Veridise Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Soroban

Stellar Soroban Core

Veridise Inc.

August 27, 2025

► **Prepared For:**

Stellar Development Foundation
https://stellar.org

► **Prepared By:**

Benjamin Mariano
Shankara Pailoor
Nicholas Brown
Alberto Gonzalez
Burak Kadron

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

► **Version History:**

Aug. 27, 2025	V2.1 (Intended Behavior and Invalid Issues moved to the Appendix)
Dec. 22, 2023	V2
Dec. 22, 2023	V1

Contents

1 Executive Summary

From Oct. 30, 2023 to Dec. 22, 2023, Stellar Development Foundation engaged Veridise to review the security of their Stellar Soroban Core. The review covered their implementation of the Soroban smart contract language on top of the existing Stellar blockchain infrastructure. Veridise conducted the assessment over 35 person-weeks, with 5 engineers reviewing code over 7 weeks. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Stellar Soroban Core developers provided the source code of the Stellar Soroban Core contracts for review. The source code is based on the existing Stellar blockchain core implementation with significant modifications made to support the Soroban language by the Stellar Soroban Core developers. The project has extensive documentation in the form of READMEs, documentation websites, and comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the Stellar Soroban Core developers shared the official documentation for the Soroban language. They also shared multiple references to "CAPs" which contain internal discussion/explanation of certain implementation decisions.

The source code contained a test suite, which the Veridise auditors noted had good coverage of the code overall.

Summary of issues detected. The audit uncovered 7 issues in total. The most severe issue is the use of unchecked arithmetic that can lead to attackers bypassing intended budget limitations (V-SOR-VUL-001). Other issues with budgeting were also found, including both incorrect and misordered metering (V-SOR-VUL-002,V-SOR-VUL-003,V-SOR-VUL-004). In addition to these issues, Veridise auditors also identified maintainability and readability issues in the code (V-SOR-VUL-005, V-SOR-VUL-006, V-SOR-VUL-007).

Some implementation details initially identified as issues have been determined to be intended behavior or invalid after discussions with the developers; these can be found in Appendix A.

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
stellar-core	`0xf2d06fb`	Rust	Stellar
stellar-env	`0x2674d86`	Rust	Stellar
stellar-xdr	`0x16f4d7c`	Rust	Stellar

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 30 - Dec. 22, 2023	Manual & Tools	5	35 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	1	1
Low-Severity Issues	1	0
Warning-Severity Issues	2	0
Informational-Severity Issues	3	0
TOTAL	7	1

Table 2.4: Category Breakdown.

Name	Number
Logic Error	2
Maintainability	3
Incorrect Metering	1
Unchecked Arithmetic	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Stellar Soroban Core's implementation of the Soroban language. In our audit, we sought to answer questions such as:

- Are all operations appropriately metered to avoid Denial of Service attacks on Stellar nodes?
- Are all errors appropriately handled to avoid crashing Stellar nodes?
- Is authentication/authorization implemented appropriately according to the intentions of the developers?
- Can authenticated transactions be subject to replay attacks?
- Are lifetimes for data saved to the blockchain appropriately maintained and enforced?
- Are there any uses of unchecked arithmetic that could overflow?

3.2 Audit Methodology & Scope

Audit Methodology. Our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Dependency Analyzer.* We used the tool `cargo-audit` to audit the source code dependencies for potential security vulnerabilities.
- *Fuzzing/Property-based Testing.* We leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we used cargo-fuzz to (1) update developer provided tests into fuzz tests that test a wider variety of inputs and (2) test invariants/properties we expect to hold over the protocol.

Scope. This audit covered code across there repositories: stellar-core, stellar-env, and stellar-xdr. The scope of the audit was quite large, covering most code in stellar-env, all rust-based code in stellar-core from the folder `src/rust/`, and the folder `src/next/` from stellar-xdr. It should be noted that much of the code from stellar-xdr is actually automatically generated from an internal tool at Stellar, meaning this code was fuzzed but otherwise not manually audited. Additionally, because of the extremely large scope of the audit, auditors focused specially on certain modules, including the authentication and authorization modules (`auth.rs`, `account_contract.rs`), and ledger state handling (`storage.rs`). Finally, it should be noted that our audit focused on the Rust portions of the code. There is also a large portion of the codebase written in C++; for these portions of the codebase, we consulted with developers to understand how those portions of the code interacted with Rust portions of the code.

Methodology. Veridise inspected the provided tests and read the Stellar Soroban Core documentation to get an initial understanding of the code. They then began a manual audit of the code

assisted by automated testing. During the audit, the Veridise auditors met weekly with the Stellar Soroban Core developers to ask questions about the code and communicated over a shared slack channel.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

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

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

Table 3.2: Likelihood Breakdown

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

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

Table 3.3: Impact Breakdown

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

4 Vulnerability Report

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

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SOR-VUL-001	Unchecked Arithmetic in Metered Map	Medium	Fixed
V-SOR-VUL-002	Incorrect Metering When Adding Trackers	Low	Open
V-SOR-VUL-003	Incorrect Metering in new_enforcing	Warning	Open
V-SOR-VUL-004	Metering Happens after Allocation	Warning	Open
V-SOR-VUL-005	Cargo Audit Warnings	Info	Acknowledged
V-SOR-VUL-006	Confusing Naming of AllowanceValue Expiration	Info	Open
V-SOR-VUL-007	Confusing Function Implementation	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-SOR-VUL-001: Unchecked Arithmetic in Metered Map

Severity	Medium	**Commit**	2674d86
Type	Unchecked Arithmetic	**Status**	Fixed
File(s)	`rs-soroban-env/soroban-env-host/src/host/metered_map.rs`		
Location(s)	N/A		
Confirmed Fix At	c29b5a1		

The Soroban Host uses metered data structures to account for computation from smart contracts. One data structure used is called a Metered Map which is a Map for which all the usual operations are augmented to track computation and memory allocations.

However, the arithmetic used to track the resources used by the metered map operations are not checked. In Rust, by default, arithmetic can overflow without throwing an error and by using unchecked arithmetic when tracking resources can potentially allow users to bypass their resource limits via overflow.

