Crystalize - Contracts

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 25th, 2023 - September 8th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	09/04/2023
0.2	Document Update	09/08/2023
0.3	Draft Version	09/08/2023
0.4	Draft Review	09/08/2023
1.0	Remediation Plan	09/20/2023
1.1	Remediation Plan Review	09/21/2023
1.2	Remediation Plan Review	09/22/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Crystalize is an omni-chain staking protocol with the ability to purchase the token to stake on the fly.

Crystalize engaged Halborn to conduct a security assessment on their smart contracts beginning on August 25th, 2023 and ending on September 8th, 2023. The security assessment was scoped to the smart contracts provided in the crystalizefi/crystalize-contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided 2 weeks for the engagement and assigned a team of one full-time security engineer to review the security of the smart contracts in scope. The security team consists of a blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks, which were mostly addressed by Crystalize. The main one was the following:

- Move the ReentrancyGuard from the AsyncSwapper contract to the Zap contract.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs (MythX).
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry, Brownie).

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Project Name

- Repository: crystalizefi/crystalize-contracts
- Commit ID: f70c712ca981263efe104f954cb82a2861115240
- Smart contracts in scope:
    1. pool/Registry.sol
    2. pool/PoolFactory.sol
    3. pool/Pool.sol
    4. librairies/ERC20Utils.sol
    5. librairies/Error.sol
    6. lens/Lens.sol
    7. swapper/AsyncSwapper.sol
    8. zap/TokenKeeper.sol
    9. zap/Zap.sol
    10. stargate/StargateReceiver.sol
- Fix commit ID: 3169deba476a7fd7d609dc3fa5d27b9d9fcc5f57

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	1	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) SWAPS OVERWRITE THE ZAP CONTRACT OWNER	Medium (5.0)	SOLVED - 09/11/2023
(HAL-02) INCOMPATIBILITY WITH TRANSFER-ON-FEE OR DEFLATIONARY TOKENS	Low (2.5)	RISK ACCEPTED
(HAL-03) POOL CAN BE REJECTED OR APPROVED REGARDLESS OF ITS STATUS	Informational (1.5)	SOLVED - 09/15/2023
(HAL-04) CENTRALIZATION RISK: OWNER CAN WITHDRAW TOKENS FROM TOKENKEEPER	Informational (1.5)	ACKNOWLEDGED
(HAL-05) STAKERS CAN NOT WITHDRAW THEIR TOKENS DURING SEEDING PHASE	Informational (1.2)	SOLVED - 09/11/2023
(HAL-06) TEMPLATES WITH DIFFERENT REGISTRY CAN BE ADDED TO THE FACTORY	Informational (1.0)	SOLVED - 09/11/2023
(HAL-07) FOR LOOPS CAN BE GAS OPTIMIZED	Informational (0.0)	SOLVED - 09/11/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) SWAPS OVERWRITE THE ZAP CONTRACT OWNER - MEDIUM (5.0)

Description:

It was identified that the swap function from the AsyncSwapper contract employs the nonReentrant modifier to prevent executing reentrancy attacks on its swap function. However, the swap function is called with a delegatecall from the Zap contract. This results in an unexpected behavior during swaps, as the two contracts have different layouts of state variables, the ReentrancyGuard overwrites the owner of the Zap contract. Without an owner, modifying the Stargate configuration would no longer be possible.

Code Location:

The AsyncSwapper and Zap contracts have different layouts of state variables:

Listing 1: src/swapper/AsyncSwapper.sol (Line 11)

```
11 contract AsyncSwapper is IAsyncSwapper, ReentrancyGuard {
12     address public immutable aggregator;
13
```

Listing 2: ssrc/zap/Zap.sol (Line 19)

```
19 contract Zap is IZap, Ownable2Step {
20     using SafeERC20 for IERC20;
21     using Address for address;
22
23     address public immutable swapper;
24     address public immutable registry;
25     address public immutable stargateRouter;
26     address public immutable tokenKeeper;
```

The swap function is called using a delegatecall from the Zap contract:

Listing 3: src/zap/Zap.sol (Line 214)

```
212 function _swap(SwapParams memory _swapParams) internal returns (
 ↳ uint256) {
213     bytes memory returnedData = swapper.functionDelegateCall(
214         abi.encodeWithSelector(IAsyncSwapper.swap.selector,
 ↳ _swapParams), _delegateSwapFailed
215     );
216     return abi.decode(returnedData, (uint256));
217 }
```

The swap function employs the nonReentrant modifier:

Listing 4: src/swapper/AsyncSwapper.sol (Line 24)

```
20 function swap(SwapParams memory swapParams)
21     public
22     payable
23     virtual
24     nonReentrant
25     returns (uint256 buyTokenAmountReceived)
26 {
```

Proof of Concept:

```
>>> zap.owner()
'0xfCF33bD480610109B491Bc9C53b20Bc4f54aA354'
>>> swapParams = (
    token2, # sellTokenAddress,
    SELL_AMOUNT, # sellAmount,
    token1, # buyTokenAddress,
    1, # buyAmount,
    DATA
)
>>> tx = zap.swapAndStake(swapParams, pool, {'from': alice})
Transaction sent: 0x822bedf10f58358c1a5d51ca2046375cb3eaec4dd18dc2b3764d46e89ed1862d
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 1
  Zap.swapAndStake confirmed   Block: 16731656   Gas used: 657198 (2.19%)

>>> zap.owner()
'0x0000000000000000000000000000000000000001'
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to move the ReentrancyGuard from the AsyncSwapper contract to the Zap contract to avoid overwriting the owner.

Remediation Plan:

SOLVED: The Crystalize team solved the issue in commits e248c4d and 4a69397 by applying the suggested recommendation.

4.2 (HAL-02) INCOMPATIBILITY WITH TRANSFER-ON-FEE OR DEFLATIONARY TOKENS - LOW (2.5)

Description:

It was identified that the Pool and TokenKeeper contracts assume that the safeTransferFrom() calls transfers the full amount of tokens. This may not be true if the tokens being transferred are fee-on-transfer tokens, causing the received amount to be lesser than the accounted amount. For example, DGX (Digix Gold Token) and CGT (CACHE Gold) tokens apply transfer fees, and the USDT (Tether) token also has a currently disabled fee feature.

It was also identified that the contract assumes that its token balance does not change over time without any token transfers, which may not be true if the tokens being transferred were deflationary/rebasing tokens. For example, the supply of AMPL (Ampleforth) tokens automatically increases or decreases every 24 hours to maintain the AMPL target price.

In these cases, the contracts may not have the full rewardAmount amounts, and the associated functions may revert.

Code Location:

Listing 5: src/pool/Pool.sol (Lines 293-299)

```
285 function _stake(address _staker, uint256 _amount) internal {
286     if (_status() != Status.Seeding) revert Error.DepositsDisabled
 ↳ ();
287     if (_amount == 0) revert Error.ZeroAmount();
288     if (balances[_staker] + _amount > maxStakePerAddress) revert
 ↳ Error.MaxStakePerAddressExceeded();
289     if (totalSupply + _amount > maxStakePerPool) revert Error.
 ↳ MaxStakePerPoolExceeded();
290
291     if (balances[_staker] == 0) stakersCount++;
```

```
292
293     totalSupply += _amount;
294     totalSupplyLocked += _amount;
295     balances[_staker] += _amount;
296     balancesLocked[_staker] += _amount;
297
298     emit Staked(_staker, _amount);
299     token.safeTransferFrom(msg.sender, address(this), _amount);
```

Listing 6: src/zap/TokenKeeper.sol (Lines 56-57)

```
48 function transferFromStargateReceiver(address _account, address
↳ _token, uint256 _amount) external {
49     if (msg.sender != stargateReceiver) revert Error.Unauthorized
↳ ();
50     if (_account == address(0)) revert Error.ZeroAddress();
51     if (_token == address(0)) revert Error.ZeroAddress();
52     if (_amount == 0) revert Error.ZeroAmount();
53
54     emit BridgedTokensReceived(_account, _token, _amount);
55
56     balances[_account][_token] += _amount;
57     IERC20(_token).safeTransferFrom(msg.sender, address(this),
↳ _amount);
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:P/S:U (2.5)

Recommendation:

It is recommended to get the exact received amount of the tokens being transferred by calculating the difference of the token balance before and after the transfer and using it to update all the variables correctly.

