A CONSENSYS DILIGENCE AUDIT REPORT

Skale Token

Date	January 2020
Lead Auditor	Sergii Kravchenko
Co-auditors	Shayan Eskandari

1 Executive summary

In January 2020, SKALE engaged us to conduct a security assessment of the Skale Network Delegation contracts and ERC777 implementation.

> The SKALE Manager orchestrates and administers the entirety of the SKALE Network with respect to business, engineering, and security operations. The Manager is comprised as a set of Solidity contracts and is built to be deployed on the Ethereum mainnet. The Manager system is organized for upgradability by separating data and functional contract functionality.

The first review was conducted over two weeks, from January 27th to February 7th, by Sergii Kravchenko and Shayan Eskandari. A total of 15 person-days were spent during this period.

Because of the massive code changes following the first review, SKALE engaged us for a secondary review. This portion of the engagement consisted of 160 hours (20 person-days).

2 Scope

Our review focused on the token and delegation components of the skale-manager repository. We analyzed smart contract code at commit `50c8f4e037f6bf578d62bd752516b17237b55811`. We did not review any changes made after this commit during the initial review. A complete list of files in scope can be found in the Appendix.

SKALE provided the following documentation for use during our review:

- SKALE Manager Specification
- SKALE Network Whitepaper

2.1 Mitigation & Secondary Review

Following our initial review of the SKALE Manager codebase, the SKALE team implemented changes to address many of our findings and recommendations. Due to the massive scope of these changes, we elected to perform a second code review.

This review was performed on smart contract code at commit `7f9eefa99fe1eb10e5d600dec40cde396e35912c`. Changes made in the interim period were primarily merged in skalenetwork/skale-manager#92.

3 Key Observations/Recommendations

- The Skale network is a trusted system. Their goal is to move to a more trustless system over time.
- The codebase is very complex and has changed significantly over a short period. This new, less mature codebase is more likely to contain issues that have not been discovered by routine testing, development, and iteration.
- ~~There are many duplicate function names that are separated by only the contract they are in. (e.g., Most of the functions in DelegationService will call the same function name in another contract)~~
- The gas consumption of most function calls are not optimized and are high.
    - The cost of the initial deployment of the contracts is close to the block gas limit.

- Inline documentation is scarce, and most functions don't contain any comments. Inline documentation can help with future development and increases the readability of the code, especially for code reviews.
- ~~The naming of the functions should reflect their nature, such as functions starting with "get" should be only getters and do not change state. However, in SKALE, this pattern is not followed. For example, the~~ `getPurchasedAmount` ~~function calls the~~ `getState` ~~, which changes the state of the delegations. (For more examples see issue 5.12)~~
- The owner can change launch time at any point using the `setLaunchTimestamp` function.
- ~~If a delegation is not canceled by the end of its delegation period, it renews its period every 3 months. A multiplier remains the same.~~
- Delegation can be made for one of the following periods: 3 months, 6 months, 12 months. Depending on the period, each delegation has a multiplier to bounty sharing distribution: the longer you delegate, the more bounties you get. Periods and their multipliers can be changed anytime by the owner, which may result in different multipliers (even 0) for already existing delegation.
- Many functionalities of the code we reviewed are tightly related to out of scope code, which makes it harder to understand the intended behavior of the code in scope. It is possible issues exist that are relevant to how the out of scope code works.

- Optimization: ~~There are many iterations over delegations in different functions in the codebase, such as getting locked/delegated/slashed balances. Also, all transfers call functions that iterate through delegations, which uses more gas than normally expected. These can be optimized by removing unnecessary iterations in many external functions, which are expected to be cheap and straightforward.~~ **Update**: even though there are still some calculations happening during the token transfers, a lot of work has been done to make the system more optimized. Iterations over delegations switched to iterations over months and slashes in a more optimized and predictable way.

- ~~Not all mechanisms for proof of use is implemented correctly; the checks are not obvious and implicit where they should be more readable by the legal authorities. (See issue 5.9, issue 5.8, issue 5.7, issue 5.4 for more details)~~

Individual issues are listed below, but it is important to note that the contract system is quite large, with complex multi-function calls, upgradeability mechanisms, and exponential combinations of modules. There are potentially vulnerabilities in the system that our team did not find. The administrative control retained by SKALE (Owner, Token Manager, etc.) makes it possible to recover from some types of vulnerabilities, though this is not a catch-all.

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

4.1 Actors

The relevant actors are as follows:

- The **owner** of the system can upgrade all the contracts in the system. There is only one `owner` for all contracts, but each contract in the system can potentially have their own `owner` addresses. The most critical functionality is that the `ContractManager` owner can change the contract address of all modules in the system.
- **Validators** register their names, addresses (`msg.sender`), descriptions, fee rate, and minimum delegation amount.
- **Token holders** submit a delegation request through the `DelegationService` , specifying delegation value, delegation period, and validator ID.

Permissions/Roles

Roles in the SKALE network are defined by either actors or smart contracts:

- The **owner** (*Actor*)
    - Can update all contract addresses in the `ContractManager` .
    - Set and remove Delegation Period in the `DelegationPeriodManager` which also changes the delegation multiplier for bounties.
    - Forgive slashed amount for an account.

    - Set Minimum staking Requirement (Function details not Implemented).
    - Set the launch timestamp.
    - And in general, can call any public function that has the `allow` modifier.

- The **SkaleManager** (*Actor*)
    - Can mint tokens for the `SkaleToken`.
- **Validators** (*Actor*)
    - Accept Pending Delegation.
    - Link and unlink Nodes.
    - Can request and confirm the new validator's address (change their validator's address).
- **Token holders** (*Actor*)
    - Withdraw Bounty.
    - Request new delegation.
    - Cancel their delegation request.
    - Request Undelegation.
- **SkaleDKG** (*Contracts*) //Out of Scope
    - Slash.
- **Anyone**
    - Register as a Validator.

