Zellic

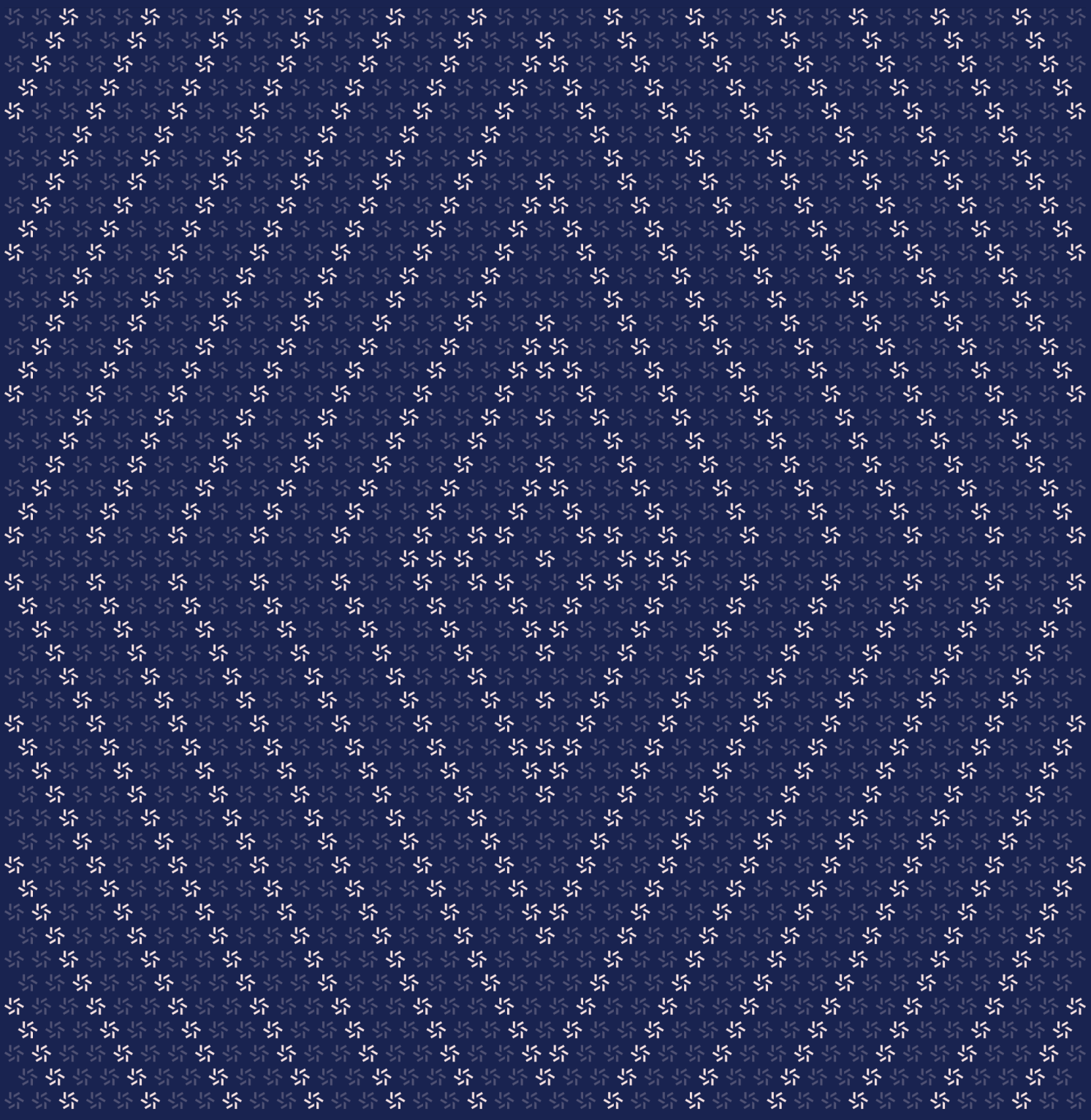
Prepared for
Rish Sharma
Smart Transaction Corp

Prepared by
Kuilin Li
Mohit Sharma
Zellic

August 15, 2024

Smart Transaction (STXN)

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Smart Transaction Corp from July 23rd 2024 to August 2nd 2024. During this engagement, Zellic reviewed Smart Transaction (STXN)'s code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can the frontrun and backrun protection guarantees be bypassed?
- Can the storage cleanup be bypassed to steal other users' tips?
- Can a malicious user obstruct functionality for others?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Smart Transaction (STXN) contracts, we discovered five findings. No critical issues were found. Two findings were of high impact, one was of medium impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Smart Transaction Corp's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	1
Low	0
Informational	2