Impact By using unchecked arithmetic users could, in principle, bypass their budget limitations by allocating sufficiently large maps and adding to/deleting/resizing existing maps.

The impact is mitigated by the fact that the user must first successfully allocate a map of large enough size to allow the budget to potentially overflow which would cost the user a significant amount in the first place.

Recommendation We recommend changing all arithmetic in the map to use saturating addition or checked addition.

Developer Response The developers have acknowledged this issue and plan to fix it.

4.1.2 V-SOR-VUL-002: Incorrect Metering When Adding Trackers

Severity	Low	**Commit**	2674d86
Type	Logic Error	**Status**	Open
File(s)	`rs-soroban-env/soroban-env-host/src/auth.rs`		
Location(s)	add_invoker_contract_auth()		
Confirmed Fix At	N/A		

The function `add_invoker_contract_auth` undercharges the corresponding computation. In particular, the implementation allocates space for `num_entries` additional trackers but instead charges for `num_entries` `Val` objects. This can be seen in the following code:

```
let auth_entries =
    host.visit_obj(auth_entries, |e: &HostVec| e.to_vec(host.budget_ref()))?;
let mut trackers: std::cell::RefMut<'_, Vec<InvokerContractAuthorizationTracker>> =
    self.try_borrow_invoker_contract_trackers_mut(host)?;
Vec::<Val>::charge_bulk_init_cpy(auth_entries.len() as u64, host)?;
trackers.reserve(auth_entries.len());
for e in auth_entries {
    trackers.push(InvokerContractAuthorizationTracker::new(host, e)?)
}
```

Note that the call to `charge_bulk_init_cpy` is parameterized by `Val` when it should be parameterized by `InvokerContractAuthorizationTracker`.

Impact Since `InvocationContractAuthorizationTrackers` are considerably larger than `Val` the budgeting does not properly account for the amount of computation. In practice the users would be charged a large amount for creating a lot of trackers and so the amount this difference could be used to perform a DOS attack is limited.

Recommendation We recommend changing `Vec::<Val>::charge_bulk_init_cpy(auth_entries.len() as u64, host)?;` to `Vec::<InvokerContractAuthorizationTracker>::charge_bulk_init_cpy(auth_entries.len() as u64, host)?;`

Developer Response This has been acknowledged and a Github Issue has been created.

4.1.3 V-SOR-VUL-003: Incorrect Metering in new_enforcing

Severity	Warning	**Commit**	2674d86
Type	Incorrect Metering	**Status**	Open
File(s)	`rs-soroban-env/soroban-env-host/src/auth.rs`		
Location(s)	new_enforcing()		
Confirmed Fix At	N/A		

The function `new_enforcing` in `auth.rs` allocates a vector of `RefCell<AccountAuthorizationTracker>`s of size `num_entries` but only charges for allocating `num_entries*sizeof(AccountAuthorizationTracker)` memory. Thus, the metering does not account for `sizeof(RefCell)*num_entries` memory which ends up being `8*num_entries` memory since the RefCell includes an additional counter of size `u64` to track references.

Impact The host does not account for `8*num_entries` amount of memory allocated when charging. However, since this is a small constant factor over the existing charge, it shouldn't allow users to perform denial of service attacks.

Recommendation We recommend accounting for the additional memory allocated by charging for `sizeof(RefCell)*num_entries` bytes.

Developer Response "That's a bit of an oversight, but it's really marginal and updating this would be a protocol change. It's okay if we miss a few of bytes here."

4.1.4 V-SOR-VUL-004: Metering Happens after Allocation

Severity	Warning	**Commit**	2674d86
Type	Logic Error	**Status**	Open
File(s)	`rs-soroban-env/soroban-env-host/src/auth.rs`		
Location(s)	snapshot()		
Confirmed Fix At	N/A		

The `snapshot` function in `auth.rs` first allocates a vector of `AccountAuthorizationTrackerSnapshot`s and then charges for this allocation. This is seen in the following code:

```
let len = self.try_borrow_account_trackers(host)?.len();
let mut snapshots: Vec<Option<AccountAuthorizationTrackerSnapshot>> = Vec::
    with_capacity(len);
Vec::<Option<AccountAuthorizationTrackerSnapshot>>::charge_bulk_init_cpy(
                    len as u64, host,
                )?;
```

In general, this pattern of allowing computation prior to charging for it is dangerous because malicious users could perform a denial of service attack by allocating a very large amount of memory before being charged for it. Thus, we recommend that the implementation charge for the vector being allocated before performing the allocation.

Impact While this pattern is dangerous it is unlikely to be exploited in this scenario since the user would need to spend a lot in the first place to allocate a large number of trackers.

Recommendation We recommend swapping the `charge_bulk_init_cpy` call with the allocation of `snapshots`.

Developer Response It doesn't matter much since the capacity is really low, but we can indeed swap these 2 lines. This is not a protocol update.

4.1.5 V-SOR-VUL-005: Cargo Audit Warnings

Severity	Info	**Commit**	16f4d7c
Type	Maintainability	**Status**	Acknowledged
File(s)	rs-stellar-xdr, rs-soroban-env		
Location(s)	N/A		
Confirmed Fix At	N/A		

`cargo audit` gives a warning for the package `bumpalo` due to a potential use after free due to an issue with `Vec::into_iter()` . It also gives warnings for a couple dependencies of the package `textplots` .

Impact Upgrading the dependencies will improve maintainability by avoiding the potential errors caused by these issues.

Recommendation Run `cargo update -p bumpalo` for `rs-stellar-xdr` avoid the risk of this memory error.

Updating the packages `textplots` and `colored` for `rs-soroban-env` will resolve the warnings for this repository.

Both projects build and pass all tests with these updates.

Developer Response For the `bumpalo` dependency, developer's indicated they are not concerned with the issue at this time as it is compile-time. As for `textplots` and `colored`, these are only `[dev-dependencies]` which are used for tests. As a result they will wait to address these concerns at the next "protocol boundary".