Note: Some permissions in the `DelegationService` and `ValidationService` are related to out of scope contracts and are not included in the above list of actors, such as node functionality. Most of the functionalities are called through the `DelegationService` contract.

4.2 Trust Model

In any smart contract system, it's important to identify what trust is expected/required between various actors. For this audit, we established the

following trust model:

- **SKALE** is run on the Ethereum Network and uses **ERC-777** token; the owner can call **all** of the permissioned functions (e.g., slash, lock, mint tokens).
- The **Owner** (upgrade key) of the `ContractManager` contract can upgrade each contract. The SKALE Network Upgrade key will soon transition to an on-chain voting mechanism, therefore, making the ownership a function of community governance. It will be centrally managed through a multi-sig process the initial 3 months to prioritize agility for resolving critical issues prior to becoming a community-owned on-chain function.
- The N.O.D.E. Anstalt will support the community in providing a temporary **whitelist of trusted validators** who will be the only ones able to run mainnet nodes for the launch and for a period of time post-launch. Subsequent mainnet launches will incorporate new validators who pass certification and/or participate in testnet activities.
- More details of the Trust Model can be found in the Actors and Permissions section.

Proof-of-use

SKALE uses Activate by Codefi (ConsenSys) for their token launch. Activate introduces some requirements to minimize passive speculation and some other legal benefits. The main aspect affecting the workflows in the SKALE smart contracts is Proof of Use. The requirements obligate all token holders that receive tokens from the initial launch to delegate at least 50% of their tokens for the first 3 months. To be precise, we checked the code according to the following formal rules:

- All the tokens that are bought from initial token launch are locked until at least 50% of them were delegated for at least 3 months.
- All tokens received not from the token launch (transferred from another participant) should not be locked.
- All bounties and fees are locked for the first 3 months after the token launch.
- Slashing is not active during the first 3 months.

Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 `uint` overflow may lead to stealing funds Critical

✓ Addressed

Resolution

safeMath was added in SKALE-215. At the time of the writing this comment, the review has not been comprehensive to all arithmetic calculations in the scope.

Note that in some cases usage of safeMath due to reverts can result in unexpected halting of the system, that too should be reviewed again.

Description

It's possible to create a delegation with a very huge amount which may result in a lot of critically bad malicious usages:

code/contracts/delegation/DelegationRequestManager.sol:L74-L76

```
uint holderBalance = SkaleToken(contractManager.getContract("SkaleToken")).b
uint lockedToDelegate = tokenState.getLockedCount(holder) - tokenState.getPu
require(holderBalance >= amount + lockedToDelegate, "Delegator hasn't enough
```

`amount` is passed by a user as a parameter, so if it's close to `uint` max value, `amount + lockedToDelegate` would overflow and this requirement would pass.

Having delegation with an almost infinite amount of tokens can lead to many various attacks on the system up to stealing funds and breaking everything.

Recommendation

Using `SafeMath` everywhere should prevent this and other similar issues. There should be more critical attacks caused by overflows/underflows, so `SafeMath` should be used everywhere in the codebase.

5.2 Holders can burn locked funds Major ✓ Addressed

Resolution
Fixed in SKALE-2144 by adding proper checks in `_burn()`.

Description

Skale token is a modified ERC-777 that allows locking some part of the balance. Locking is checked during every transfer:

code/contracts/ERC777/LockableERC777.sol:L433-L441

```
// Property of the company SKALE Labs inc.--------------------------------
        uint locked = _getLockedOf(from);
        if (locked > 0) {
            require(_balances[from] >= locked + amount, "Token should be unl
        }
//-------------------------------------------------------------------------
        _balances[from] = _balances[from].sub(amount);
        _balances[to] = _balances[to].add(amount);
```

But it's not checked during `burn` function and it's possible to "burn" locked tokens. Tokens will be burned, but `locked` amount will remain the same. That can result in having more `locked` tokens than the balance which may have very unpredictable behaviour.

Recommendation

Allow burning only unlocked tokens.

5.3 Node can unlink validator Major ✓ Addressed

Resolution

Fixed in SKALE-2145-unlink-node by adding a check in `unlinkNodeAddress()` that only `validatorAddress` has the permission to remove nodes from `validators[validatorId]` where `msg.sender == validators[validatorId].validatorAddress`

Description

Validators can link a node address to them by calling `linkNodeAddress` function:

code/contracts/delegation/ValidatorService.sol:L109-L119

```
function linkNodeAddress(address validatorAddress, address nodeAddress) exte
    uint validatorId = getValidatorId(validatorAddress);
    require(_validatorAddressToId[nodeAddress] == 0, "Validator cannot overr
    _validatorAddressToId[nodeAddress] = validatorId;
}

function unlinkNodeAddress(address validatorAddress, address nodeAddress) ex
    uint validatorId = getValidatorId(validatorAddress);
    require(_validatorAddressToId[nodeAddress] == validatorId, "Validator ha
    _validatorAddressToId[nodeAddress] = 0;
}
```

After that, the node has the same rights and is almost indistinguishable from the validator. So the node can even remove validator's address from `_validatorAddressToId` list and take over full control over validator. Additionally, the node can even remove itself by calling `unlinkNodeAddress`, leaving validator with no control at all forever.

Also, even without nodes, a validator can initially call `unlinkNodeAddress` to remove itself.

Recommendation

Linked nodes (and validator) should not be able to unlink validator's address from the `_validatorAddressToId` mapping.