2. Introduction

2.1. About Smart Transaction (STXN)

Smart Transaction Corp contributed the following description of Smart Transaction (STXN):

Smart Transaction (STXN) allows users to queue up calls in the present for execution by a solver in the future by pushing them to a Mempool (LaminatedProxy). Users can configure the calls in several ways, such as preventing frontruns and backruns or arranging them to be executable only after a delay.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.
>
> **Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.
>
> **Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.
>
> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Smart Transaction (STXN) Contracts

Type	Solidity
Platform	EVM-compatible
Target	stxn-contracts-core
Repository	https://github.com/smart-transaction/stxn-contracts-core ↗
Version	`a3eb0228810b313e3e75a77a7ca0d8dbe0f5df46`
Programs	`src/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.5 person-weeks. The assessment was conducted by two consultants over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Jr. Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Kuilin Li
Engineer
kuilin@zellic.io ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 23, 2024	Kick-off call
July 23, 2024	Start of primary review period
August 2, 2024	End of primary review period

3. Detailed Findings

3.1. EOA check in CallBreaker can be bypassed

Target	CallBreaker.sol		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `verify` method in CallBreaker.sol makes the following check to ensure the caller is an EOA.

```
if (msg.sender.code.length != 0) {
    revert MustBeEOA();
}
```

However, this check can be easily bypassed by deploying a contract that calls the CallBreaker contract from within its constructor. Since this call will be executed during the contract deployment transaction, the code at the contract's address will still be zero. This allows an attacker to bypass the EOA check.

Impact

A result of the verify function being called by a smart contract is that the caller contract may execute calls in the same transaction before and after the `pull` calls made to LaminatedProxy. These calls will be executed without mutating the value stored in `EXECUTING_CALL_INDEX_SLOT`. This can be used as a form of request smuggling to bypass the front-run and back-run blockers defined in SmarterContract.

```
function frontrunBlocker() public view {
    if (callbreaker.getCurrentlyExecuting() != 0) {
        revert IllegalFrontrun();
    }
}

function backrunBlocker() public view {
    uint256 currentlyExecuting = callbreaker.getCurrentlyExecuting();
    uint256 reversecurrentlyExecuting
    = callbreaker.getReverseIndex(currentlyExecuting);
    if (reversecurrentlyExecuting != 0) {
        revert IllegalBackrun();
    }
```

```
}
```

We would also like to note that even if the EOA check is fixed, the front-run and back-run protection features are labeled misleadingly since they do not actually prevent an attacker from front-running or back-running the transaction itself. Instead, they simply constrain the index of a call within the same transaction. True front-run or back-run protection (in terms of the colloquial meaning of the terms in the context of Ethereum) would require a proof of inclusion for the transaction in a particular block and require constraining the transaction to be the first or last transaction in said block.

Recommendations

We recommend modifying the EOA check to

```
if (msg.sender != tx.origin) {
    revert MustBeEOA();
}
```

Remediation

This issue has been acknowledged by Smart Transaction Corp, and a fix was implemented in commit ad206c54 ↗. Smart Transaction Corp indicated that they are considering removing this check altogether.

3.2. During `_cleanUpStorage`, `block.coinbase` may reenter

Target	CallBreaker		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

CallBreaker's `_cleanupStorage` method, which is called at the end of executing a job, transfers any remaining native token to the `blockBuilder`.

```
address payable blockBuilder = payable(block.coinbase);
blockBuilder.transfer(address(this).balance);
```

This allows `block.coinbase` to reenter at this point.

Impact

This becomes an issue because the portal is still marked open at this point. This allows the attacker to reenter and interact with contracts that use SmarterContract while the portal is still open.

Recommendations

A simple fix is to set the portal to closed before calling `cleanupStorage`. Note that this still allows the coinbase to reenter in the same transaction, which can potentially break the back-run protection guarantee (see Finding 3.1. ↗).

