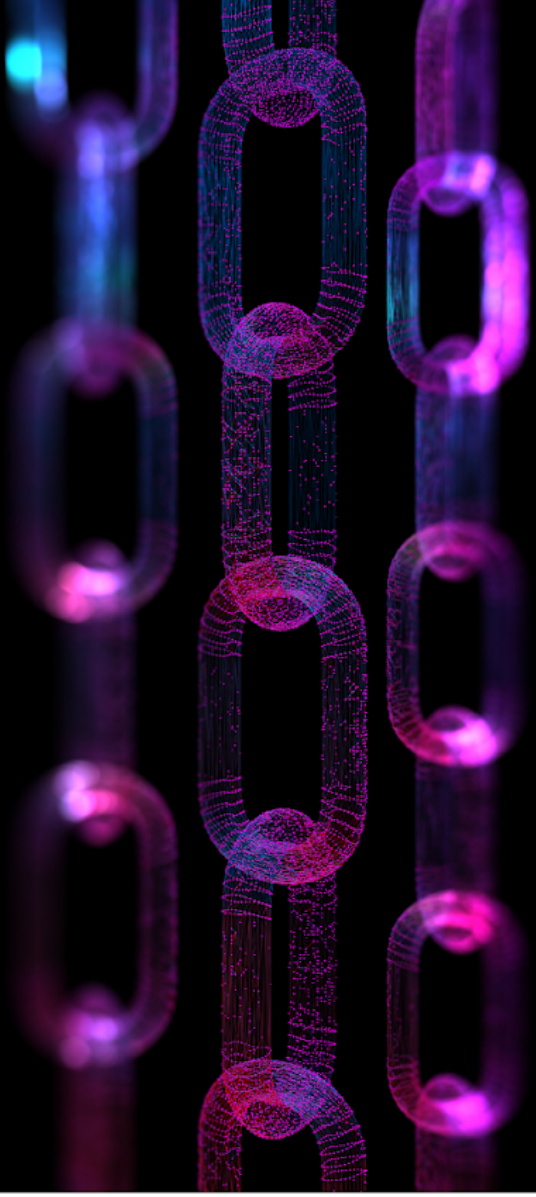


REPORT

Smart contract security review for Synco sp. z o.o.

Prepared by: Composable Security
Test time period: 2023-01-03 - 2023-01-06
Retest time period: 2023-02-09 - 2023-02-10
Report date: 2023-02-10
Visit: composable-security.com

1. Contents

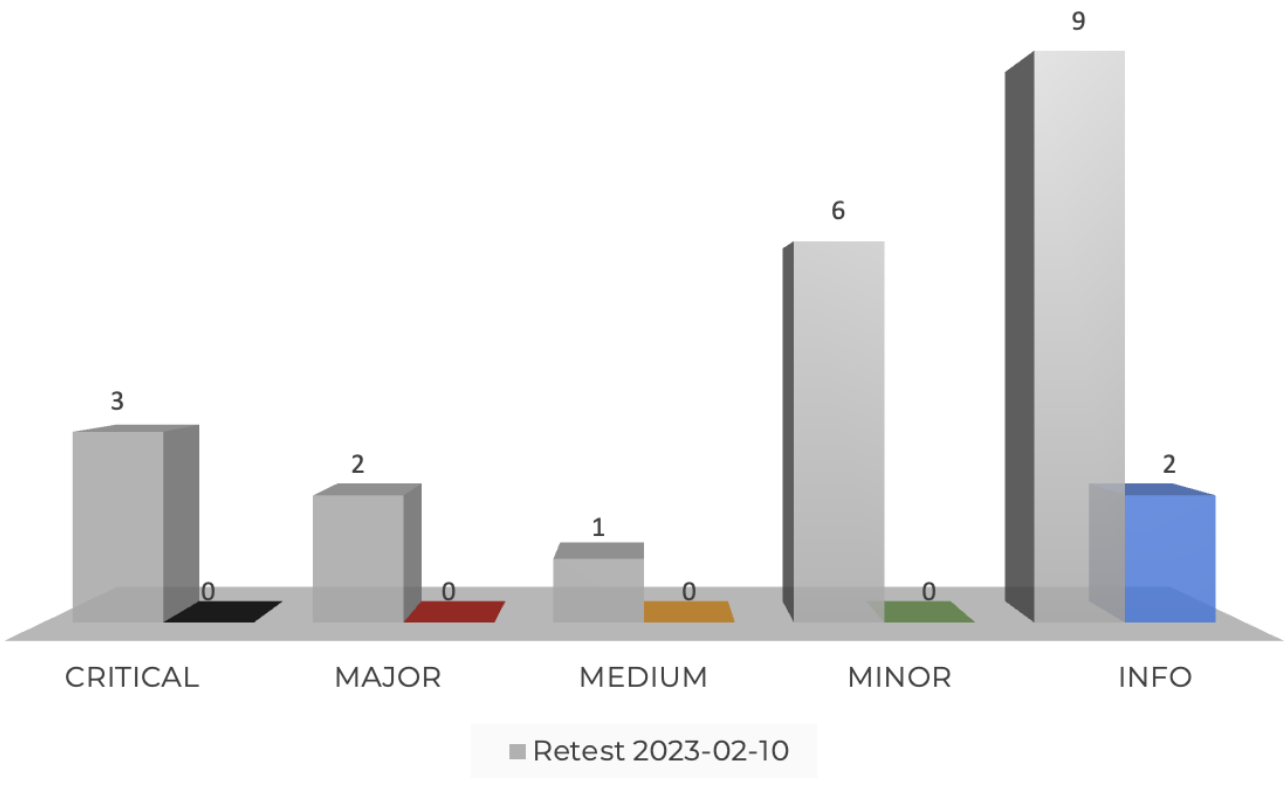
2. Retest (2023-02-10)