5.4 Unlocking funds after slashing Major ✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

The initial funds can be unlocked if 51+% of them are delegated. However if any portion of the funds are slashed, the rest of the funds will not be unlocked at the end of the delegation period.

code/contracts/delegation/TokenState.sol:L258-L263

```
if (_isPurchased[delegationId]) {
    address holder = delegation.holder;
    _totalDelegated[holder] += delegation.amount;
    if (_totalDelegated[holder] >= _purchased[holder]) {
        purchasedToUnlocked(holder);
    }
```

Recommendation

Consider slashed tokens as delegated, or include them in the calculation for process to unlock in `endingDelegatedToUnlocked`

5.5 Bounties and fees should only be locked for the first 3 months Major ✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

Bounties are currently locked for the first 3 months after delegation:

code/contracts/delegation/DelegationService.sol:L315

```
skaleBalances.lockBounty(shares[i].holder, timeHelpers.addMonths(delegationS
```

Instead, they should be locked for the first 3 months after the token launch.

Recommendation

It's better just to forbid any withdrawals for the first 3 months, no need to track it separately for every delegation. This recommendation is mainly to simplify the process.

5.6 `getLockedCount` is iterating over all history of delegations Major ✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

`getLockedCount` is iterating over all delegations of a specific holder and may even change the state of these delegations by calling `getState`.

code/contracts/delegation/TokenState.sol:L60-L71

```
function getLockedCount(address holder) external returns (uint amount) {
    amount = 0;
    DelegationController delegationController = DelegationController(contrac
    uint[] memory delegationIds = delegationController.getDelegationsByHolde
    for (uint i = 0; i < delegationIds.length; ++i) {
        uint id = delegationIds[i];
        if (isLocked(getState(id))) {
            amount += delegationController.getDelegation(id).amount;
        }
    }
    return amount + getPurchasedAmount(holder) + this.getSlashedAmount(holde
}
```

This problem is major because delegations number is growing over time and may even potentially grow more than the gas limit and lock all tokens forever. `getLockedCount` is called during every transfer which makes any token transfer much more expensive than it should be.

Recommendation

Remove iterations over a potentially unlimited amount of tokens. All the necessary data can be precalculated before and `getLockedCount` function can have O(1) complexity.

5.7 Tokens are unlocked only when delegation ends Major

✓ Addressed

Resolution
Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

After the first 3 months since at least 50% of tokens are delegated, all tokens should be unlocked. In practice, they are only unlocked if at least 50% of tokens, that were bought on the initial launch, are undelegated.

code/contracts/delegation/TokenState.sol:L258-L264

```
if (_isPurchased[delegationId]) {
    address holder = delegation.holder;
    _totalDelegated[holder] += delegation.amount;
    if (_totalDelegated[holder] >= _purchased[holder]) {
        purchasedToUnlocked(holder);
    }
}
```

Recommendation

Implement lock mechanism according to the legal requirement.

5.8 Tokens after delegation should not be unlocked automatically Major ✓ Addressed

Resolution
Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

When some amount of tokens are delegated to a validator when the delegation period ends, these tokens are unlocked. However these tokens should be added to `_purchased` as they were in that state before their delegation.

code/contracts/delegation/TokenState.sol:L258-L264

```
if (_isPurchased[delegationId]) {
    address holder = delegation.holder;
    _totalDelegated[holder] += delegation.amount;
    if (_totalDelegated[holder] >= _purchased[holder]) {
        purchasedToUnlocked(holder);
    }
}
```

Recommendation

Tokens should only be unlocked if the main legal requirement `(_totalDelegated[holder] >= _purchased[holder])` is satisfied, which in the above case this has not happened.

5.9 Some `unlocked` tokens can become `locked` after delegation is rejected Major ✓ Addressed

Resolution
Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

When some amount of tokens are requested to be delegated to a validator, the validator can reject the request. The previous status of these tokens should be intact and not changed (locked or unlocked).

Here the initial status of tokens gets stored and it's either completely `locked` or `unlocked`:

code/contracts/delegation/TokenState.sol:L205-L214

```
if (_purchased[delegation.holder] > 0) {
    _isPurchased[delegationId] = true;
    if (_purchased[delegation.holder] > delegation.amount) {
        _purchased[delegation.holder] -= delegation.amount;
    } else {
        _purchased[delegation.holder] = 0;
    }
} else {
    _isPurchased[delegationId] = false;
}
```

The problem is that if some amount of these tokens are locked at the time of the request and the rest tokens are unlocked, they will all be considered as locked after the delegation was rejected.

code/contracts/delegation/TokenState.sol:L272-L278

```
function _cancel(uint delegationId, DelegationController.Delegation memory d
    if (_isPurchased[delegationId]) {
        state = purchasedProposedToPurchased(delegationId, delegation);
    } else {
        state = proposedToUnlocked(delegationId);
    }
}
```

Recommendation

Don't change the status of the rejected tokens.

5.10 Gas limit for bounty and slashing distribution Major

✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

After every bounty payment (should be once per month) to a validator, the bounty is distributed to all delegators. In order to do that, there is a `for` loop that iterates over all active delegators and sends their bounty to `SkaleBalances` contract:

code/contracts/delegation/DelegationService.sol:L310-L316

```
for (uint i = 0; i < shares.length; ++i) {
    skaleToken.send(address(skaleBalances), shares[i].amount, abi.encode(sha

    uint created = delegationController.getDelegation(shares[i].delegationId
    uint delegationStarted = timeHelpers.getNextMonthStartFromDate(created);
    skaleBalances.lockBounty(shares[i].holder, timeHelpers.addMonths(delegat
}
```

There are also few more loops over all the active delegators. This leads to a huge gas cost of distribution mechanism. A number of active delegators that can be processed before hitting the gas limit is limited and not big enough.

The same issue is with slashing:

code/contracts/delegation/DelegationService.sol:L95-L106

```
function slash(uint validatorId, uint amount) external allow("SkaleDKG") {
    ValidatorService validatorService = ValidatorService(contractManager.get
    require(validatorService.validatorExists(validatorId), "Validator does r

    Distributor distributor = Distributor(contractManager.getContract("Distr
    TokenState tokenState = TokenState(contractManager.getContract("TokenSta

    Distributor.Share[] memory shares = distributor.distributePenalties(vali
    for (uint i = 0; i < shares.length; ++i) {
        tokenState.slash(shares[i].delegationId, shares[i].amount);
    }
}
```

Recommendation

The best solution would require major changes to the codebase, but would eventually make it simpler and safer. Instead of distributing and centrally calculating bounty for each delegator during one call it's better to just store all the necessary values, so delegator would be able to calculate the bounty on withdrawal. Amongst the necessary values, there should be history of total delegated amounts per validator during each bounty payment and history of all delegations with durations of their active state.