Remediation

This issue has been acknowledged by Smart Transaction Corp, and fixes were implemented in the following commits:

- `c4b7c48b` ↗
- `410d04e7` ↗

3.3. The `cancelAllPending` method does not cancel all pending calls

Target	LaminatedProxy		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `cancelAllPending` method in the LaminatedProxy contract works by iterating through all the queued jobs indexed between the `currentlyExecutingSequenceNumber` and `nextSequenceNumber` and marking them all as executed.

```
function cancelAllPending() external onlyOwner {
    uint256 _count = nextSequenceNumber();
    for (uint256 i = executingSequenceNumber(); i < _count; i++) {
        if (_deferredCalls[i].executed == false) {
            _deferredCalls[i].executed = true;
        }
    }
}
```

However, since the jobs can be pulled and executed out of order, there may be pending calls before the currently executing one, which will not be canceled by this function.

Impact

There may be pending calls before the currently executing one, which will not be canceled by this function. These calls can then be unexpectedly pulled when the owner expected them to be canceled.

Recommendations

We recommend adding a nonce to the call struct that is set to the current nonce when new calls are pushed as well as incrementing the nonce to cancel all pending unpulled calls.

This also fixes any potential out-of-gas DOS issues with this function if there are too many pending calls for `cancelAllPending` to succeed.

Remediation

This issue has been acknowledged by Smart Transaction Corp, and fixes were implemented in the following commits:

- b18f2b43
- 5dac6447

3.4. Executing call may delete itself by calling `cleanupLaminatorStorage`

Target	LaminatorProxy		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

A `callObject` inside a queued job may reenter and call `cleanupLaminatorStorage` with the current job's sequence number. Since the job is marked as executed before the calls are executed, `cleanupLaminatorStorage` will see it as having been executed and will delete it from the `_deferredCalls` map.

Impact

If a job is deleted from the `_deferredCalls` map and another call in the job tries to call `copyCurrentJob`, it will always revert.

Recommendations

We recommend adding the following check to `cleanupLaminatorStorage`.

```
function cleanupLaminatorStorage(uint256[] memory seqNumbers) public {
    for (uint256 i = 0; i < seqNumbers.length; i++) {
        if (!_deferredCalls[seqNumbers[i]].executed || (isCallExecuting() &&
    seqNumbers[i] == executingSequenceNumber)) {
            continue;
        }
        delete _deferredCalls[seqNumbers[i]];
    }
}
```

Remediation

This issue has been acknowledged by Smart Transaction Corp, and a fix was implemented in commit 0ec04922 ↗.

3.5. Nonexistent jobs can be cancelled via `cancelPending`

Target	LaminatorProxy		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

Since `cancelPending` does not perform any checks on `callSequenceNumber`, it can be called for a nonexistent job.

```
function cancelPending(uint256 callSequenceNumber) external onlyOwner {
    if (_deferredCalls[callSequenceNumber].executed == false) {
        _deferredCalls[callSequenceNumber].executed = true;
    }
}
```

Impact

There is no impact because if it is called for a nonexistent job, it will still be unset and overwritable by a later push. Note that calling `cancelPending` for a nonexistent job makes it able to be cleaned up, but that also has no impact.

Recommendations

A simple fix can be checking if the call being canceled is initialized, since `coh.initialized` will only be true for valid calls.

```
function cancelPending(uint256 callSequenceNumber) external onlyOwner {
    if (_deferredCalls[callSequenceNumber].executed == false &&
    _deferredCalls[callSequenceNumber].initialized == true) {
        _deferredCalls[callSequenceNumber].executed = true;
    }
}
```

Remediation

This issue has been acknowledged by Smart Transaction Corp, and a fix was implemented in commit 6499bb3c ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Gas wastage due to deploying laminator proxy multiple times

In the current design, the Laminator contract deploys a new copy of the LaminatorProxy contract for every new user. This incurs significant gas costs due to the size of the LaminatorProxy contract, since the entire contract code has to be deployed every time. The users can save a lot of gas if the protocol simply shifts to a beacon proxy pattern for the Laminator.

The shift would include the following changes.

1. A central implementation contract for LaminatorProxy is deployed, which has a constructor that disables its use.
2. Instead of a deploying a new contract for every new user, Laminator just deploys a new proxy that delegatecalls to the central implementation contract.

Since a proxy contract would be much smaller in size than the full implementation contract, it would help save a lot of gas in the protocol for onboarding users.

This issue has been acknowledged by Smart Transaction Corp. Smart Transaction Corp responded with the following:

> This will be a good improvement, but the impact of this change will reflect in all our core contracts, which is why we would like to skip it at least for this phase of our development.