It is also recommended that all tokens are thoroughly checked and tested before they are used to avoid tokens that are incompatible with the contracts.

Remediation Plan:

RISK ACCEPTED: The Crystalize team made a business decision to accept the risk of this finding and not alter the contracts. The team stated that the owner will only approve Pools with compatible tokens.

4.3 (HAL-03) POOL CAN BE REJECTED OR APPROVED REGARDLESS OF ITS STATUS - INFORMATIONAL (1.5)

Description:

It was identified that the registry can approve or reject pools regardless of their statuses, and the creator can withdraw the tokens from rejected pools using the retrieveRewardToken function. A malicious registry can set the pool's status to Rejected after the Seeding phase to allow the creator to withdraw the tokens deposited by the stakers.

Code Location:

Listing 7: src/pool/Pool.sol

```
205 function approvePool() external onlyRegistry {
206     _underlyingStatus = Status.Approved;
207     emit PoolApproved();
208 }
209
210 /// @inheritdoc IPool
211 function rejectPool() external onlyRegistry {
212     _underlyingStatus = Status.Rejected;
213     emit PoolRejected();
214 }
215
216 /// @inheritdoc IPool
217 function retrieveRewardToken() external onlyCreator {
218     if (_underlyingStatus != Status.Rejected) revert Error.
 ↳ PoolNotRejected();
219     uint256 balance = token.balanceOf(address(this));
220     emit RewardsRetrieved(creator, balance);
221     rewardAmount = 0;
222     feeAmount = 0;
223     token.safeTransfer(creator, balance);
224 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (1.5)

Recommendation:

It is recommended to only allow the registry to reject or approve pools before their Seeding phase to improve decentralization.

Remediation Plan:

SOLVED: The Crystalize team solved the issue in commit 3169deb by only allowing the registry to reject or approve pools before their Seeding phase.

4.4 (HAL-04) CENTRALIZATION RISK: OWNER CAN WITHDRAW TOKENS FROM TOKENKEEPER - INFORMATIONAL (1.5)

Description:

It was identified that the owner can withdraw the tokens stored in the TokenKeeper contract with the pullToken function by changing the zap state variable to their own address.

Code Location:

Listing 8: src/zap/TokenKeeper.sol

```
34 function setZap(address _zap) external onlyOwner {
35     _setZap(_zap);
36 }
```

Listing 9: src/zap/TokenKeeper.sol

```
61 function pullToken(address _token, address _account) external
↳ returns (uint256) {
62     if (msg.sender != zap) revert Error.Unauthorized();
63     return _transferToken(_account, zap, _token);
64 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (1.5)

Recommendation:

It is recommended to employ multi-signature access for high privileged users to improve decentralization.

Remediation Plan:

ACKNOWLEDGED: The Crystalize team acknowledged this finding and will use a multi-signature wallet to access privileged functions to improve decentralization.

4.5 (HAL-05) STAKERS CAN NOT WITHDRAW THEIR TOKENS DURING SEEDING PHASE - INFORMATIONAL (1.2)

Description:

It was identified that stakers cannot unstake their tokens during the Seeding phases, and their deposits remain locked in the Pool contracts until the appropriate Locked phases end. This contradicts the documentation, which states that deposits can also be taken out during the Seeding phases.

Code Location:

Listing 10: src/pool/Pool.sol

```
254 function unstakeAll() external {
255     if (_status() != Status.Unlocked) revert Error.
 ↳ WithdrawalsDisabled();
256
257     uint256 amount = balances[msg.sender];
258     if (amount == 0) revert Error.ZeroAmount();
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:L/Y:N/R:P/S:U (1.2)

Recommendation:

Consider updating the documentation to reflect the current behavior of the Pool contracts.

Remediation Plan:

SOLVED: The Crystalize team solved the issue in commit 0111f64 by updating the documentation to reflect the current behavior of the Pool contracts.

4.6 (HAL-06) TEMPLATES WITH DIFFERENT REGISTRY CAN BE ADDED TO THE FACTORY - INFORMATIONAL (1.0)

Description:

It was identified that the addTemplate function in the PoolFactory contract does not verify that the factory and the pool template have the same registry configured. Because the registry of the pools cannot be changed, and only the factory can add pools to the registry, it may not be possible to approve and start these contracts.

Code Location:

Listing 11: src/pool/PoolFactory.sol