5.11 ERC-777 callback issue Major ✓ Partially fixed

Resolution

reentrancyGaurd was added in SKALE-2153 to `transfer()` and `transferFrom()`. However other functions are still may contain reentrancy bug, such as `burn()`, `send`, etc. Even if all the functions in the token contract (even `view` functions like `balanceOf`) have re-entrancy

protection, some projects might be still potentially vulnerable to re-entrancy attacks that use callbacks of ERC-777.

UPDATE: in skalenetwork/skale-manager#128 `nonReentrant` modifier is now only added to callbacks: `_callTokensToSend` and `_callTokensReceived`. So far it's impossible to make balance changes inside of the callbacks because any new balance change also triggers a callback. Therefore, it addresses the issue of re-entrancy by a malicious outside party (non-SKALE). Note since SKALE network retains upgrade capacity of smart contracts. Therefore, it's potentially possible to do re-entrancy from the `_getAndUpdateLockedAmount` function call, if the corresponding contract is upgraded in a specific way.

This report raises this as an **unfixed minor issue**. This issue will be fixed if the upgrade capability for _getAndUpdateLockedAmount() is revoked by SKALE network governance in the future.

Description

ERC-777 token comes with callback functions to the receiver and the sender on every token transfer. This gives re-entrancy opportunities for everyone who's using this token. There is a chance that other systems might not handle ERC-777 correctly.

Examples

Uniswap reentrancy critical bug: https://medium.com/consensys-diligence/uniswap-audit-b90335ac007

Recommendation

Use ERC-20 standard or remove callback function calls.

Remove callback function usage from the system and replace them with a standard ERC-20 flow:

code/contracts/delegation/SkaleBalances.sol:L55-L68

```
function tokensReceived(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes calldata userData,
    bytes calldata operatorData
)
    external
    allow("SkaleToken")
{
    address recipient = abi.decode(userData, (address));
    stashBalance(recipient, amount);
}
```

code/contracts/delegation/DelegationService.sol:L275-L289

```
function tokensReceived(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes calldata userData,
    bytes calldata operatorData
)
    external
    allow("SkaleToken")
{
    require(userData.length == 32, "Data length is incorrect");
    uint validatorId = abi.decode(userData, (uint));
    distributeBounty(amount, validatorId);
}
```

5.12 Rename functions Medium ✓ Addressed

Resolution

Fixed in SKALE-2154-naming by renaming the functions. The functions that are not solely getters and update the state of the smart contract are renamed to have `getAndUpdate` in their names. At the time of the writing this comment, the review has not been comprehensive to all functions in the scope.

Description

The naming of the functions should reflect their nature, such as functions starting with “get” should be only getters and do not change state. This will result in confusion developments and the implicit state changes might not be noticed.

Other than getters, some other function or variable names are misleading.

Examples

The following functions are a few examples that are named as getters but they change the state.

- getState -> updateState
  - getDelegationsTotal
  - getDelegationsForValidator
  - getDelegationsByHolder

Some other naming that does not reflect the nature of the functionality:

- getPurchasedAmount -> getPurchasedUnlocked
- tokenState.Sold -> lock

Recommendation

For functions that get and update variables use `getAndUpdate` naming. Similarly use variable names that reflect the nature of the values they store.

5.13 Delegations might stuck in non-active validator Medium

Pending

Resolution

Skale team acknowledged this issue and will address this in future versions.

Description

If a validator does not get enough funds to run a node (`MSR - Minimum staking requirement`), all token holders that delegated tokens to the validator cannot switch to a different validator, and might result in funds getting stuck with the nonfunctioning validator for up to 12 months.

Example

code/contracts/delegation/ValidatorService.sol:L166

```
require((validatorNodes.length + 1) * msr <= delegationsTotal, "Validator ha
```

Recommendation

Allow token holders to withdraw delegation earlier if the validator didn't get enough funds for running nodes.

5.14 Disabled Validators still have delegated funds Medium

Pending

Resolution
Skale team acknowledged this issue and will address this in future versions.

Description

The owner of `ValidatorService` contract can enable and disable validators. The issue is that when a validator is disabled, it still has its delegations, and delegated funds will be locked until the end of their delegation period (up to 12 months).

code/contracts/delegation/ValidatorService.sol:L84-L90

```
function enableValidator(uint validatorId) external checkValidatorExists(val
    trustedValidators[validatorId] = true;
}

function disableValidator(uint validatorId) external checkValidatorExists(va
    trustedValidators[validatorId] = false;
}
```

Recommendation

It might make sense to release all delegations and stop validator's nodes if it's not trusted anymore. However, the rationale behind disabling the validators might be different that what we think, in any case there should be a way to handle this scenario, where the validator is disabled but there are funds delegated to it.

5.15 Fees can be > 100% Medium ✓ Addressed

Resolution

Added a check to prevent fee rates equal or higher than 100% in SKALE-2157-fee-check.

Description

A validator can be created with `feeRate > 1000` which would mean that the fee rate would be higher than 100%. Severity is not high because that validator will most likely be not whitelisted.

Also, 100%+ fees would still somehow work and not revert because of the absence of `SafeMath`.

Recommendation

Add sanity check for the input values in `registerValidator`, and do not allow adding a validator with a fee rate higher than 100%.

5.16 `getState` changes state implicitly Medium ✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

`getState` function is checking and changing the state of a delegation struct. This function is called in many places in the codebase. Every delegation has a lot of different possible states and all of them are changed implicitly during other transactions, which makes it hard to track the logic in the code and make future changes in the code close to impossible without breaking some functionalities.