4.1.6 V-SOR-VUL-006: Confusing Naming of AllowanceValue Expiration

Severity	Info	**Commit**	2674d86
Type	Maintainability	**Status**	Open
File(s)	`rs-soroban-env/soroban-env-host/src/.../allowance.rs`		
Location(s)	N/A		
Confirmed Fix At	N/A		

Allowances are stored as a `Temporary` ledger entry consisting of an `AllowanceValue` struct declared as the following:

```
#[contracttype]
pub(crate) struct AllowanceValue {
    pub amount: i128,
    pub live_until_ledger: u32,
}
```

The `AllowanceValue.live_until_ledger` is used to determine whether or not the allowance has expired, and this value can be lower than the `TTL` of the ledger entry by calling `write_allowance` with a low value passed as the `live_until` argument or by bumping the `TTL` of the ledger entry.

Since this struct value is called `live_until_ledger` it may be easy for new smart contract developers to confuse this value with the `TTL` value of the ledger entry, which is an unsafe way to enforce the expiration.

Impact If future developers confuse the `live_until_ledger` with the `TTL` of the ledger entry, they may write unsafe code that relies on the `TTL` of the ledger entry which could cause critical vulnerabilities in their code.

Recommendation Clarify the distinction between the allowance expiration and the `TTL` of the AllowanceValue entry by renaming the `live_until_ledger` to something like `allowance_expiration_ledger` , so developers who don't have a thorough understanding of the docs will be less likely to make this mistake.

Also update the params of some of the allowance functions to use `expiration_ledger` instead of `live_until`.

Developer Response TBD

4.1.7 V-SOR-VUL-007: Confusing Function Implementation

Severity	Info	**Commit**	9c53940
Type	Maintainability	**Status**	Acknowledged
File(s)	`rs-soroban-env/soroban-env-common/src/env.rs`		
Location(s)	check_obj_integrity()		
Confirmed Fix At	N/A		

In `Env`, the function `check_val_integrity()` is used to perform validity checks on `Val`s. The function first calls `Val::is_good()` and then calls `check_obj_integrity()` to (presumably) perform additional validity checks on `Val`s corresponding to `Object`s.

The default implementation of `check_obj_integrity()` implementation just returns `Ok(())` without checking anything. This implementation is never used and is replaced in it's only use case in `Host`. As far as auditors can tell, this "dummy" implementation is never used.

```
/// Check that a [`Val`] is good according to the current Env. This is a
/// superset of calling `Val::good` as it also checks that if the `Val` is
/// an [`Object`], that the `Object` is good according to
/// [`Self::check_obj_integrity`].
fn check_val_integrity(&self, val: Val) -> Result<(), Self::Error> {
    if !val.is_good() {
        return Err(self.error_from_error_val(Error::from_type_and_code(
            ScErrorType::Value,
            ScErrorCode::InvalidInput,
        )));
    }
    if let Ok(obj) = Object::try_from(val) {
        self.check_obj_integrity(obj)
    } else {
        Ok(())
    }
}

/// Check that an Object handle is good according to the current Env. For
/// general Val-validity checking one should use Val::good().
fn check_obj_integrity(&self, _obj: Object) -> Result<(), Self::Error> {
    Ok(())
}
```

Furthermore, it should be noted that the docstring for the function `check_val_integrity` references `Val::good` which is not a function in the code base — this should be updated to reference `Val::is_good` which is the name of the function it intends to call. The same is true of the docstring for `check_obj_integrity`.

Impact If future developers use `Env` without reimplementing `check_obj_integrity()`, calling `check_val_integrity()` would just be a wrapper on `Val::is_good()` and would not check the object integrity which contradicts with the docstring in `Env`.

Recommendation We suggest to either mark that `check_obj_integrity()` is a dummy implementation or replace it with just a function definition which needs to be implemented explicitly.

Developer Response Developers pointed out that `Env` is actually implemented in `rs-soroban-guest` via a macro definition (as well as in another repository that was out of scope for this audit), so this "dummy" implementation of `check_obj_integrity` is actually used. The docstring issue is valid and developers intend to fix it.

5 Fuzz Testing

5.1 Methodology

Given the scope of the audit, in addition to extensive manual auditing we also used fuzz testing to get further coverage of the code and to test for particularly worrisome occurances, such as errors that could crash nodes. We additionally created fuzz tests to test basic functional correctness of sub-components.

We used cargo-fuzz to implement the fuzz tests. Some of our tests are expansions of existing unit tests, while some are new tests we devised with the intention of exposing potential crashing bugs. In both cases, it was sometimes required to add additional tooling to construct interesting inputs to the system.

5.2 Properties Fuzzed

Table 5.1 describes the properties we fuzz-tested. The first column links the specification. The second describes the invariant informally in English, and the third shows the total amount of compute time spent fuzzing this property. The last column notes whether we found a bug when fuzzing the invariant (✗ indicates no bug was found and ✓ means fuzzing this invariant revealed a bug).

The Veridise auditors devoted a total of 344 compute-hours to fuzzing this protocol. At this time, no violations have been found via these tests.

Table 5.1: Invariants Fuzzed.

Specification	Invariant	Minutes Fuzzed	Bugs Found
V-SOR-SPEC-001	Address-to-string conversions work as intended	480	✗
V-SOR-SPEC-002	Bytes operations work as intended	480	✗
V-SOR-SPEC-003	Invalid Maps will return an Error	600	✗
V-SOR-SPEC-004	Map has function works as intended	300	✗
V-SOR-SPEC-005	Map insert/delete functions work as intended	600	✗
V-SOR-SPEC-006	Map keys are stored in sorted order	600	✗
V-SOR-SPEC-007	Map keys/values functions work as intended	300	✗
V-SOR-SPEC-008	Non-metered Xdr functions work as intended	4320	✗
V-SOR-SPEC-009	Primitive i64 conversion works as intended	480	✗
V-SOR-SPEC-010	Primitive u64 conversion works as intended	480	✗
V-SOR-SPEC-011	Buffer to Xdr conversions cause no crashes	4320	✗
V-SOR-SPEC-012	Tuple value conversion works as intended	480	✗
V-SOR-SPEC-013	Vector append function works as intended	1440	✗
V-SOR-SPEC-014	Vector back function works as intended	1440	✗
V-SOR-SPEC-015	Vector front function works as intended	1440	✗
V-SOR-SPEC-016	Vector length function works as intended	1440	✗
V-SOR-SPEC-017	Vector pop_back function works as intended	1440	✗
V-SOR-SPEC-018	Vector pop_front function works as intended	1440	✗
V-SOR-SPEC-019	Vector push_back function works as intended	1440	✗
V-SOR-SPEC-020	Vector push_front function works as intended	1440	✗
V-SOR-SPEC-021	Wasmi module works with no crashes	4320	✗