Scope

The retest scope included the same contracts, on a different commit in the same repository.

GitHub repository:
https://github.com/codefunded/smartcontracts/
CommitID:
864b85d57f7c1e3b245cf1773e0d1fc79edc45fc

Results diagram



Results

The Composable Security team was involved in a one-time iteration of verification whether the vulnerabilities detected during the tests were removed correctly and no longer appear in the code.

Previous security review was carried out 2023-01-06. Verified fixes have been made in the following repository:

GitHub repository:
https://github.com/codefunded/smartcontracts/

CommitID:
864b85d57f7c1e3b245cf1773e0d1fc79edc45fc

- **All 3 critical** vulnerabilities have been **fully removed** from the project.
- **All 2 major** vulnerabilities have been **fully removed** from the project.
- **One medium** vulnerability has been **fully removed**. The transfer of ownership is included in the deployment script.
- **All 6 vulnerabilities** with a **minor** impact on risk have been fixed.
- **Nine security recommendations** were handled as follows:
  - 7 have been implemented,
  - 1 has been partially implemented,
  - 1 has not been implemented.

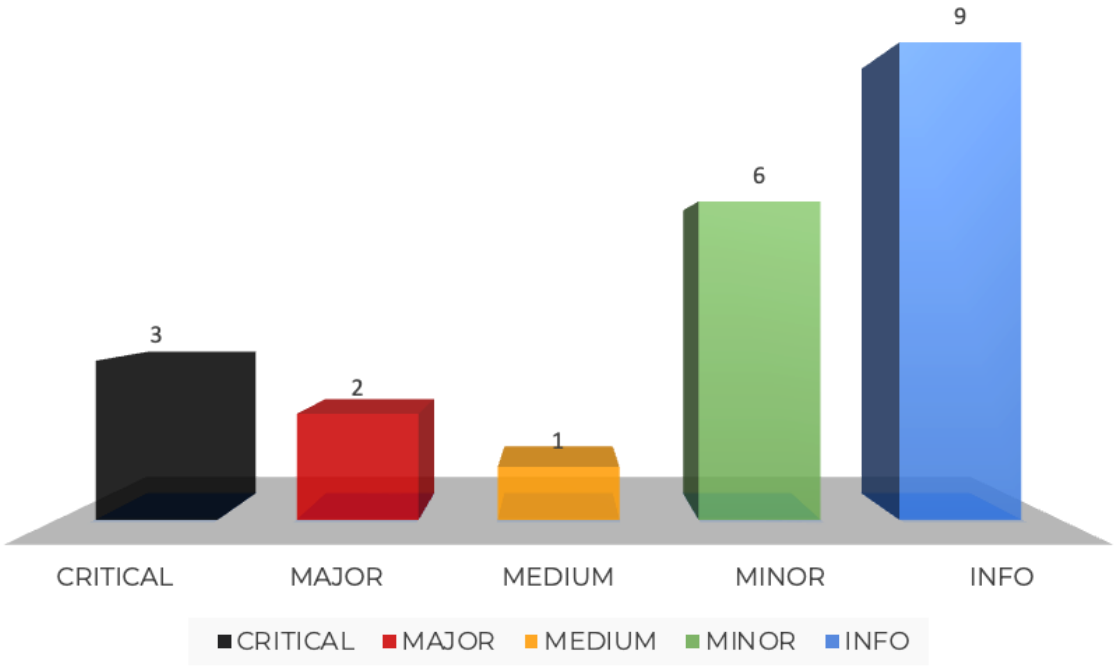
Findings overview

ID	Severity	Vulnerability	Retest 2023-02-10
MIC_91e451_5.1	CRITICAL	Use of spot reserves in DEX pool	FIXED
MIC_91e451_5.2	CRITICAL	No access control in *withdrawFor* function	FIXED
MIC_91e451_5.3	CRITICAL	Unauthorized mint of staking contract tokens	FIXED
MIC_91e451_5.4	MAJOR	Invalid amount of burnt tokens in staking contract	FIXED
MIC_91e451_5.5	MAJOR	Theft of rewards and denial of service via unauthorized schedule of staking period	FIXED
MIC_91e451_5.6	MEDIUM	Instant change of sensitive protocol parameters	FIXED
MIC_91e451_5.7	MINOR	Inability to handle all ERC20 tokens	FIXED
MIC_91e451_5.8	MINOR	Inconsistent deposit variables values	FIXED
MIC_91e451_5.9	MINOR	Lack of parameters validation	FIXED
MIC_91e451_5.10	MINOR	Invalid value of locked amounts variable	FIXED
MIC_91e451_5.11	MINOR	Invalid update of current period reward	FIXED
MIC_91e451_	MINOR	Invalid value of collected rewards variable	FIXED