4.2. The `execute` function defined in LaminatorProxy can never be called

The following function is defined in LaminatorProxy:

```
function execute(bytes calldata input)
    external onlyLaminator nonReentrant returns (bytes memory) {
    CallObject[] memory callsToMake = abi.decode(input, (CallObject[]));
    return _executeAll(callsToMake);
}
```

The function is marked `onlyLaminator` but is never called from the Laminator contract code, so it is not possible to call it. Smart Transaction Corp acknowledged that the function is redundant and can

be safely removed.

Note that if this function were changed to `onlyOwner` instead of being removed, a guard would be necessary to prevent it from being called while the portal is open, because `_executeAll` will set the executing sequence number and leave it at a value different from what the subsequent smarter contract logic expects it to be.

This issue has been acknowledged by Smart Transaction Corp, and a fix was implemented in commit `d2220c17` ↗.

4.3. Unnecessary complexity in some storage types

The CompactBytes, CompactDynArray, CallBreakerTypes, and TimeTypes files contain complex Yul assembly code to manually implement loading and storing various constant-length and variable-length types.

Even though we could not identify any specific issues with the implementation, we would like to caution against this additional complexity for gas savings. The economic value of gas savings generated by manually implementing these operations is small compared to the various pieces of off-chain overhead it induces, including the effort that casual users may spend reading the contract, the new-developer onboarding time required to understand the semantics and safety guarantees that this assembly implements, and more.

Additionally, the use of these custom storage types is not consistent. For instance, calldata and returndata are both the same type of type, a variable-length array of bytes. However, in storage, one of them is a custom storage type, and the other is an ordinary Solidity-managed variable-length array:

```
CallObjectStorage[] public callStore;
ReturnObject[] public returnStore;

// [... in _resetTraceStoresWith]
callStore.push().store(calls[i]);
returnStore.push(returnValues[i]);
```

It is unclear to someone reading the code why calldata needs to be custom but returndata can be a regular Solidity array.

This issue has been acknowledged by Smart Transaction Corp. Smart Transaction Corp responded clarifying that they will add documentation to clarify the complexity for these libraries.

4.4. Transient storage recommendation

This project uses storage slots in order to hold authoritative information about a call within a transaction so that a smarter contract (an external contract that uses the SmarterContract contract) can obtain information about the surrounding call context. These storage slots are cleared before the Ethereum transaction ends.

This is a perfect use case for the new EIP-1153 transient storage feature. We recommend using transient storage for information that needs to be in storage in between call contexts but not in between Ethereum transactions, such as the portal status, the calldatas and expected returndatas of the current pull, and so on.

Using transient storage would save a lot of gas on storage as well as simplify the mental model of the contract considerably because the expected storage that is always cleared at the start of execution will be guaranteed to be cleared.

This issue has been acknowledged by Smart Transaction Corp, and fixes were implemented in the following commits:

- `10e2e1c2` ↗
- `f56bd8c7` ↗

4.5. Miscellaneous gas optimizations

During the audit, we noted a few miscellaneous gas and logic optimizations:

- In the `pull` function in LaminatedProxy, `_setCurrentlyExecutingCallIndex(0)` is explicitly called. However, each iteration of the loop that goes through the calls does `_setCurrentlyExecutingCallIndex(i)`, which means that the currently executing call index will be set to zero twice, without any reentrancy in between the two sets.
- In the `copyCurrentJob` function in LaminatedProxy, the `onlyWhileExecuting` modifier is unnecessary because the sender is checked to be the proxy itself, and the proxy only calls itself while a call is being executed. Additionally, for code clarity, we recommend using the `onlyProxy` modifier instead of doing the check in the function body.
- In the `cleanupLaminatorStorage` function in LaminatedStorage, the `delete _deferredCalls[seqNumbers[i]]` operation does not delete all the storage data associated with the deferred call, because in TimeTypes, the implementation of `CallObjectHolderStorage` can store extra data at a custom storage location if the calldata length is long enough. We recommend implementing a wipe function in CallObjectLib in order to potentially obtain more gas refunds for callers of this function.
- In CallBreakerStorage, the `onlyPortalClosed` modifier calls `_setPortalOpen` and `_setPortalClosed`, but the `verify` function in CallBreaker, which is the only function that has this modifier, also calls `_setPortalOpen` and `_setPortalClosed`, resulting in the portal being opened and closed twice each call.

This issue has been acknowledged by Smart Transaction Corp, and fixes were implemented in the following commits:

- c060d0a5
- b8d71735
- c25b5852

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: CallBreaker.sol

Function: `verify(byte[] callsBytes, byte[] returnsBytes, byte[] associatedData, byte[] hintdices)`

This is the entry point for the solver to execute a solver-specified sequence of calls from the CallBreaker, likely including `pull` calls on individual LaminatedProxy contracts.