5.3 Detailed Description of Fuzzed Specifications

5.3.1 V-SOR-SPEC-001: Address-to-string conversions work as intended

Minutes Fuzzed	480	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `address.rs` tests which take a fixed list of hexadecimal values between 0 and 255 and convert it into a string using the address conversion functions of the host.

Specification In this case, we test two different features of string-to-address conversion. In the first case, we create a fuzzed address object. We then convert it to a string and back to an address object and assert that the two addresses created are the same. In the second case, we test that invalid address strings always error out (i.e., address strings that are not the correct length).

5.3.2 V-SOR-SPEC-002: Bytes operations work as intended

Minutes Fuzzed	480	Bugs Found	0

Scope We are testing the functions in `rs-soroban-env/soroban-env-host` based on the `bytes.rs` tests.

Specification Here we test a variety of the APIs that are implemented for bytes, such as `bytes_push` and `bytes_len`. For each API, we check that the behavior matches the expected behavior as indicated in the original tests.

5.3.3 V-SOR-SPEC-003: Invalid Maps will return an Error

Minutes Fuzzed	600	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `map.rs` tests, performing the possible operations on the map types.

Specification Verifies that any map initialized with duplicate keys, initialized with keys that are not in ascending order, or containing an invalid value will return an error.

5.3.4 V-SOR-SPEC-004: Map has function works as intended

Minutes Fuzzed	300	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `map.rs` tests, performing the possible operations on the map types.

Specification Verify that if an `ScMap` is constructed correctly with some initial values, `map_has` will correctly determine whether or not a key is present in the Map.

5.3.5 V-SOR-SPEC-005: Map insert/delete functions work as intended

Minutes Fuzzed	600	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `map.rs` tests, performing the possible operations on the map types.

Specification Verifies that calling `map_put()` and `map_del()` on any properly constructed mapping will correctly insert and delete elements respectively. We check that calling `map_put` with a key not present in the map will insert the value at the new key and that calling `map_get` with a key present in the map will remove that entry from the map.

5.3.6 V-SOR-SPEC-006: Map keys are stored in sorted order

Minutes Fuzzed	600	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `map.rs` tests, performing the possible operations on the map types.

Specification To ensure determinism when accessing map elements by their index, verify that map entries are stored in sorted order based on their keys. We check that integer keys are stored in ascending order of the value of the integer and that keys of different types are sorted by the pre-defined ordering of the different types.

5.3.7 V-SOR-SPEC-007: Map keys/values functions work as intended

Minutes Fuzzed	300	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `map.rs` tests, performing the possible operations on the map types.

Specification Verifies that calling `map_keys()` and `map_values()` on any correctly constructed map will return the keys and the values inside the map respectively.

5.3.8 V-SOR-SPEC-008: Non-metered Xdr functions work as intended

Minutes Fuzzed	4320	Bugs Found	0

Scope The scope of the tests include the non-metered functions (`non_metered_xdr_to_rust_buf()` and `non_metered_xdr_from_cxx_buf()`) in `stellar-core` module.

Specification In stellar-core, we are testing the XDR conversion in non-metered functions (`non_metered_xdr_to_rust_buf()` and `non_metered_xdr_from_cxx_buf()`) by generating random `ScVal` objects defined in `generated.rs` and testing to see whether the read/write depth limits are enforced and whether the objects pre and post conversion are the same.

5.3.9 V-SOR-SPEC-009: Primitive i64 conversion works as intended

Minutes Fuzzed	480	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `basic.rs` tests which take a fixed primitive value (like u64, i64, etc.), convert it into a host value, convert it back and check equality.

Specification Given any integer within a set of i64 integers, converting it into a host value and converting it back does not change its value.

5.3.10 V-SOR-SPEC-010: Primitive u64 conversion works as intended

Minutes Fuzzed	480	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `basic.rs` tests which take a fixed primitive value (like u64, i64), convert it into a host value, convert it back and check equality.

Specification Given any integer within a set of u64 integers, converting it into a host value and converting it back does not change its value.

5.3.11 V-SOR-SPEC-011: Buffer to Xdr conversions cause no crashes

Minutes Fuzzed	4320	Bugs Found	0

Scope The scope of tests on this specification include buffer-to-xdr conversion function `non_metered_xdr_from_cxx_buf()` in `stellar-core` module.

Specification In `stellar-core`, we are testing to see if converting random cxxbuf objects (which are random arrays containing u8 values) to random objects would cause any crashes on `non_metered_xdr_from_cxx_buf()` functions.

5.3.12 V-SOR-SPEC-012: Tuple value conversion works as intended

Minutes Fuzzed	480	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the `basic.rs` tests which take a fixed primitive value (like u64, i64), convert it into a host value, convert it back and check equality.

Specification Given any tuple containing u64 and i64 integers, converting it into a host value and converting it back does not change its value.

5.3.13 V-SOR-SPEC-013: Vector append function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given two random vectors, `vec_append()` appends two vectors accurately. In our fuzz test, we check that the length of this new vector is equal to the sum of the length of two vectors; and ordering of the elements is the same. For example, we check that the first vector with the length n matches with the first n elements of the new vector and the rest matches with the second vector.