5.12			
ID	**Severity**	**Vulnerability**	**Retest 2023-02-10**
MIC_91e451_6.1	INFO	Do not import whole contract for simple calculation	IMPLEMENTED
MIC_91e451_6.2	INFO	Remove *nonReentrant* modifier for the functions without external calls	IMPLEMENTED
MIC_91e451_6.3	INFO	Remove unused inheritance	IMPLEMENTED
MIC_91e451_6.4	INFO	Consider using specific solidity version	PARTIALLY IMPLEMENTED
MIC_91e451_6.5	INFO	Monitor and update draft version contract	NOT IMPLEMENTED
MIC_91e451_6.6	INFO	Use consistent variable naming	IMPLEMENTED
MIC_91e451_6.7	INFO	Make variables' names self-explanatory	IMPLEMENTED
MIC_91e451_6.8	INFO	Favor pull over push	IMPLEMENTED
MIC_91e451_6.9	INFO	Get the block.timestamp directly instead of using the view function	IMPLEMENTED

3. Executive summary

3.1. Audit results diagram (2022-01-06)



3.2. Audit results

The *Synco sp. z o.o.* engaged Composable Security to review security of *Milky Ice* protocol. Composable Security conducted this assessment over half person-week with 2 engineers.

The scope of the tests included selected contracts from the following repository.

GitHub repository: https://github.com/codefunded/smartcontracts/
CommitID: *91e45182755567df3a048115f3c202e33864a3d8*

Audit findings:

- **3** vulnerabilities with a critical impact on risk were identified. Their potential consequences are:
  - Minting a huge amount of sMIC tokens and potentially gMIC tokens (if the LP token is entitled to vote).
  - Users cannot withdraw their staked assets.
  - Minting arbitrary amounts of tokens in staking contracts and stealing rewards tokens.
- **2** vulnerabilities with a major impact on risk were identified. One of them was found in a contract that was not in the scope of testing. Their potential consequences are:

    - Still possessing tokens in the staking contract after withdrawing the locked assets (non-collateralized staking contract tokens).
    - No possibility to set new rewards period (long duration) or stealing rewards by setting huge rewardRate passing a huge amount of reward.
- **1** vulnerability with a medium impact on risk was identified. Its potential consequence is:
    - Ability to generate a huge amount of *sMIC* and *gMIC* tokens.
- 6 vulnerabilities with a minor impact on risk were identified.
- 9 recommendations have been proposed that can improve overall security and help implement best practice.
- The multiple important issues detected concern access control which needs to be improved.

Composable Security recommends that *Synco sp. z o.o.* complete the following:

- Address all reported issues.
- Take care of access control in the project by creating a permission matrix. Each role should be clearly defined by its access to features. Access control should be verified in a set of unit tests written at the beginning, which will help avoid such problems in the future.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible.
- Consider whether the detected vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.

4. Project details

4.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for *Synco sp. z o.o.* and their users.

The secondary goal is to improve code clarity and optimize code where possible.

4.2. Agreed scope of tests

The subjects of the test were selected contracts from the *CodeFunded* repository.
GitHub repository: https://github.com/codefunded/smartcontracts/
CommitID: *91e45182755567df3a048115f3c202e33864a3d8*

Files in scope:

```
.
├── staking
│   ├── MintStaking.sol
│   └── Staking.sol
└── tokens
    ├── DividendToken.sol
    └── MultiERC20WeightedLocker.sol
```

Documentation: The architecture overview was briefly described in the GitHub repository.

4.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security issues that could be exploited to achieve these threats.

Potential attackers goals:

- Theft of user's funds.
- Lock users' funds in the contract.
- Block the contract, so that others cannot use it.

- Minting unlimited amounts of tokens.

Potential scenarios to achieve the indicated attacker's goals:

- Influence or bypass the business logic of the system.
- Take advantage of arithmetic errors.
- Privilege escalation through incorrect access control to functions or badly written modifiers.
- Existence of known vulnerabilities (e.g., front-running, re-entrancy).
- Design issues.
- Excessive power, too much in relation to the declared one.
- Unintentional loss of the ability to govern the system.
- Poor security against taking over the managing account.
- Private key compromise, rug-pull.
- Withdrawal of more funds than expected.
- Oracle price manipulation.
- Impersonating other users.