Recommendation

The general suggestion would be to minimize the number of implicit storage changes. Many states can be either changed explicitly or be calculated without additional storage changes.

As an option, it's possible to get rid of state storage slot at all. `startDate` and `endDate` fields may set the current state:

- `initProposed` can be called during the creation of the proposal.
- no need to explicitly change states between `ACCEPTED` and `DELEGATED`, you can set the start date on acceptance and no further changes are required.
- no need to switch states between `DELEGATED` and `ENDING_DELEGATED`, when delegation is set to end, it's fine to just have `end_date` storage slot and make assign the date there when `undelegate` function is called.
- unlocking funds from delegation (or not accepted request) can be explicit.

Also see issue 5.19 for other suggestions regarding `getState` usage in the code

7.17 `_endingDelegations` list is redundant Medium

✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

`_endingDelegations` is a list of delegations that is created for optimisation purposes. But the only place it's used is in `getPurchasedAmount` function, so only a subset of all delegations is going to be updated.

code/contracts/delegation/TokenState.sol:L159-L164

```
function getPurchasedAmount(address holder) public returns (uint amount) {
    // check if any delegation was ended
    for (uint i = 0; i < _endingDelegations[holder].length; ++i) {
        getState(_endingDelegations[holder][i]);
    }
    return _purchased[holder];
```

But `getPurchasedAmount` function is mostly used after iterating over all delegations of the holder.

Recommendation

Remove `_endingDelegations` and switch to a mechanism that does not require looping through delegations list of potentially unlimited size.

5.18 Some functions are defined but not implemented

Medium ✓ Addressed

Resolution

Fixed by removing the empty functions and implementing some others in SKALE-2160. At the time of the writing this comment, the review has not been comprehensive to all functions in the scope.

Description

There are many functions that are defined but not implemented. They have a revert with a message as not implemented.

This results in complex code and reduces readability. Here is a some of these functions within the scope of this audit:

- DelegationService.setMinimumStakingRequirement()
- DelegationService.getAllDelegationRequests()
- DelegationService.getDelegationRequestsForValidator()
- DelegationService.listDelegationRequests()
- DelegationService.getDelegationRequestsForValidator() Many more functions in `DelegationService.sol`

Examples

code/contracts/delegation/DelegationService.sol:L152-L158

```
function getAllDelegationRequests() external returns(uint[] memory) {
    revert("Not implemented");
}

function getDelegationRequestsForValidator(uint validatorId) external returr
    revert("Not implemented");
}
```

Recommendation

If these functions are needed for this release, they must be implemented. If they are for future plan, it's better to remove the extra code in the smart contracts.

5.19 `tokenState.setState` redundant checks Medium

✓ Addressed

Resolution

Issue is fixed as a part of the major code changes in skalenetwork/skale-manager#92

Description

`tokenState.setState` is used to change the state of the token from: * PROPOSED to ACCEPTED (in `accept()`) * DELEGATED to ENDING_DELEGATED (in `requestUndelegation()`

The if/else statement in `setState` is too complicated and can be simplified, both to optimize gas usage and to increase readability.

Examples

code/contracts/delegation/TokenState.sol:L173-L197

```
function setState(uint delegationId, State newState) internal {
    TimeHelpers timeHelpers = TimeHelpers(contractManager.getContract("TimeH
    DelegationController delegationController = DelegationController(contrac

    require(newState != State.PROPOSED, "Can't set state to proposed");

    if (newState == State.ACCEPTED) {
        State currentState = getState(delegationId);
        require(currentState == State.PROPOSED, "Can't set state to accepted

        _state[delegationId] = State.ACCEPTED;
        _timelimit[delegationId] = timeHelpers.getNextMonthStart();
    } else if (newState == State.DELEGATED) {
        revert("Can't set state to delegated");
    } else if (newState == State.ENDING_DELEGATED) {
        require(getState(delegationId) == State.DELEGATED, "Can't set state
        DelegationController.Delegation memory delegation = delegationContro

        _state[delegationId] = State.ENDING_DELEGATED;
        _timelimit[delegationId] = timeHelpers.calculateDelegationEndTime(de
        _endingDelegations[delegation.holder].push(delegationId);
    } else {
        revert("Unknown state");
    }
}
```

Recommendation

Some of the changes that do not change the functionality of the `setState` function: * Remove `reverts()` and add the valid states to the `require()` at the beginning of the function * Remove multiple calls to `getState()` * Remove final else/revert as this is an internal function and States passed should be valid More optimization can be done which requires further understanding of the system and the state machine.

```
function setState(uint delegationId, State newState) internal {
        TimeHelpers timeHelpers = TimeHelpers(contractManager.getContract("T
        DelegationController delegationController = DelegationController(cor

        require(newState != State.PROPOSED || newState != State.DELEGATED, "
        State currentState = getState(delegationId);

        if (newState == State.ACCEPTED) {
            require(currentState == State.PROPOSED, "Can't set state to acce

            _state[delegationId] = State.ACCEPTED;
            _timelimit[delegationId] = timeHelpers.getNextMonthStart();
        } else if (newState == State.ENDING_DELEGATED) {
            require(currentState == State.DELEGATED, "Can't set state to enc
            DelegationController.Delegation memory delegation = delegationCo

            _state[delegationId] = State.ENDING_DELEGATED;
            _timelimit[delegationId] = timeHelpers.calculateDelegationEndTim
            _endingDelegations[delegation.holder].push(delegationId);
        }
    }
```

5.20 Validator should be able to remove delegator Medium

✓ Addressed

Resolution

Code added in SKALE-2162, If the delegation is not in `DELEGATED` state, both validator and the delegator can request `undelegation`.

Description

In order to delegate tokens to a validator, the validator should accept the delegation request, however it's not possible to remove the delegator for the next period.

Recommendation

For consistency, either allow a validator to `undelegate` delegators for the next period or remove acceptance mechanism if it's not needed.