5.3.14 V-SOR-SPEC-014: Vector back function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_back()` function accurately returns the last element of the vector. In our fuzz test, we have assertions that check the equality of the last element of the random Rust vector and the result of the `vec_back()` of the host vector object converted from the random vector.

5.3.15 V-SOR-SPEC-015: Vector front function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_front()`, returns the first element of the vector accurately. In our fuzz test, we have assertions that check the equality of the first element of the random Rust vector and the result of the `vec_front()` of the host vector object converted from the random vector.

5.3.16 V-SOR-SPEC-016: Vector length function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_len()` function, which checks a given vector's length returns the random vector's length accurately. In our fuzz test, we have assertions that check the equality of the length of the random Rust vector and the length of the host vector object converted from the random vector.

5.3.17 V-SOR-SPEC-017: Vector pop_back function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_pop_back()` removes the last element of the vector accurately. In our fuzz test, we check that the length of the vector has decreased by one and the list is not altered in any way except the last element.

5.3.18 V-SOR-SPEC-018: Vector pop_front function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_pop_front()` removes the first element of the vector accurately. In our fuzz test, we check that the length of the vector has decreased by one and the list is not altered in any way except the first element.

5.3.19 V-SOR-SPEC-019: Vector push_back function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_push_back()` adds one element to the back of the vector (making that element the last element) accurately. In our fuzz test, we push a random value to the vector and check equality of the last value of the host vector object and the random value and check that the length of the vector has increased by one.

5.3.20 V-SOR-SPEC-020: Vector push_front function works as intended

Minutes Fuzzed	1440	Bugs Found	0

Scope We are testing type conversion functions in `rs-soroban-env/soroban-env-host` based on the tests in `vec.rs` which perform all the possible operations on vector types (e.g. `len()`, `pop()`, `push()`, etc.)

Specification Given a random vector, `vec_push_front()` adds one element to the front of the vector (making that element the first element) accurately. In our fuzz test, we push a random value to the vector and check equality of the first value of the host vector object and the random value and check that the length of the vector has increased by one.

5.3.21 V-SOR-SPEC-021: Wasmi module works with no crashes

Minutes Fuzzed	4320	Bugs Found	0

Scope We are testing the `wasmi` module which is used by `soroban-env-rs` repository. `wasmi` module provides an interpreter to run Web Assembly (WASM) code and as Soroban depends on `wasmi` as an interpreter, we test it with random Web Assembly code.

Specification In our fuzz test, we are using `wasm-smith` module which generates random Web Assembly modules based on some parameters (number of imports, whether to use reference types, number of instructions, etc.) and we run the functions generated by `wasm-smith` with random inputs (either primitive values or random function references) over `wasmi` and check if the interpreter crashes at any step of the run.

Appendix A

In addition to the confirmed findings, this report includes an appendix with issues that were ultimately classified as intended behavior or invalid after discussions with the development team. We chose to include these items in the report for two main reasons:

- **Transparency.** To document the scope of our review and ensure that all potential concerns identified during the audit are visible to stakeholders.
- **Context.** To provide the development team with a clear record of which findings were analyzed, discussed, and intentionally dismissed, avoiding future confusion if similar questions arise.

A.1 Intended Behavior: Non-Issues of Note

A.1.1 V-SOR-APP-VUL-001: Possible Unmetered Clones

Severity	Warning	**Commit**	2674d86
Type	Incorrect Metering	**Status**	Intended Behavior
File(s)	`rs-soroban-env/soroban-env-host/src/e2e_invoke.rs`		
Location(s)	build_storage_map_from_xdr_ledger_entries()		
Confirmed Fix At	N/A		

In the `e2e_invoke.rs` file there are some cloning operations that are metered. For example, in the `build_storage_footprint_from_xdr` function, the cloning of the `key` variable of type `LedgerKey` is metered:

```
Rc::metered_new(key.metered_clone(budget)?, budget)?
```

Snippet A.1: Code snippet from the `build_storage_footprint_from_xdr`.

Whereas other clone operations in the same file are not metered. For example, in the `build_storage_map_from_xdr_ledger_entries` function, the cloning of the `key` variable of type `Rc<LedgerKey>` is not metered (other cloning of references are not metered either):

```
ttl_map = ttl_map.insert(key.clone(), ee, budget)?;
```

Snippet A.2: Code snippet from the `build_storage_map_from_xdr_ledger_entries`.

Also in the `invoke_host_function` function, the cloning of the `budget` variable of type `Budget` is not metered:

```
let host = Host::with_storage_and_budget(storage, budget.clone());
```

Snippet A.3: Code snippet from the `invoke_host_function`.

Impact The cost of non-metered operations is paid by nodes and not users.

Recommendation Either meter the operations or document in the code why metering them is not necessary.

Developer Response Many operations in the code are non-metered even when they are operations which are being run on a node during enforcement time. The developers chose to make some small constant cost operations (such as a clone of a reference cell) unmetered, as the cost of metering would be higher than it is worth and it should not be possible (or at least not practical) to exploit these cheap unmetered operations to perform any real attack.

A.2 Invalid Issues

A.2.1 V-SOR-APP-VUL-002: Ledger Entries Deleted Arbitrarily

Severity	Critical	Commit	2674d86
Type	Logic Error	Status	Investigated
File(s)	rs-soroban-env/soroban-env-host/src/e2e_invoke.rs		
Location(s)	build_storage_footprint_from_xdr()		
Confirmed Fix At	N/A		

In `InvokeHostFunctionOpFrame.cpp`, we find the following code:

```
// Erase every entry not returned.
// NB: The entries that haven't been touched are passed through
// from host, so this should never result in removing an entry
// that hasn't been removed by host explicitly.
for (auto const& lk : footprint.readWrite) {
    if (createdAndModifiedKeys.find(lk) == createdAndModifiedKeys.end())
        {
            auto ltxe = ltx.load(lk);
            if (ltxe)
            {
                releaseAssertOrThrow(isSorobanEntry(lk));
                ltx.erase(lk);

                // Also delete associated TTLEntry
                auto ttlLK = getTTLKey(lk);
                auto ttlLtxe = ltx.load(ttlLK);
                releaseAssertOrThrow(ttlLtxe);
                ltx.erase(ttlLK);
            }
        }
  }
```

In this code, if a `LedgerKey` was declared in the `footprint.readWrite` but was not returned by the host as a modified key, then the code deletes the entry.