4.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the *Synco sp. z o.o.* and *CodeFunded* development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using *slither*.
- Custom scripts (e.g. unit tests) to verify scenarios from initial threat modeling.
- **Manual review of the code.**

4.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable Security team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of development and maintenance.

5. Findings overview

ID	Severity	Vulnerability
MIC_91e451_5.1	CRITICAL	Use of spot reserves in DEX pool
MIC_91e451_5.2	CRITICAL	No access control in *withdrawFor* function
MIC_91e451_5.3	CRITICAL	Unauthorized mint of staking contract tokens
MIC_91e451_5.4	MAJOR	Invalid amount of burnt tokens in staking contract
MIC_91e451_5.5	MAJOR	Theft of rewards and denial of service via unauthorized schedule of staking period
MIC_91e451_5.6	MEDIUM	Instant change of sensitive protocol parameters
MIC_91e451_5.7	MINOR	Inability to handle all ERC20 tokens
MIC_91e451_5.8	MINOR	Inconsistent deposit variables values
MIC_91e451_5.9	MINOR	Lack of parameters validation
MIC_91e451_5.10	MINOR	Invalid value of locked amounts variable
MIC_91e451_5.11	MINOR	Invalid update of current period reward
MIC_91e451_5.12	MINOR	Invalid value of collected rewards variable
ID	**Severity**	**Recommendation**
MIC_91e451_6.1	INFO	Do not import whole contract for simple calculation
MIC_91e451_6.2	INFO	Remove *nonReentrant* modifier for the functions without external calls
MIC_91e451_6.3	INFO	Remove unused inheritance

MIC_91e451_6.4	INFO	Consider using specific solidity version
MIC_91e451_6.5	INFO	Monitor and update draft version contract
MIC_91e451_6.6	INFO	Use consistent variable naming
MIC_91e451_6.7	INFO	Make variables' names self-explanatory
MIC_91e451_6.8	INFO	Favor pull over push
MIC_91e451_6.9	INFO	Get the block.timestamp directly instead of using the view function

6. Vulnerabilities

6.1. Use of spot reserves in DEX pool

Status 2023-02-10	FIXED
The UniswapV2TwapOracle contract (extended Uniswap's example) has been added to the protocol to track the token price in a TWAP manner. The prices are semi-automatically updated using the *update* that can be called manually or using Gelato's task. The above-mentioned contract is used by LiquidityValueCalculator contract that gets the price from TWAP oracle and calculates the current price of an LP token share. Such an approach mitigates the risk of using spot prices and protects from instant price manipulation within one transaction. It is important to remember however, that pools with low liquidity can be manipulated for many blocks. It is important to monitor the pool and detect abnormal prices.	

Severity

CRITICAL

Affected smart contracts

MultiERC20WeightedLocker

Description

The *stake* function allows to stake LP tokens for pools that contain the *MIC* token. In order to calculate the amount of *sMIC* tokens to be minted, the contract takes the balance of *MIC* tokens in the DEX pool (MultiERC20WeightedLocker#L225, LiquidityValueCalculator.sol#L27).

However, the reserves can be easily imbalanced under certain conditions.

Additionally, there is a typo in the *computeLiquidityShareValue* function call, because the LP token is passed as the argument instead of *MIC* token (MultiERC20WeightedLocker#L228).

Attack scenario

The attacker would have to possess a big amount of *MIC* tokens or there should be another DEX pool with *MIC* token.

The attackers might take the following steps in turn:

- If there is another DEX pool with *MIC* token, take a flash loan and swap the coin for a big amount of *MIC*.
- Sell all *MIC* tokens in the lockable LP token pool to increase the balance of *MIC* tokens in the pool.
- Stake LP tokens in the contract.
- The contract gets the inflated balance of *MIC* tokens in the pool and mints twice as much *sMIC* tokens.
- Buy back the *MIC* tokens sold in the second step.
- If the flash loan was taken, pay it back.

Result of the attack: Minting a huge amount of *sMIC* tokens and potentially *gMIC* tokens (if the LP token is entitled to vote).

Recommendation
• As it is hard to base the business logic on the spot parameters we would recommend storing historical values of price (*see references*) and detect a situation when the DEX pool is imbalanced, e.g. by comparing the value of both tokens in the pool. • Alternatively, the protocol could use the Uniswap V3 pool that contains TWAP oracle by default and calculate the value of staked UniswapV3 position token in *MIC* token. • In the end, it is important to make sure that the cost of imbalancing the pool in the long term (slippage) is greater than income (e.g., profits from stake tokens and governance tokens).

References

SCSVS V8: Access Control
https://composablesecurity.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html
UniswapV2 Price Oracle
https://docs.uniswap.org/contracts/v2/concepts/core-concepts/oracles

6.2. No access control in *withdrawFor* function

Status 2023-02-10	FIXED
Functions in the *Staking* contract are protected with a modifier that allows them to be called only by addresses with assigned *LOCKER_ROLE*. Functions in the *MintStaking* contract are protected with a modifier that allows them to be called only by the owner.	