5.21 Lack of logs and events on state changes Minor Pending

Resolution
Skale team acknowledged this issue and will address this in future versions, but given the minor level and need to begin remediation, it's left out of scope from the re-mediation tag.

Description

Events in Solidity are used to log major state changes in the system, as for tracebility and also trigger UI changes or user notifications. It is a good practice to use events for every value storage change to be able to trace back the system.

Recommendation

emit events whenever a state change happens. As an example slashing does not emit any events and cannot notify a user unless a service is polling the system state regularly.

5.22 `DelegationService` redundancy Minor ✓ Addressed

Resolution
`DelegationService` was removed in pull/114 and the functionality is distributed in `ValidatorService`, `DelegationController`.

Description

`DelegationService` acts as a gateway for every external call. The problem is that it adds extra complexity to the code, which makes it harder to read and add a new code. Also, it costs more gas because of extra calls between contracts.

Recommendation

The same functionality of DelegationService can be added through UI to allow direct calls to each contract. However, as the whole system is modular and upgradable, it is understandable why using one main contract as the point of interaction might make sense as well.

5.23 Add timelock for some `onlyOwner` functions Minor

Pending

Resolution

Skale team acknowledged this issue and gave us the following response:

> The SKALE Network Upgrade key will soon transition to an on-chain voting mechanism therefore making the ownership a function of community governance. It will be centrally managed through a multi-sig process for the initial 3 months to prioritize agility for resolving critical issues prior to becoming a community owned on-chain function. Successful Ethereum projects such as Maker have given clear data points on successful voting mechanism and community control which the SKALE Network will employ as soon as possible.

Description

The system is trusted in a way that there are some `owner`s have the power to do major changes in the system. The most powerful is `owner` of `tractManager` which can update any contract in any way. Even though the

system is trusted and this is intended behaviour, it's possible to mitigate this trust a bit.

Recommendation

Add timelock to major admin functions, so people would know about it beforehand (2 weeks before) and would be able to react somehow.

Severity is minor because if owners of SKALE would want to attack the system in that way, tokens would lose the value anyway, and security of SKALE chains would be unreliable. So it's unclear what can be done even having that knowledge beforehand.

6 Mitigation issues

This section lists the issues found in the mitigation phase. The audit team, reviewed the code fixes after the initial report was delivered,

6.1 Users can burn delegated tokens using re-entrancy attack Critical ✓ Addressed

Resolution
Mitigated in skalenetwork/skale-manager#128

Description

When a user burns tokens, the following code is called:

new_code/contracts/ERC777/LockableERC777.sol:L413-L426

```
        uint locked = _getAndUpdateLockedAmount(from);
        if (locked > 0) {
            require(_balances[from] >= locked.add(amount), "Token should be
        }
//-------------------------------------------------------------------------

        _callTokensToSend(
            operator, from, address(0), amount, data, operatorData
        );

        // Update state variables
        _totalSupply = _totalSupply.sub(amount);
        _balances[from] = _balances[from].sub(amount);
```

There is a callback function right after the check that there are enough unlocked tokens to burn. In this callback, the user can delegate all the tokens right before burning them without breaking the code flow.

Recommendation

`_callTokensToSend` should be called before checking for the unlocked amount of tokens, which is better defined as Checks-Effects-Interactions Pattern.

6.2 Rounding errors after slashing Major ✓ Addressed

Resolution

Mitigated in skalenetwork/skale-manager#130. `epsilon` of 10^6 is added. Most subtractions are not throwing errors anymore and just assign value to zero.

Description

When slashing happens `_delegatedToValidator` and `_effectiveDelegatedToValidator` values are reduced.

new_code/contracts/delegation/DelegationController.sol:L349-L355

```
function confiscate(uint validatorId, uint amount) external {
    uint currentMonth = getCurrentMonth();
    Fraction memory coefficient = reduce(_delegatedToValidator[validatorId],
    reduce(_effectiveDelegatedToValidator[validatorId], coefficient, current
    putToSlashingLog(_slashesOfValidator[validatorId], coefficient, currentM
    _slashes.push(SlashingEvent({reducingCoefficient: coefficient, validator
}
```

When holders process slashings, they reduce `_delegatedByHolderToValidator`, `_delegatedByHolder`, `_effectiveDelegatedByHolderToValidator` values.

new_code/contracts/delegation/DelegationController.sol:L892-L904

```
if (oldValue > 0) {
    reduce(
        _delegatedByHolderToValidator[holder][validatorId],
        _delegatedByHolder[holder],
        _slashes[index].reducingCoefficient,
        month);
    reduce(
        _effectiveDelegatedByHolderToValidator[holder][validatorId],
        _slashes[index].reducingCoefficient,
        month);
    slashingSignals[index.sub(begin)].holder = holder;
    slashingSignals[index.sub(begin)].penalty = oldValue.sub(getAndUpdateDel
}
```

Also when holders are undelegating, they are calculating how many tokens from `delegations[delegationId].amount` were slashed.

new_code/contracts/delegation/DelegationController.sol:L316

```
uint amountAfterSlashing = calculateDelegationAmountAfterSlashing(delegatior
```

All these values should be calculated one from another, but they all will have different rounding errors after slashing. For example, the assumptions that the total sum of all delegations from holder `X` to validator `Y` should still be equal to `_delegatedByHolderToValidator[X][Y]` is not true anymore. The problem is that these assumptions are still used. For example, when undelegating some

delegation with delegated amount equals `amount` (after slashing), the holder will reduce `_delegatedByHolderToValidator[X][Y]`, `_delegatedByHolder[X]` and `_delegatedToValidator[Y]` by `amount`. Since rounding errors of all these values are different that will lead to 2 possible scenarios:

1. If rounding error reduces `amount` not that much as other values, we can have `uint` underflow. This is especially dangerous because all calculations are delayed and we will know about underflow and `SafeMath` revert in the next month or later.
   Developers already made sure that rounding errors are aligned in a correct way, and that the reduced value should always be larger than the subtracted, so there should not be underflow. This solution is very unstable because it's hard to verify it and keep in mind even during a small code change.