The array of modified `LedgerKeys` returned by Soroban is built in `contract.rs`, specifically in the `extract_ledger_changes` function, where we can see the folllowing:

```
for change in entry_changes {
    // Extract ContractCode and ContractData entry changes first
    if !change.read_only {

        if let Some(encoded_new_value) = change.encoded_new_value {
            modified_entries.push(encoded_new_value.into());
        }
     }
```

In this code, if the entry change is `readOnly` then the entry is NOT appended to the `modified_entries` vector. The `entry_changes` vector is built in the `e2e_invoke.rs` file in the `get_ledger_changes` function, where at the end we can see the following:


```
let maybe_access_type: Option<AccessType> =
      footprint_map.get::<Rc<LedgerKey>>(key, budget)?.copied();

match maybe_access_type {
    Some(AccessType::ReadOnly) => {
       entry_change.read_only = true;
    }

        Some(AccessType::ReadWrite) => {
      if let Some((entry, _)) = entry_with_live_until_ledger {
         let mut entry_buf = vec![];
         metered_write_xdr(budget, entry.as_ref(), &mut entry_buf)?;
         entry_change.encoded_new_value = Some(entry_buf);
      }
    }

        None => {
        return Err(internal_error);
      }
}

changes.push(entry_change);
```

If the `key` in the `footprint_map` has an `AccessType` of `ReadOnly` then the `read_only` flag is set to true.

Now, we have the prerequisites needed to carry the attack of deleting a ledger entry with a LedgerKey `key`:

- The `footprint.readWrite` array in the context of the `InvokeHostFunctionOpFrame.cpp` code must contain the `key`.
- The `footprint_map` in the context of the `e2e_invoke.rs` code must contain the `key` but with an access type of `readOnly`.

To better understand, consider the `footprint_map` built using `build_storage_footprint_from_xdr` in `e2e_invoke.rs`:

```
fn build_storage_footprint_from_xdr(
    budget: &Budget,
    footprint: LedgerFootprint,
) -> Result<Footprint, HostError> {
    let mut footprint_map = FootprintMap::new();

    for key in footprint.read_write.as_vec() {
        Storage::check_supported_ledger_key_type(&key)?;
        footprint_map = footprint_map.insert(
            Rc::metered_new(key.metered_clone(budget)?, budget)?,
            AccessType::ReadWrite,
            budget,
        )?;
    }

    for key in footprint.read_only.as_vec() {
        Storage::check_supported_ledger_key_type(&key)?;
```

```
        footprint_map = footprint_map.insert(
            Rc::metered_new(key.metered_clone(budget)?, budget)?,
            AccessType::ReadOnly,
            budget,
        )?;
    }

    Ok(Footprint(footprint_map))
}
```

Basically, it makes a for loop over `footprint.readWrite` and `footprint.read_only` and inserts them in the map. Because the `read_write` is processed before `read_only` if they both contain a repeated `LedgerKey` then the `ReadOnly` access type will be in the final `footprint_map`.

Impact An arbitrary Ledger Entry can be deleted from the ledger.

Recommendation Process first the `read_only` vector in the `build_storage_footprint_from_xdr` function and then the `read_write`.

Why Invalid? Given the severity, we tried to confirm the attack with the following test:

```
var SorobanClient = require('soroban-client');
var server = new SorobanClient.Server('https://soroban-testnet.stellar.org');

// Contract already deployed to the Testnet.
// It declares two LedgerKeys:
//         - One with the symbol COUNTER
//         - The other with symbol ACOUNTER.
// It defines two functions each one to increment one of the counters.
let scAddress = new SorobanClient.Contract('
    CANUDSWUO7COOK7MUNADD4UQGSSOAFJJE2DFA7ETKXEHYQNZO7WILDEH').address().toScAddress
    ();

// The ACOUNTER key is the one we want to delete.
let key_a = SorobanClient.xdr.ScVal.scvSymbol("ACOUNTER");

let xdrDurability = SorobanClient.xdr.ContractDataDurability.persistent();

// Build the LedgerKey.
let contractKey2 = SorobanClient.xdr.LedgerKey.contractData(
    new SorobanClient.xdr.LedgerKeyContractData({
      key: key_a,
      contract: scAddress,
      durability: xdrDurability
    })
);

async function sendTransaction() {
  // Create an account and fund it.
  var keypair = SorobanClient.Keypair.random();
  var address = keypair.publicKey();
```



```
29
30  await server.requestAirdrop(address)
31
32  const account = await server.getAccount(address);
33
34  // Fee hardcoded for this example.
35  const fee = 200;
36
37  const contract = new SorobanClient.Contract('
      CANUDSWUO7COOK7MUNADD4UQGSSOAFJJE2DFA7ETKXEHYQNZO7WILDEH');
38
39  // Build the transaction. The idea is to increment the value in the Ledger Entry
      given by the key with symbol COUNTER.
40  let transaction = new SorobanClient.TransactionBuilder(account, { fee,
      networkPassphrase: SorobanClient.Networks.TESTNET })
41          .addOperation(
42              // An operation to call increment on the contract
43              contract.call("increment_counter")
44          )
45          .setTimeout(30)
46          .build();
47
48  // Simulate the transaction to discover the storage footprint, and update the
49  // transaction to include it.
50  // This will include the necessary footprint to carry on the operation of '
      increment_counte'.
51  transaction = await server.prepareTransaction(transaction);
52
53  let envelope = transaction.toEnvelope();
54
55  let txFootprint = envelope._value._attributes.tx._attributes.ext._value._attributes
      .resources._attributes.footprint;
56  console.log('Original Footprint: ')
57  console.log(txFootprint._attributes.readOnly);
58  console.log(txFootprint._attributes.readWrite);
59
60  let newEnvelope = envelope;
61
62  // To carry-on the attack we append to the previous footprint the key of the entry
      we want to delete. We include it in both the
63  // readOnly footprint and the readWrite footprint.
64  newEnvelope._value._attributes.tx._attributes.ext._value._attributes.resources.
      _attributes.footprint._attributes.readWrite = newEnvelope._value._attributes.tx.
      _attributes.ext._value._attributes.resources._attributes.footprint._attributes.
      readWrite.concat([contractKey2]);
65  newEnvelope._value._attributes.tx._attributes.ext._value._attributes.resources.
      _attributes.footprint._attributes.readOnly = newEnvelope._value._attributes.tx.
      _attributes.ext._value._attributes.resources._attributes.footprint._attributes.
      readOnly.concat([contractKey2]);
66
67  let newFootprint = newEnvelope._value._attributes.tx._attributes.ext._value.
      _attributes.resources._attributes.footprint;
68  console.log('New Footprint: ')
```

```
  console.log(newFootprint._attributes.readOnly);
  console.log(newFootprint._attributes.readWrite);

  // Create the new tx with the new created footprint.
  let newTx = new SorobanClient.Transaction(newEnvelope, SorobanClient.Networks.
    TESTNET);
  // sign the transaction
  newTx.sign(keypair);

  try {
    const transactionResult = await server.sendTransaction(newTx);
    //console.log(transactionResult.errorResult._attributes.result);//.errorResult.
    _attributes.result);
    console.log(transactionResult.errorResult._attributes.result);//errorResult.
    _attributes.result);//.errorResult._attributes.result);
  } catch (err) {
     console.error(err);
  }
}

sendTransaction()
```