Severity

CRITICAL

Affected smart contracts

Staking, MintStaking

Description

The *withdrawFor* function in Staking.sol#L153 and MintStaking.sol#L103 contracts is an external function that allows users to decrease the staking balance of indicated users. There is no access control, thus the function can be called by any address.

Attack scenario

The vulnerable scenario might include the following steps in turn:

- The victim stakes asset using *stake* function (MultiERC20WeightedLocker.sol#L200). This function call *stakeFor* function (Staking.sol#L117, MintStaking.sol#L89) providing the user's address and the staked amount.
- Malicious user calls the *withdrawFor* function (Staking.sol#L153, MintStaking.sol#L103) giving the victim's address as the argument.
- If the victim calls *liquidateStaleDeposit* function (MultiERC20WeightedLocker.sol#L379) or withdraw funcion (MultiERC20WeightedLocker.sol#L308) they will revert, making it impossible to withdraw assets..

Result of the attack: Users cannot withdraw their staked assets.

Recommendation
• Limit the access to *withdrawFor* function only for *LOCKER_ROLE* by adding an Access Control modifier.

References

SCSVS V2: Access Control
https://composablesecurity.github.io/SCSVS/1.2/0x11-V2-Access-Control.html

6.3. Unauthorized mint of staking contract tokens

Status 2023-02-10	FIXED
Functions in the *MintStaking* contract are protected with a modifier that allows them to be called only by the owner.	

Severity

CRITICAL

Affected smart contracts

MintStaking, MintableToken

Description

The *MintStaking* contract mints a fixed amount of reward in *ERC20* tokens owned by the project creator. The reward is paid in *MintableToken* which can only be minted by the staking contract.

Functions *stakeFor* (MintStaking.sol#L89) and *collectRewardsFor* (MintStaking.sol#L117) are not protected and anyone can call them.

Attack scenario

The attacker might take the following steps in turn:

- Call the *stakeFor* function providing an arbitrary amount as an argument (e.g. *type(uint256).max*) and their address as receiver.
- After some time, call the *collectRewardFor* function which performs an external *rewardsToken* mint of the accrued reward.

Result of the attack: Minting arbitrary amounts of tokens in staking contracts and stealing rewards tokens.

Recommendation
Limit the access to *MintStaking*'s user facing functions (listed below) by adding an Access Control modifier - *onlyRole(LOCKER_ROLE)*. • *stakeFor* (MintStaking.sol#L89) • *withdrawFor* (MintStaking.sol#L103)

- *collectRewardsFor* (MintStaking.sol#L117)

References

SCSVS V2: Access control
https://composablesecurity.github.io/SCSVS/1.2/0x11-V2-Access-Control.html

6.4. Invalid amount of burnt tokens in staking contract

Status 2023-02-10	FIXED
The amount has been fixed.	

Severity

MAJOR

Affected smart contracts

MultiERC20WeightedLocker

Description

The *_removeDeposit* function (MultiERC20WeightedLocker.sol#L416) updates the contract's state on each deposit, including withdrawing the stake from the staking contract.

However, the amount of withdrawn tokens from staking contract is not the same as was minted (when the user was staking the assets - MultiERC20WeightedLocker.sol#L258). Instead, the amount of locked assets is withdrawn.

Attack scenario

The attackers might take the following steps in turn:

- Stake LP token and get double the amount of staked tokens (MultiERC20WeightedLocker.sol#L235).
- Wait until the lock period is finished.
- Withdraw the stake.
- The staking contract burns the locked amount of lockable asset instead of the minted amount and leaves the user with tokens in the staking contract.

Result of the attack: The attacker after withdrawing the locked assets (non-collateralized staking contract tokens) is left with tokens in staking contract.

The amount is equal to the number of minted tokens subtracted by the number of locked tokens. This leftover can be used multiple times to calculate rewards.

There is also a theoretical possibility that the number of minted tokens is lower than locked tokens and that would cause Denial of Service and not allow users to withdraw all deposits.

Recommendation
Burn the *mintedAmount* tokens instead of *lockedAmount* tokens.

References

SCSVS V5: Arithmetic
https://composablesecurity.github.io/SCSVS/1.2/0x14-V5-Arithmetic.html
SCSVS V8: Business Logic
https://composablesecurity.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html

6.5. Theft of rewards and denial of service via unauthorized schedule of staking period

Status 2023-02-10	FIXED
The process to start a new staking period has been divided in two phases. The first phase is an authenticated call (only addresses with assigned *SCHEDULER_ROLE* are allowed) that sets the values for the next period (finish date and rewards amount). The second phase is a call to *startNewRewardsPeriod* function by anyone which verifies that the next period can be started and takes all parameter values from the first phase.	

Severity

MAJOR

Affected smart contracts

PeriodStarter

Description

***Note**: This contract was not in the testing scope, however it has a direct impact on the tested contracts. Therefore, during the analysis of its operation, a vulnerability was detected in it.*

Although vulnerability was found in it, this contract cannot be considered fully tested and is recommended to be included in the scope of future testing.