Inputs

- `callsBytes`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Calls must succeed.
- `returnsBytes`
  - **Control**: Arbitrary.
  - **Constraints**: Must match the actual return values of the calls.
  - **Impact**: Constraint.
- `associatedData`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Provides extra arbitrary key-value data available to contracts that are called.
- `hintdices`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Additional data that assists in proving relations on the call array in a gas-efficient way, so the solver can avoid a revert.

Branches and code coverage

Intended branches

- Calls succeed.
  - ☑ Test coverage

Negative behavior

- Reverts if reentrancy is attempted.
  - ☐ Negative test
- Reverts if any call reverts.
  - ☐ Negative test
- Reverts if any call's returndata does not match.
  - ☑ Negative test

5.2. Module: LaminatedProxy.sol

Function: `cancelAllPending()`

This cancels all pending calls.

Branches and code coverage

Intended branches

- Cancels all pending calls.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☐ Negative test

Function: `cancelPending(uint256 callSequenceNumber)`

This cancels one pending call, by the sequence number.

Inputs

- `callSequenceNumber`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Call to cancel.

Branches and code coverage

Intended branches

- Cancels the call.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☐ Negative test

Function: `copyCurrentJob(uint256 delay, byte[] shouldCopy)`

This copies the current job to be executed later.

Inputs

- `delay`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Blocks in the future to schedule the next call.
- `shouldCopy`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Arbitrary external call that returns whether the job should be copied.

Branches and code coverage

Intended branches

- Copies the current job if the `shouldCopy` check passes.
  - ☐ Test coverage
- Does not copy the current job if the `shouldCopy` check fails.
  - ☐ Test coverage

Negative behavior

- Reverts if not called by the proxy.
  - ☐ Negative test
- Reverts if the `shouldCopy` check reverts.
  - ☐ Negative test

Function: `pull(uint256 seqNumber)`

This executes a deferred call. This function can only be called by the CallBreaker, through a `verify` call from a solver.

Inputs

- `seqNumber`
  - **Control**: Arbitrarily specified by the solver.
  - **Constraints**: Must be a valid deferred call ID. Call must succeed.
  - **Impact**: Index of the call to execute.

Branches and code coverage

Intended branches

- Call succeeds.
  - ☑ Test coverage

Negative behavior

- Reverts if reentrancy is attempted.
  - ☐ Negative test
- Reverts if the caller is not the CallBreaker contract.
  - ☑ Negative test
- Reverts if the deferred call does not exist.
  - ☐ Negative test
- Reverts if the deferred call is not ready yet.
  - ☑ Negative test
- Reverts if the deferred call has already been executed.
  - ☑ Negative test
- Reverts if the deferred call reverts.
  - ☐ Negative test

Function: `push(byte[] input, uint256 delay)`

This adds a deferred call to be executed after the specified delay. This function can only be called by the Laminator contract that deployed this proxy (through `pushToProxy`) or by itself.

Inputs

- `input`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Call data of the pushed call.
- `delay`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Delay, in blocks, before the call is valid.

Branches and code coverage

Intended branches

- Pushes a call.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not authorized.
  - ☑ Negative test
- Reverts if the input is malformed.
  - ☐ Negative test

5.3. Module: LaminatedStorage.sol

Function: `cleanupLaminatorStorage(uint256[] seqNumbers)`

This permissionlessly clears storage slots for calls that have been executed, to claim a gas refund.

Inputs

- `seqNumbers`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Calls to attempt to clear.

Branches and code coverage

Intended branches

- Clears deferred calls that have been executed.
  - ☐ Test coverage
- Skips deferred calls that have not been executed.
  - ☐ Test coverage

Negative behavior

- None.

5.4. Module: Laminator.sol

Function: `pushToProxy(byte[] cData, uint32 delay)`

This deploys a user's LaminatedProxy, if it does not exist, and then pushes a new call to it.

Inputs

- `cData`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Call data of the pushed call.
- `delay`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Delay, in blocks, before the call is valid.

Branches and code coverage

Intended branches

- Pushes a call after deploying a proxy.
  - ☐ Test coverage
- Pushes a call to an already-deployed proxy.
  - ☐ Test coverage

Negative behavior

- None.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Smart Transaction (STXN) contracts, we discovered five findings. No critical issues were found. Two findings were of high impact, one was of medium impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.