It failed and show the error `txMalformed`. In the `TransactionFrame.cpp` we can find the following checking that `footprint.readOnly` and `footprint.readWrite` have no duplicates within and between them:

```
// check for duplicates
UnorderedSet<LedgerKey> set;
auto checkDuplicates =
        [&](xdr::xvector<stellar::LedgerKey> const& keys) -> bool {
            for (auto const& lk : keys)
        {
            if (!set.emplace(lk).second)
            {
              getResult().result.code(txMALFORMED);
              return false;
            }
        }
        return true;
        };

        if (!checkDuplicates(sorobanData.resources.footprint.readOnly) ||
            !checkDuplicates(sorobanData.resources.footprint.readWrite))
        {
            return false;
        }
```


A.2.2 V-SOR-APP-VUL-003: Denial of Service During Authorization

Severity	Critical	**Commit**	2674d86
Type	Denial of Service	**Status**	Investigated
File(s)	`rs-soroban-env/src/auth.rs`		
Location(s)	require_auth_enforcing()		
Confirmed Fix At	N/A		

In Soroban, contracts can request authorization for requests by invoking `request_auth` on an address. If that address corresponds to a contract, the authorization module will invoke the `__check_auth` function of that module which is responsible for accepting or rejecting the requested authorization.

As indicated in the documentation and related examples (such as this), it is usually frowned upon for the `__check_auth` function to call `require_auth` on its own address, as this can lead to infinite recursion. The concern here is that an attacker can abuse this frowned upon behavior to cause the algorithm to spent a disproportionate amount of time evaluating authorizations, very little of which is metered to the caller.

In particular, suppose there is a contract whose implementation of `__check_auth` simply calls `require_auth` on its own address. This will lead to repeated invocations of the `require_auth_enforcing` function, which has the following loop:

```
for tracker in self.try_borrow_account_trackers(host)?.iter() {
    // Tracker can only be borrowed by the authorization manager itself.
    // The only scenario in which re-borrow might occur is when
    // `require_auth` is called within `__check_auth` call. The tracker
    // that called `__check_auth` would be already borrowed in such
    // scenario.
    // We allow such call patterns in general, but we don't allow using
    // tracker to verify auth for itself, i.e. we don't allow something
    // like address.require_auth()->address_contract.__check_auth()
    // ->address.require_auth(). Thus we simply skip the trackers that
    // have already been borrowed.
    if let Ok(mut tracker) = tracker.try_borrow_mut() {
        // If tracker has already been used for this frame or the address
        // doesn't match, just skip the tracker.
        if !host.compare(&tracker.address, &address)?.is_eq() {
            continue;
        }
        match tracker.maybe_authorize_invocation(host, function, !has_active_tracker)
    {
            // If tracker doesn't have a matching invocation,
            // just skip it (there could still be another
            // tracker  that matches it).
            Ok(false) => continue,
            // Found a matching authorization.
            Ok(true) => return Ok(()),
            // Found a matching authorization, but another
            // requirement hasn't been fulfilled (for
            // example, incorrect authentication or nonce).
            Err(e) => return Err(e),
```

```
29          }
30      }
31  }
```

Informally, this loop iterates over each of the trackers (which are derived from the user provided invocation signature tree) attempting to match the current call to one of the trackers. If the authorization succeeds, the call is allowed as usual. If the call fails, we either continue or propagate an error if one has arisen.

If there is a call to `require_auth` within a `__check_call`, the call `tracker.maybe_authorize_invocation (host, function, !has_active_tracker)` could potentially call `require_auth_enforcing` again, which will again iterate through the trackers. Because each tracker is "borrowed" via that call to `tracker.try_borrow_mut()`, the same `tracker` cannot be used to match multiple calls to `require_auth` at once. However, specially crafted trackers and a recursive call to `require_auth` in `__check_auth` can still lead to a long (but finite) sequence of calls which can eat up significant resources of Stellar nodes with little cost to the user.

Impact An attacker could use this vulnerability to perform a denial of service attack.

Why Invalid It turns out that a failed call to `require_auth` in `__check_auth` will return an `Err`. As shown in the loop above, `Err`s are propagated up, meaning the number of iterations of this loop will be linear in the number of trackers. Because constructing a large number of trackers is costly to the attacker in other ways, this would be a costly attack to perform, and thus likely not worth it.