The Project uses *Gelato* as a scheduler to call the new staking periods. The external *startNewRewardsPeriod (PeriodStarter.sol#L98)* function can be called by anyone and creates a new staking period if the previous one is already ended.

Attack scenario

The attackers might take the following steps in turn:

- Wait for the current scheduled task to be finished or front-run the transaction that sets a new rewards period.
- Call the *startNewRewardsPeriod* function providing an arbitrary number (*e.g. type(uint256).max*) as the duration time or the reward amount.

Result of the attack: No possibility to set new rewards period (long duration) or stealing rewards by setting huge *rewardRate* passing a huge amount of reward.

Recommendation
Limit the access to *PeriodStarter* contracts' *startNewRewardsPeriod* function. Make it callable only by the trusted Gelato operators.

References

SCSVS V2: Access control
https://composablesecurity.github.io/SCSVS/1.2/0x11-V2-Access-Control.html

6.6. Instant change of sensitive protocol parameters

Status 2023-02-10	FIXED
The team added a *Timelock* contract and plans to transfer ownership of the *MultiERC20WeightedLocker* and *Airdrop* contracts to it. The deployment script includes this ownership transfer.	

Severity

MEDIUM

Affected smart contracts

MultiERC20WeightedLocker

Description

The *MultiERC20WeightedLocker* contract allows locking of multiple assets and stake in multiple staking contracts. Those assets and contracts can be added using *addLockableAsset* (MultiERC20WeightedLocker.sol#L157) and *addStakingContract (*MultiERC20WeightedLocker.sol#L145*)*.

A malicious asset could be added instantly and the attacker could easily mint new *sMIC* and *gMIC* tokens without any limits if the owner's private key was leaked.

Additionally, it is a good practice to increase protocol's truthfulness to protect from centralization and make the protocol not rug-pullable.

Attack scenario

The attackers might take the following steps in turn:

- Call *addStakingContract* function that adds a new lockable token which is a fake token controlled by the attacker and is entitled to vote.
- Stake a huge number of fake tokens to get a huge number of *sMIC* and *gMIC* tokens.

Result of the attack: Ability to generate a huge amount of *sMIC* and *gMIC* tokens.

Recommendation

- Add timelocks to functions that update sensitive protocol parameters (e.g., add new lockable assets, add new staking contracts).
- However, the staking periods may take longer than the timelock period so it is reasonable to allow withdrawal (as emergency) with proportional interests or without interests.
- In the long-term, use the DAO governance contract to update sensitive protocol parameters.

References

SCSVS V2: Access Control
https://composablesecurity.github.io/SCSVS/1.2/0x11-V2-Access-Control.html

6.7. Inability to handle all ERC20 tokens

Status 2023-02-10	FIXED
The team has used *SafeERC20* library.	

Severity

MINOR

Affected smart contracts

MultiERC20WeightedLocker, Staking

Description

The functions:

- *stake* (MultiERC20WeightedLocker.sol#L200),
- *withdraw* (MultiERC20WeightedLocker.sol#L308),
- *collectRewardsFor* (Staking.sol#L144),

check the result of the *transferFrom* or *transfer* functions calls and revert if the *false* value is returned.

There are ERC20 tokens that do not return any value on transfers (simply reverts on failures) and in their case, all before-mentioned functions would revert and would not allow handling such lockable assets.

The impact on risk has been decreased because the team wants to handle only *MIC* and Uniswap V2 LP tokens.

Vulnerable scenario

The vulnerable scenario includes the following steps in turn:

- The governance adds a new lockable asset token that does not return boolean on transfers (e.g. USDT).
- User tries to stake USDT.
- The call is reverted and the user loses gas.

Result of the attack: Denial of service of ERC20 assets that do not return *true* on transfer (e.g. USDT).

Recommendation
Use SafeERC20 library to make sure that the return value is *true* if and only if any value is returned.

References

SCSVS V14: Communications
https://composablesecurity.github.io/SCSVS/1.2/0x13-V4-Communications.html
SafeERC20
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

6.8. Inconsistent deposit variables values

Status 2023-02-10	FIXED
The team has used the *EnumerableSet* library.	

Severity

MINOR

Affected smart contracts

MultiERC20WeightedLocker

Description

The *_addDeposit* function updates the contract's state on each deposit. Two of the state variables are the list of *depositors* (MultiERC20WeightedLocker.sol#L466) and *depositorsAmount* (MultiERC20WeightedLocker.sol#L467).

When the same depositor adds two deposits, they are reflected in the list and in the number of depositors twice.

Vulnerable scenario

The vulnerable scenario includes the following steps in turn:

- A user deposits a stake.
- The same user deposits another stake.
- Protocol sets the inconsistent values for *depositors* and *depositorsAmount* variables.

Result of the attack: Inconsistent values of state parameters, i.e. multiple repetitions of the same depositor in the list and inflated number of depositors.

Recommendation
• Consider adding new depositors to the set only if they do not exist in it (use *EnumerableSet* library). • Update the number of depositors analogously.