2. If rounding errors make `amount` smaller then it should be, when other values should be zero (for example, when all the delegations are undelegated), these values will become some very small values. The problem here is that it would be impossible to compare values to zero.

Recommendation

1. Consider not calling `revert` on these subtractions and make result value be equals to zero if underflow happens.
2. Consider comparing to some small `epsilon` value instead of zero. Or similar to the previous point, on every subtraction check if the value is smaller then `epsilon`, and make it zero if it is.

6.3 Slashes do not affect bounty distribution Major ✓ Addressed

Resolution
Mitigated in skalenetwork/skale-manager#118

Description

When slashes are processed by a holder, only `_delegatedByHolderToValidator` and `_delegatedByHolder` values are reduced. But `_effectiveDelegatedByHolderToValidator`

value remains the same. This value is used to distribute bounties amongst delegators. So slashing will not affect that distribution.

contracts/delegation/DelegationController.sol:L863-L873

```
uint oldValue = getAndUpdateDelegatedByHolderToValidator(holder, validatorId
if (oldValue > 0) {
    uint month = _slashes[index].month;
    reduce(
        _delegatedByHolderToValidator[holder][validatorId],
        _delegatedByHolder[holder],
        _slashes[index].reducingCoefficient,
        month);
    slashingSignals[index.sub(begin)].holder = holder;
    slashingSignals[index.sub(begin)].penalty = oldValue.sub(getAndUpdateDel
}
```

Recommendation

Reduce `_effectiveDelegatedByHolderToValidator` and `_effectiveDelegatedToValidator` when slashes are processed.

6.4 Iterations over slashes Medium ✓ Addressed

Resolution

Partially mitigated in skalenetwork/skale-manager#163 . `sendSlashingSignals` function is now aggregating slashes per `holder` (if it's sorted by `holder`), which optimises gas cost.

Description

Every user should iterate over each slash (but only once) and process them in order to determine whether this slash impacted his delegations or not.

However, the check is done during almost every action that the user does because it updates the current state of the user's balance. The downside of method is that if there are a lot of slashes in the system, every user would

be forced to iterate over all of them even if the user is only trading tokens and only calls `transfer` function.

If the number of slashes is huge, checking them all in one function would impossible due to the block gas limit. It's possible to call the checking function separately and process slashes in batches. So this attack should not result in system halt and can be mitigated with manual intervention.

Also, there are two separate pipelines for iterating over slashes. One pipeline is for iterating over months to determine amount of slashed tokens in separate delegations. This one can potentially hit gas limit in many-many years. The other one is for modifying aggregated delegation values.

Recommendation

Try to avoid all the unnecessary iterations over a potentially unlimited number of items. Additionally, it's possible to optimize some calculations:

1. When slashing signals are processed, all of them always have the same `holder`. There's no reason for having an array of signals with the same holder (always with predefined length and values will most likely be zero). It seems possible to remove signals functionality and just aggregate the changes for the `Punisher`.
2. Try merge two pipelines into one.

6.5 Storage operations optimization Medium ✓ Addressed

Resolution
Mitigated in skalenetwork/skale-manager#179

Description

There are a lot of operations that write some value to the storage (uses `SSTORE` opcode) without actually changing it.

Examples

1. `getAndUpdateValue` function of `DelegationController` and `TokenLaunchLocker`:

new_code/contracts/delegation/DelegationController.sol:L711-L715

```
for (uint i = sequence.firstUnprocessedMonth; i <= month; ++i) {
    sequence.value = sequence.value.add(sequence.addDiff[i]).sub(sequence.su
    delete sequence.addDiff[i];
    delete sequence.subtractDiff[i];
}
```

In `handleSlash` function of `Punisher` contract `amount` will be zero in most cases:

new_code/contracts/delegation/Punisher.sol:L66-L68

```
function handleSlash(address holder, uint amount) external allow("Delegatior
    _locked[holder] = _locked[holder].add(amount);
}
```

Recommendation

Check if the value is the same and don't write it to the storage in that case.

6.6 Duplicate function implementation `addMonths()`

Medium ✓ Addressed

Resolution
Fixed in skalenetwork/skale-manager#127

Description

`TimeHelpers.addMonths()` implementation is redundant as it can directly use `BokkyPooBahsDateTimeLibrary.addMonths()` function.

Recommendation

Simply use return `BokkyPooBahsDateTimeLibrary.addMonths()` on the same function to prevent further code changes, it's still a good idea to call addMonth through TimeHelpers contract.

6.7 Function overloading Minor ✓ Addressed

Resolution
Fixed in skalenetwork/skale-manager#181

Description

Some functions in the codebase are overloaded. That makes code less readable and increases the probability of missing bugs.

For example, there are a lot of `reduce` function implementations in `DelegationController`:

new_code/contracts/delegation/DelegationController.sol:L722-L820

```
function reduce(PartialDifferencesValue storage sequence, uint amount, uint
    require(month.add(1) >= sequence.firstUnprocessedMonth, "Can't reduce va
    if (sequence.firstUnprocessedMonth == 0) {
        return createFraction(0);
    }
    uint value = getAndUpdateValue(sequence, month);
    if (value == 0) {
        return createFraction(0);
    }

    uint _amount = amount;
    if (value < amount) {
        _amount = value;
    }

    Fraction memory reducingCoefficient = createFraction(value.sub(_amount),
    reduce(sequence, reducingCoefficient, month);
    return reducingCoefficient;
}

function reduce(PartialDifferencesValue storage sequence, Fraction memory re
    reduce(
        sequence,
        sequence,
        reducingCoefficient,
        month,
        false);
```

```
}

function reduce(
    PartialDifferencesValue storage sequence,
    PartialDifferencesValue storage sumSequence,
    Fraction memory reducingCoefficient,
    uint month) internal
{
    reduce(
        sequence,
        sumSequence,
        reducingCoefficient,
        month,
        true);
}

function reduce(
    PartialDifferencesValue storage sequence,
    PartialDifferencesValue storage sumSequence,
    Fraction memory reducingCoefficient,
    uint month,
    bool hasSumSequence) internal
{
    require(month.add(1) >= sequence.firstUnprocessedMonth, "Can't reduce va
    if (hasSumSequence) {
        require(month.add(1) >= sumSequence.firstUnprocessedMonth, "Can't re
    }
    require(reducingCoefficient.numerator <= reducingCoefficient.denominator
    if (sequence.firstUnprocessedMonth == 0) {
        return;
    }
    uint value = getAndUpdateValue(sequence, month);
    if (value == 0) {
        return;
    }

    uint newValue = sequence.value.mul(reducingCoefficient.numerator).div(re
    if (hasSumSequence) {
        subtract(sumSequence, sequence.value.sub(newValue), month);
    }
    sequence.value = newValue;

    for (uint i = month.add(1); i <= sequence.lastChangedMonth; ++i) {
        uint newDiff = sequence.subtractDiff[i].mul(reducingCoefficient.nume
        if (hasSumSequence) {
            sumSequence.subtractDiff[i] = sumSequence.subtractDiff[i].sub(se
        }
        sequence.subtractDiff[i] = newDiff;
    }
}
```

```
function reduce(
    PartialDifferences storage sequence,
    Fraction memory reducingCoefficient,
    uint month) internal
{
    require(month.add(1) >= sequence.firstUnprocessedMonth, "Can't reduce va
    require(reducingCoefficient.numerator <= reducingCoefficient.denominator
    if (sequence.firstUnprocessedMonth == 0) {
        return;
    }
    uint value = getAndUpdateValue(sequence, month);
    if (value == 0) {
        return;
    }

    sequence.value[month] = sequence.value[month].mul(reducingCoefficient.nu

    for (uint i = month.add(1); i <= sequence.lastChangedMonth; ++i) {
        sequence.subtractDiff[i] = sequence.subtractDiff[i].mul(reducingCoef
    }
}
```

Recommendation

Avoid function overloading as a general guideline.

Appendix 1 - Files in Scope

This review covered the following files:

File	SHA-1 hash
contracts/ERC777/LockableERC777.sol	774ed92ab7a1b26387d94e2daff6105827a45cab
contracts/delegation/DelegationRequestManager.sol	e027a8f3804d595aba6dadee624481b7e96805c9
contracts/delegation/DelegationPeriodManager.sol	71ccf8845bc3e24defc49ff8dd58c988d7ba2e32
contracts/delegation/ValidatorServi.sol	63bcd203739df93075205afea616727c1a990b15

File	SHA-1 hash
contracts/delegation/TokenState.sol	05be2a6535baa8e45bcaa1ff73df22e37a146ca3
contracts/delegation/TimeHelpers.sol	f10dd716f86673f50099128f8fb43ef2fcc24bcf
contracts/delegation/Distributor.sol	cb1035588de4ce1f36c8dd5db731dc17730eec41
contracts/delegation/SkaleBalances.sol	13e18bd3c9d634ae9a7201bde4ff48be395b30d5
contracts/delegation/DelegationService.sol	c2cf301f4eddd85861b20732dddc7fa7a576c3fa
contracts/delegation/DelegationController.sol	e20ded8fc9d4aba0f607c78608beed6aaf41b7a1
contracts/delegation/TokenSaleManager.sol	8a9adaff1c7e77c474e23ffdfe394fde67a24cc6

The following contracts were inherited and used by contracts within the scope of our review. These contracts were reviewed at a high level based on their use in the above contracts:

File	SHA-1 hash
contracts/interfaces/delegation/IHolderDelegation.sol	d4e32bbfad4685a9ae426fa11732de46c9986707
contracts/interfaces/delegation/IValidatorDelegation.sol	3081480c1c53ad86343c9e2c53939962f1c71213
contracts/interfaces/delegation/IDelegatableToken.sol	99b1845a470bea5dff476c68de144f0171df28dd
contracts/interfaces/IManagerData.sol	e55b739243140b509855645b842f359e5c63c385
contracts/interfaces/tokenSale/ITokenSaleManager.sol	434d21f55b8454e2cdaef615fea93ce0bb30c3b5

File	SHA-1 hash
contracts/interfaces/IConstants.sol	bf912011c45499d6ab873ae55b0d53278d7f883b
contracts/Permissions.sol	b44f5e3b0ed755695b760a0a89ef603c7422cb10
contracts/ContractManager.sol	5c56d01c59fead17235531b0df1ad73b93062065
contracts/SkaleToken.sol	1120b9a046a2f96d5f5bada977b09164c7c4c513
contracts/thirdparty/BokkyPooBahsDateTimeLibrary.sol	b00a2888d7b718dd4ac6dc8a009ac0960d982464

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

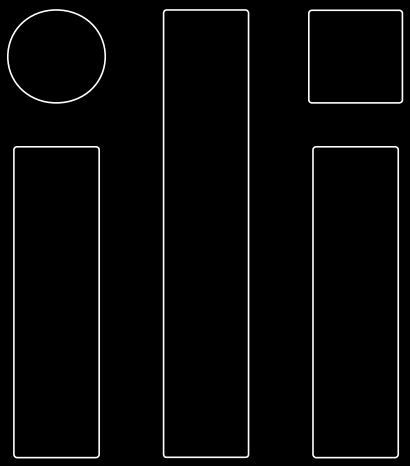
LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US



AUDITS

FUZZING

SCRIBBLE

BLOG

TOOLS

RESEARCH

ABOUT

CONTACT

CAREERS

PRIVACY POLICY

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

Email*

e-mail address

→