

Severity	Medium	**Commit**	142ee6f
Type	Usability Issue	**Status**	Fixed
Location(s)	`contracts/registry/src/registry/wasm.rs`		
Confirmed Fix At	56e2035		

Description

In the `Registry` contract implementation, the Time-To-Live (TTL) of individual storage entries is not explicitly updated during contract operation. On Soroban, all storage items have a finite TTL that decreases with ledger progression unless explicitly extended. Once the initial default TTL expires, the associated entries will be archived by the Stellar network.

As a result, the `Registry` contract will not be able to process requests related to these entries once they are archived. Although archived entries can be recovered by users or developers, the recovery process requires additional transactions and coordination. This creates operational overhead and increases the risk that contracts depending on the registry may appear to become unavailable unexpectedly if TTLs are not actively managed.

Impact

- **Loss of Availability:** Contracts referencing registry entries may become unusable after TTL expiration.
- **Operational Overhead:** Recovery of archived data requires additional effort by users or developers, which may involve increased fees and complexity.
- **User Experience Risk:** End users may perceive contracts as broken or unavailable, reducing trust in the system.
- **Ecosystem Impact:** If the registry is a core component relied upon by multiple contracts, widespread TTL expiry could cause cascading failures.

Recommendation

- Implement explicit TTL management within the `Registry` contract by updating or extending the TTL of critical storage entries during normal contract interactions.
- Consider applying a strategy such as refreshing TTLs upon read/write access or introducing a dedicated maintenance function callable by an authorized account or via incentives.
- Document the TTL behavior clearly for users and developers so they understand the lifecycle of stored entries and their responsibilities in maintaining availability.
- Where feasible, use contract design patterns that minimize reliance on long-lived storage or delegate TTL upkeep to automated processes.

Developers Response

Developers have fixed the issue at commit `56e2035`.

4.1.4 V-AHA-VUL-004: Duplicate wasm binary publication may pose security threats

Severity	Medium	**Commit**	142ee6f
Type	Logic Error,Data Validation	**Status**	Fixed
Location(s)	`contracts/registry/src/registry/wasm.rs:83`		
Confirmed Fix At	56e2035		

Description

The `publish()` function of `wasm` module allows the same WASM binary to be published multiple times, attributing it to different versions or authors. The underlying `upload_contract_wasm()` function, invoked by `publish()`, does not reject duplicate binaries; instead, it simply returns the corresponding hash as if the upload were new and successful.

Impact

Allowing repeated publication of identical WASM binaries introduces security risks for the Registry:

1. Users may accidentally or deliberately republish the same flawed WASM binary, perpetuating vulnerabilities.
2. A malicious actor could impersonate another author by republishing their contract, gaining trust from unsuspecting users, and later exploiting this trust by releasing a malicious "next version."

Recommendation

Enforce a restriction that prevents identical binaries from being re-published under new versions or authors. This ensures stronger integrity, avoids redundancy, and reduces opportunities for impersonation and trust exploitation.

Developers Response

Developers have fixed the issue at commit `56e2035`.

4.1.5 V-AHA-VUL-005: Potential front-running in Registry contract publishing

Severity	Low	**Commit**	142ee6f
Type	Access Control	**Status**	Acknowledged
Location(s)	`contracts/registry/src/registry/wasm.rs`		
Confirmed Fix At	N/A		

Description

The `Registry` contract provides a mechanism for associating a specific WASM hash with a `name + version` pair, enabling discoverability and versioning of deployed contracts. However, the current design does not take the publisher (or author) identity into account when creating a record.

This creates a potential front-running vector: if a malicious actor observes a pending transaction to publish a new contract under a given `name + version`, they could submit their own transaction first, occupying the desired slot with a malicious WASM hash. As a result, the intended publisher would be prevented from registering their contract under the expected identifier, and downstream users relying on the registry may unknowingly interact with a malicious implementation.

Impact

- **Hijacking of Namespaces:** Malicious actors could register popular or expected `name + version` combinations before legitimate developers.
- **Malicious Code Injection:** Users querying the registry may retrieve and execute attacker-controlled WASM contracts under trusted names.
- **Loss of Trust:** Developers and users may lose confidence in the registry if naming collisions and hijacks occur.
- **Operational Risk:** Legitimate publishers may be forced to choose alternate identifiers, breaking assumptions or dependencies in their applications.

Recommendation

- Modify the identifier scheme to include the publisher/author address in addition to the `name + version`, forming a triple `(name, version, publisher)`. This ensures uniqueness and prevents one party from occupying another's namespace.
- Provide clear documentation for developers and users to understand the importance of verifying the publisher/author when resolving registry entries.
- Optionally, implement cryptographic signing of published contracts so that clients can validate contract provenance before use.

Developers Response

The developers acknowledged the issue and are considering introducing protocol improvements in the future, but are not planning to provide a fix at the moment.

4.1.6 V-AHA-VUL-006: Maintainability issues

Severity	Warning	**Commit**	142ee6f
Type	Maintainability	**Status**	Fixed
Location(s)	`contracts/registry/src/error.rs`		
Confirmed Fix At	1f6376f		

Description

During the security review, several code constructs were identified that are not currently used anywhere in the codebase:

- In `utils.rs`, the `MAX_BUMP` constant is never referenced.
- In `error.rs`, the following error codes are defined but not utilized: `AlreadyClaimed`, `InitFailed`, `RedeployDeployedFailed`, `NoOwnerSet`.
- In `wasm.rs`, the `new()` function appears to be redundant.

Impact

The presence of unused or redundant code may introduce confusion for maintainers, increase the attack surface, or create uncertainty about intended functionality.

Recommendation

It is recommended to either remove these constructs if they are not needed or document their purpose if they are reserved for future use.

Developers Response

The developers provided a fix for the issue.

Veridise Response

It should be noted that the codebase has undergone a significant refactoring, and the fix included many other changes beyond those mentioned in the Recommendation section. Veridise analysts did their best to ensure that the refactoring did not introduce any unintended behavior.

While reviewing the refactored code, Veridise analysts identified another issue related to an incorrect order of normalization and keyword checking. The issue was fixed in the commit `1f6376f` .