References

SCSVS V4: Arithmetic
https://composablesecurity.github.io/SCSVS/1.2/0x14-V5-Arithmetic.html
EnumerableSet
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableSet.sol

6.9. Lack of parameters validation

Status 2023-02-10	FIXED
The validation has been added.	

Severity

MINOR

Affected smart contracts

MultiERC20WeightedLocker

Description

The *addLockableAsset* function (MultiERC20WeightedLocker.sol#L157) does not validate the parameters of added assets.
It is reasonable to check whether the reward modifier is greater than 100% to make sure that it is profitable. The same validation should be applied to lock periods.

Vulnerable scenario

The vulnerable scenario includes the following steps in turn:

- The governance adds a new lockable asset with unprofitable rewards, by mistake.
- User stake lockable assets.
- When withdrawing, the user loses rewards and some locked assets.

Result of the attack: Users could use unprofitable staking.

Recommendation
Add validation to make sure that rewards are profitable (reward modifiers greater than 10000).

References

SCSVS V7: Business Logic:
https://composablesecurity.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html

6.10. Invalid value of locked amounts variable

Status 2023-02-10	FIXED
The variable has been changed to *userLockedAssetAmount* to track locked amount per asset and per user and the statement has been added in *_removeDeposit* function.	

Severity

MINOR

Affected smart contracts

MultiERC20WeightedLocker

Description

The *liquidateStaleDeposit* (MultiERC20WeightedLocker.sol#L379) function updates the contract's state on each liquidation (e.g. removes deposit), but it forgets to decrease the *lockedAssetAmount* variable.

Result of the attack: Invalid value of *lockedAssetAmount* variable which is not decreased after stale deposit is liquidated.

Recommendation
Add statement that decreases the value: `lockedAssetAmount[deposit.lockableAssetIndex] -= deposit.amountLocked;`

References

SCSVS V4: Arithmetic
https://composablesecurity.github.io/SCSVS/1.2/0x14-V5-Arithmetic.html

6.11. Invalid update of current period reward

Status 2023-02-10	FIXED
The new period cannot be started before the previous one is finished. Even though the code that incorrectly updated the *currentPeriodRewardsAmount* variable still exists, it is not reachable.	

Severity

MINOR

Affected smart contracts

Staking

Description

The *_notifyRewardAmount* function (Staking.sol#L198) sets *currentPeriodRewardsAmount* variable to *_amount* while there is a case when rewards from the previous period remain and are taken into account when calculating the reward rate.

The *currentPeriodRewardsAmount* variable does not include the remaining rewards.

Result of the attack: Invalid value (too small) of the *currentPeriodRewardsAmount* variable.

Recommendation
Set the *currentPeriodRewardsAmount* variable to correct value (including remaining rewards if necessary) in both mentioned cases.

References

SCSVS V4: Arithmetic
https://composablesecurity.github.io/SCSVS/1.2/0x14-V5-Arithmetic.html
SCSVS V8: Business Logic
https://composablesecurity.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html

6.12. Invalid value of collected rewards variable

Status 2023-02-10	FIXED
The *collectedRewardsInCurrentPeriod* variable is increased in the *collectRewardsFor* function.	

Severity

MINOR

Affected smart contracts

Staking.sol

Description

The *_notifyRewardAmount* function (Staking.sol#L198) sets *collectedRewardsInCurrentPeriod* variable to *0*, and this variable is never increased on the rewards withdrawal.

Result of the attack: Invalid value returned by the *collectedRewardsInCurrentPeriod* variable.

Recommendation
Increase the *collectedRewardsInCurrentPeriod* variable with the value of collected rewards by the user in the current period in the *collectRewardsFor* function.

References

SCSVS V8: Business Logic
https://composablesecurity.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html

7. Recommendations

7.1. Do not import whole contracts for simple calculations

Status 2023-02-10	IMPLEMENTED
Implemented according to the recommendation.	

Severity

INFO

Description

The *lastTimeRewardApplicable* (Staking.sol#L90) function uses OZ's *Math* contract to indicate whether the *block.timestamp* or *finishAt* variable is smaller.

Recommendation

Instead of importing the whole contract to calculate the minimum value, use ternary operator. It saves 220 gas units during deployment and 61 gas per execution.

```
90  function lastTimeRewardApplicable() public view returns (uint256) {
91    return finishAt < block.timestamp ? finishAt : block.timestamp
92  }
```

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

7.2. Remove *nonReentrant* modifier for the functions without external calls

Status 2023-02-10	IMPLEMENTED
Implemented according to the recommendation.	

Severity

INFO

Description

The *Staking* contract uses ReentranctGuard's *nonReentrant* modifier in order to protect against reentrancy. However, without using external calls inside the function, the reentrancy attack is not possible.

Recommendation
• Remove the *nonReentrant* modifier for *stakeFor* and *withdrawFor* functions. **Note:** If the protocols plans to chose only USDC (and other well-known ERC20 tokens) for *rewardsToken*, it is also recommended to remove *nonReentrant* modifier from *collectRewardFor* (Staking.sol#L137) functions and then remove the *ReentrancyGuard* import (Staking.sol#L92).