A.2.3 V-SOR-APP-VUL-004: Signature Replay Attack

Severity	High	**Commit**	16f4d7c
Type	Data Validation	**Status**	Investigated
File(s)	`rs-stellar-xdr/src/next/generated.rs`		
Location(s)	N/A		
Confirmed Fix At	N/A		

`SorobanAddressCredentials` contains a signature expiration ledger and a nonce for each signature. These credentials are only stored on the ledger as temporary values so they will be completely erased once they expire. As a result, if the signature itself isn't hashed with an expiration date or timestamp, there would be no way to know if an attacker reuses a previous signature that has expired off of the ledger to replay a previous transaction on behalf of a user who had executed that transaction in the past.

Impact If this signature can be reused for a replay attack, this could allow attackers to steal funds etc. If the signature cannot be reused, then there is no vulnerability.

Recommendation Hash the signature with some timestamp for the transaction, so that if the signature is reused later in time, it is possible to know that the signature has expired, even if the previous nonce/expiration timestamp have been erased from the ledger.

Why Invalid Developers have indicated that the signature is already hashed with a timestamp that would prevent such an attack.

A.2.4 V-SOR-APP-VUL-005: Wasm Interpreter Crash

Severity	High	**Commit**	2674d86
Type	Denial of Service	**Status**	Investigated
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

This issue concerns a crash found in `wasmi` fuzz test only in the Rust build `rustc 1.76.0-nightly (49b3924bd 2023-11-27)`, with the nightly commit linked here. In the fuzz test, we use `wasm_smith` module to generate random Wasm objects and call each function of the Wasm object with random inputs.

When the fuzz test is run, it crashes with the error below in `ty_to_arg()` which matches the function types (`wasmi::ValueType`) to random values (`wasmi::Value`).

Error message:

```
==51503== ERROR: libFuzzer: deadly signal
    #0 0x105166800 in __sanitizer_print_stack_trace+0x28 (librustc-nightly_rt.asan.
    dylib:arm64+0x5a800)
    #1 0x1035d4ee8 in fuzzer::PrintStackTrace()+0x30 (wasmi:arm64+0x101320ee8)
    #2 0x1035c7ae0 in fuzzer::Fuzzer::CrashCallback()+0x54 (wasmi:arm64+0x101313ae0)
    #3 0x1894b1a20 in _sigtramp+0x34 (libsystem_platform.dylib:arm64+0x3a20)
    #4 0x5e240001023b15f4  (<unknown module>)
    #5 0x1023c48d0 in wasmi::ty_to_arg::h9b94c1055ba765db wasmi.rs:81
    #6 0x1023ca5f0 in wasmi::_::__libfuzzer_sys_run::h026c7b6edd18f6e4 wasmi.rs:129
    #7 0x1023c8098 in rust_fuzzer_test_input lib.rs:297
    #8 0x1035c0790 in std::panicking::try::do_call::ha9a3096c9643ba55+0xac (wasmi:
    arm64+0x10130c790)
    #9 0x1035c6c24 in __rust_try+0x20 (wasmi:arm64+0x101312c24)
    #10 0x1035c5b9c in LLVMFuzzerTestOneInput+0x1d0 (wasmi:arm64+0x101311b9c)
    #11 0x1035c944c in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned
     long)+0x150 (wasmi:arm64+0x10131544c)
    #12 0x1035c8ac8 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long,
    bool, fuzzer::InputInfo*, bool, bool*)+0x48 (wasmi:arm64+0x101314ac8)
    #13 0x1035cb088 in fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::__1::vector<
    fuzzer::SizedFile, std::__1::allocator<fuzzer::SizedFile>>&)+0x7d4 (wasmi:arm64+0
    x101317088)
    #14 0x1035cb24c in fuzzer::Fuzzer::Loop(std::__1::vector<fuzzer::SizedFile, std::
    __1::allocator<fuzzer::SizedFile>>&)+0xd0 (wasmi:arm64+0x10131724c)
    #15 0x1035ee9e0 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char
    const*, unsigned long))+0x1e68 (wasmi:arm64+0x10133a9e0)
    #16 0x1035fc59c in main+0x24 (wasmi:arm64+0x10134859c)
    #17 0x189109054  (<unknown module>)
    #18 0x5c44fffffffffc  (<unknown module>)

NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash reports.
SUMMARY: libFuzzer: deadly signal
MS: 0 ; base unit: 0000000000000000000000000000000000000000
0x2d,0x2a,0xa,0xff,0x57,
-*\012\377W
```



```
artifact_prefix='/Users/kadron/Work/Veridise/Audits/Stellar/soroban-env-rs-veridise-fork/soroban-env-host/fuzz/artifacts/wasmi/'; Test unit written to /Users/kadron/Work/Veridise/Audits/Stellar/soroban-env-rs-veridise-fork/soroban-env-host/fuzz/artifacts/wasmi/crash-9fe80a366cd6dda4a7b1f6ba1246274f2bf47e1d
Base64: LSoK/1c=
```

`ty_to_arg()` function extended from original `wasmi` fuzz test:

```
fn ty_to_arg(ty: &ValueType) -> Value {
    let mut rng = rand::thread_rng();
    match ty {
        ValueType::I32 => Value::I32(rng.gen::<i32>()),
        ValueType::I64 => Value::I64(rng.gen::<i64>()),
        ValueType::F32 => Value::F32(rng.gen::<f32>().into()),
        ValueType::F64 => Value::F64(rng.gen::<f64>().into()),
        ValueType::FuncRef => Value::from(FuncRef::null()),
        ValueType::ExternRef => Value::from(ExternRef::null()),
        _ => panic!("Unknown type")
    }
}
```

Impact Being able to crash a node consistently could allow an attacker to perform a denial of service attack.

Why Invalid After further investigation, it appears the issue is caused by some issue in this particular nightly version of Rust in `rng.gen()` and is not caused by any vulnerability in the developer's code.

Glossary

cargo-fuzz A rust interface into libfuzzer, an LLVM fuzzing library (https://llvm.org/docs/LibFuzzer.html). See https://github.com/rust-fuzz/cargo-fuzz for more information. 5, 17