SOLIDIFIED

Audit Report for Origin Protocol - December 17, 2020

Summary

Audit Report prepared by Solidified covering the Origin protocol staking and compensation claim contracts.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on December 16, 2020, and the results are presented here.

Audited Files

The following contracts were covered during the audit:

```
contracts
├──compensation
│    └── CompensationClaims.sol
└──staking
     └── SingleAssetStaking.sol
```

Supplied in the following source code repositories:

https://github.com/OriginProtocol/origin-dollar

commit number `0936691ee0d81f53be9f50a080a0a8f5ead2ed26`

Intended Behavior

The staking smart contract implements a staking functionality with the following properties

- Users can stake a single asset multiple times
- Each individual stake can be of a certain pre-defined duration
- Each duration has a specific earning rate.
- A privileged address can pre-sign stakes for a user (for example for airdrop vouchers)

The compensation contract implements functionality to compensate users affected by a previous security incident suffered by the project.

SOLIDIFIED

Audit Report for Origin Protocol - December 17, 2020

Executive Summary

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	High	-

SOLIDIFIED

Audit Report for Origin Protocol - December 17, 2020

Issues Found

Solidified found that the audited contracts contain no critical issue, 1 major issue, and 2 minor issues, in addition to 2 informational notes.

We recommend all issues are amended, while the notes are up to the team's discretion, as it refers to best practices.

Issue #	Description	Severity	Status
1	Pre-approved stake to 0x0 Address can allow an attacker to claim invalid stake	Major	Resolved
2	Inefficient loop logic might lead to block gas issues and/or excessive gas usage in some long-lived use cases	Minor	Resolved
3	Pre-approved stake process does not implement full signature replay protection	Minor	Resolved
4	Malleable signatures accepted	Note	Resolved
5	Staking contract may run out of funds	Note	-

Critical Issues

No critical issues have been found.

Major Issues

1. Pre-approved stake to 0x0 Address can allow an attacker to claim invalid stake

The `_setPreApprover()` function allows setting 0x0 address for preApprover under an assumption that 0x00 address disables the `preApprovedStake()` functionality.
However, this is not true - when preApprover is set to 0x0 address, an attacker could claim a stake without having a valid signature.
If one supplies an invalid value uint8 v parameter when calling the `preApprovedStake()` function, the `ecrecover(messageDigest, v, r, s)` function would evaluate to 0x0 address, thus making the check below pass:

```
require(
   preApprover == ecrecover(messageDigest, v, r, s),
     "Stake not approved"
);
```

See:
https://docs.kaleido.io/faqs/why-ecrecover-fails/

Recommendation
Consider checking for the 0x0 address.

Update
Fixed by replacing off-chain signature vouchers with a Merkle proof approach.

Minor Issues

2. Inefficient loop logic might lead to block gas issues and/or excessive gas usage in some long-lived use cases

User stakes are kept in a growing array per user. The `_stake()` function always increases the array size. Staking and exiting involve iterating over these arrays. This means that for long-lived use cases with many stakes per user, the operations will become expensive and may fail due to the block gas limit. In particular `preApprovedStake()` iterates over the entire array.
The severity of this issue might be higher in long-lived use cases with a large number of stakes per user.

Recommendation
There are several ways to mitigate this issue depending on the use case, including removing elements from the array on exit and/or implement a limit for the maximum number of stakes allowed per user at any given time. Keep in mind that if elements are removed, there'll be a need to add additional checks to ensure that a new stake is not the same type as a previously exited one.

Update
Fixed.

3. Pre-approved stake process does not implement full signature replay protection

Signatures provided for pre-approved stakes for a user can be used multiple times since there is nothing in the signed message that prevents a signature replay attack. Whilst a pre-approved stake cannot be submitted again because of the `stakeType` parameter being allowed only once, signature replays are theoretically possible between different deployments, for example, a testnet deployment.

Important Note: The severity of this issue could increase depending on the implemented solution for issue 5 (see above). In some cases, pre-approved stakes could be re-used indefinitely.

Recommendation
Consider using a nonce per user to make sure signatures cannot be replayed. It is also good practice to include the contract address and chain id on the signed message to avoid replaying between contract instances or from a testnet deployment.

Update
Fixed.

Notes

4. Malleable signatures accepted

The `preApprovedStake()` function uses the built-in `ecrecover()`. This function still allows malleable signatures for backward compatibility reasons. Signatures that have an `s` value larger than `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0` are usually rejected for Ethereum address post EIP-2.

Recommendation
Consider rejecting signatures with `s` values in the upper ranges, even though it may not be a security issue in this case.
For an example solution see https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/cryptography/ECDSA.sol

Update
Fixed by replacing signature procedure with Merkle proof.

5. Staking contract may run out of funds

There is no way to control the amount `USER_STAKE_TYPE` staked.
The existing `SingleAssetStaking.sol` contract's token balance can be "consumed" by rewards for `USER_STAKE_TYPE` stakes.
It could interfere with the intended `preApprovedStake()` functionality because both operations (the user stake and pre-approved stake) use the same token's pool (the balance of the `SingleAssetStaking.sol` contract).

Recommendation

Consider adding a governor-configurable maximum total of staked amount of `USER_STAKE_TYPE` type stakes.

Audit Report for Origin Protocol - December 17, 2020

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Origin Protocol or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Solidified Technologies Inc.