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

7.3. Remove unused inheritance

Status 2023-02-10	IMPLEMENTED
The inheritance for the *DividentToken* contract has been removed while the *onlyOwner* modifier has been used in the *MintStaking* contract.	

Severity

INFO

Description

The *MintStaking* (MintStaking#L20) and *DividendToken* (DividentToken#L15) contracts inherit from *Ownable* contract. However, the *onlyOwner* modifier is not used in the contracts' code.

Recommendation
Remove unused inheritance.

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

7.4. Consider using the specific solidity version

Status 2023-02-10	PARTIALLY IMPLEMENTED
The specific version (0.8.17) is used for contracts fully implemented by the team, but the pragma is still floating for libraries that were copy-pasted and modified (e.g. *UniswapV2Library*).	

Severity

INFO

Description

Audited code use the following pragma: `pragma solidity ^0.8.17;`

It allows the team to compile contracts with various versions of the compiler and introduces the risk of using a different version when deploying that during testing.

Recommendation
Use a specific version of Solidity compiler (latest stable): `pragma solidity 0.8.17;`

References

SCSVS V1: Architecture, design and threat modeling
https://github.com/securing/SCSVS/blob/master/1.2/0x10-V1-Architecture-Design-Threat-modelling.md
Floating pragma SWC-103
https://swcregistry.io/docs/SWC-103

7.5. Monitor and update draft version contracts

Status 2023-02-10	NOT IMPLEMENTED
The *ERC20Permit* contract has not been updated.	

Severity

INFO

Description

The *DividendToken* inherits from draft version of *ERC20Permit* contract. Contract drafts may not be exhaustively tested, updated or changed.

Recommendation
Use the stable version of *ERC20Permit* contract if possible. If only the draft version of the contract is currently available, monitor it and the changes that take place in it to stay up to date

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md
EIP-2612

https://eips.ethereum.org/EIPS/eip-2612

7.6. Use consistent variable naming

Status 2023-02-10	IMPLEMENTED
Implemented according to the recommendation.	

Severity

INFO

Description

All but one constructor parameters in the DividentToken#L18 contract are prefixed with the underscore symbol. It is important to be consistent when naming variables to keep the code clear.

Recommendation
Use the same naming convention, e.g. add underscore to *name* function parameter.

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

7.7. Make variables' names self-explanatory

Status 2023-02-10	IMPLEMENTED
Implemented according to the recommendation.	

Severity

INFO

Description

The *userDepositsAmount* variable in *_addDeposit* function (MultiERC20WeightedLocker.sol#L465) represents the number of deposits while its name suggests the value of deposits.

Recommendation
• Change *userDepositsAmount* variable name to *userDepositsCount*. • Change *depositorsAmount* variable name to *depositorsCount*.

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

7.8. Favor pull over push

Status 2023-02-10	IMPLEMENTED
Implemented according to the recommendation.	

Severity

INFO

Description

The *withdraw* function calls *collectRewards* function (MultiERC20WeightedLocker.sol#L324). It is recommended to favor pulling tokens over pushing tokens, which means that if it is possible to delegate transfer to another transaction, it should be implemented this way. This pattern protects users from blocking the withdrawal of locked assets in a situation when collecting rewards (e.g. reward token transfer) reverts.

Recommendation
• Remove automatic collection of rewards from the *withdraw* function. • If you want to allow users to withdraw stake and collect rewards in one transaction, create a new function *withdrawStakeAndReward* that will call *withdraw* and *collectRewards* functions.

References

SCSVS V4: Communications
https://composablesecurity.github.io/SCSVS/1.2/0x13-V4-Communications.html

7.9. Get the *block.timestamp* directly instead of using the *view* function

Status 2023-02-10	IMPLEMENTED
Implemented according to the recommendation.	

Severity

INFO

Description

The *lastTimeRewardApplicable* function returns the current *block.timestamp*. This value can be obtained directly, depending on the need.

Recommendation
Remove the *lastTimeRewardApplicable* function.

References

SCSVS V4: Communications
https://composablesecurity.github.io/SCSVS/1.2/0x13-V4-Communications.html

8. Impact on risk classification

Risk classification is based on the one developed by OWASP, however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

		Overall risk severity		
Impact on risk	HIGH	CRITICAL	MAJOR	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	MINOR
	LOW	MINOR	MINOR	INFO
		LOW	MEDIUM	HIGH
		Exploitation conditions		

OWASP Risk Rating methodology:
https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

9. Long-term best practices

9.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

9.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

9.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

9.4. Discuss audit reports and learn from them

The best guarantee of security is the constant development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

9.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

10. Contact

Damian Rusinek
Smart Contract Security Auditor
@drdr_zz
damian.rusinek@composable-security.com

Paweł Kuryłowicz
Smart Contract Security Auditor
@wh01s7
pawel.kurylowicz@composable-security.com