```
134 function addTemplate(address _template) external onlyOwner {
135     if (_template == address(0)) revert Error.ZeroAddress();
136     if (!templates.add(_template)) revert Error.AddFailed();
137     emit TemplateAdded(_template);
138 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (1.0)

Recommendation:

Consider verifying in the addTemplate function that the factory and the pool template have the same registry configured.

Remediation Plan:

SOLVED: The Crystalize team solved the issue in commit 8417703 by applying the recommendation.

4.7 (HAL-07) FOR LOOPS CAN BE GAS OPTIMIZED - INFORMATIONAL (0.0)

Description:

It was identified that the for loops employed in the Lens and Zap contracts can be gas optimized by the following principles:

- A postfix (e.g. i++) operator was used to increment the i variables. It is known that, in loops, using prefix operators (e.g. ++i) costs less gas per iteration than postfix operators.
- It is also possible to further optimize loops by using unchecked loop index incrementing and decrementing.

Note that view or pure functions only cost gas if they are called from on-chain.

Code Location:

Listing 12: src/lens/Lens.sol

```
95 for (uint256 i = 0; i < pools; i++) {
96     poolData[i] = _getPoolData(registry.getPoolAt(i, _pending),
↳ _user);
97 }
```

Listing 13: src/zap/Zap.sol

```
140 for (uint256 i = 0; i < len; ++i) {
141     uint16 chainId = chainIds[i];
142     if (chainId == 0) revert InvalidChainId();
143     // Zero address is ok here to allow for cancelling of chains
144     stargateDestinations[chainId] = destinations[i];
145 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider using the unchecked `++i` operation instead of `i++` to increment the values of the `uint` variable inside the loop. It is noted that using unchecked operations requires particular caution to avoid overflows, and their use may impair code readability.

The following code is an example of the above recommendations:

Listing 14: For Loop Optimization

```
for (uint256 i = 0; i < pools;) {
    poolData[i] = _getPoolData(registry.getPoolAt(i, _pending),
↳ _user);
    unchecked { ++i; }
}
```

Remediation Plan:

SOLVED: The Crystalize team solved the issue in commit `1eab8a8` by using prefix operators. Unchecked operations are not used to improve readability.

DECENTRALIZATION

5.1 DESCRIPTION

In this section, the level of decentralization of the contracts that users interact with is described.

5.2 POOL CONTRACT

Users can deposit their tokens into stake pools and receive rewards. It was identified in the original version of the Pool contract that a malicious creator and registry can withdraw the funds deposited by the stakers by setting the status of an already started pool to Rejected and withdrawing the funds using the retrieveRewardToken function.

The Pool contract was updated in commit 3169deb to prevent the registry from modifying its status after it was started. This change makes the contract decentralized, as after its start, it is not be possible to withdraw the tokens of the users or to change the pool's configuration, including the reward amount, the length of the staking periods or the fee amount. This allows users to verify the pool configuration before staking.

5.3 TOKENKEEPER CONTRACT

The TokenKeeper contract is responsible for storing the tokens received from cross-chain transactions. Users can withdraw their tokens from this contract or deposit them into pools for staking using the Zap contract. It was identified that the F can withdraw the tokens from the TokenKeeper contract with the pullToken function by changing the zap state variable to their own address. This makes the TokenKeeper contract centralized.

The Crystalize team acknowledged this and will use a multi-signature wallet to access privileged functions to improve decentralization. It is noted that the TokenKeeper contract is only intended to store funds for a short period of time.

5.4 ZAP CONTRACT

Users can use the Zap contract to swap or bridge tokens before depositing them into pools for staking. After initialization, the owner can still modify the addresses to which tokens are sent on other chains (Stargate destinations). This makes the Zap contract centralized, as the owner can alter the destination of the users' cross-chain transfers.

The Crystalize team acknowledged this and uses a multi-signature wallet to access privileged functions to improve decentralization. It is noted that the Zap contract does not store any funds.

MANUAL TESTING

In the manual testing phase, the following scenarios were simulated. The scenarios listed below were selected based on the severity of the vulnerabilities Halborn was testing the contracts for.

6.1 ACCESS CONTROL AND ROLE MANAGEMENT

Description:

Proper access control on privileged functions was tested. All functions were reviewed to ensure that no sensitive functionality was left unprivileged.

Results:

It was verified that all sensitive functions in the contracts are protected with proper authorization controls.

6.2 POOL CONTRACTS

It was checked that the functions of the Pool, PoolFactory and Registry contracts are working as intended.

Results:

- It was verified that the pools created by the PoolFactory contract are properly initialized.
- It was verified that the lifecycle transitions of the pools are working as intended.
- It was verified that the pool starting fee is transferred to the treasury.
- It was verified that the creator can withdraw the deposited reward from the pool if it was rejected by the registry.

- It was verified that stakers cannot take out their staked tokens during the Seeding and Locked phases.
- It was verified that stakers can take out their staked tokens during the Unlocked phase.
- It was verified that the earned rewards and the pool starting fee were calculated correctly.
- It was verified that users can take out their earned rewards after the Seeding phase starts.
- It was found that the pools are incompatible with transfer-on-fee and deflationary tokens.

In the following example, it was examined that the properties of the pool configured properly by the factory during deployment:

```
>>> tx = factory.createPool(
        pool_impl1,                 # _template
        token,                      # _token,
        SEEDING_PERIOD,             # _seedingPeriod,
        LOCK_PERIOD,                # _lockPeriod,
        rewardAmount,               # _rewardAmount
        MAX_STAKE_PER_ADDRESS,      # _maxStakePerAddress,
        MAX_STAKE_PER_POOL,         # _maxStakePerPool
        {'from': creator}
)
Transaction sent: 0x9612bd47880d3ac045eeece4e608595d243e296ddd71cd52365f8bd495632136
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 1
  PoolFactory.createPool confirmed   Block: 16731658   Gas used: 389837 (1.30%)

>>> pool = Pool.at(tx.return_value)
>>> pool.creator() == creator
True
>>> pool.token() == token
True
>>> token.balanceOf(pool) == pool.rewardAmount() + pool.feeAmount()
True
```

In the following example, the lifecycle transition of the pool was observed from Locked to Unlocekd:

```
>>> jumpTo(_periodFinish-100)
>>> poolstatus[pool.status()]
'Locked'
>>> tx = pool.unstakeAll({'from': alice})
Transaction sent: 0xd7c96a64dc5686264145e7dcb57c261b92e53bdb1367a2064b6984fc874ff6e1
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 4
  Pool.unstakeAll confirmed (typed error: 0x46ee9e35)   Block: 16731659   Gas used: 28264 (0.09%)

>>> errors['0x46ee9e35']
'WithdrawalsDisabled'
>>> chain.sleep(105)
>>> chain.mine()
16731660
>>> poolstatus[pool.status()]
'Unlocked'
>>> pool.balances(alice)
2000000000000000000
>>> tx = pool.unstakeAll({'from': alice})
Transaction sent: 0x73e2b7c0e57cfd2188d2e13dea01842922463e1886e5c98fb95fb1d53cffdf3b
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 5
  Pool.unstakeAll confirmed   Block: 16731661   Gas used: 46420 (0.15%)

>>> printDictionary(tx.events['Transfer'])
from : 0x579589e46C5d6893Bc4fd6AE067f88B5F6f8f431
to   : 0xD72342083d0201350AE2F5d1995D1C60502c19c3
value: 2000000000000000000
```

6.3 ZAP CONTRACTS

It was checked that the functions of the Zap, TokenKeeper and StargateReceiver contracts are working as intended, and it is possible to use them to stake tokens from other chains or by utilizing token swaps.

Results:

- It was possible to swap and stake tokens employing the AsyncSwapper in the Zap contract. However, it was identified that the operation overwrites the owner of the Zap contract because of the invalid use of delegatecalls.
- It was verified that the swap parameters applied properly, and the contract reverts if it does not receive at least the minimum buy

amount of tokens.

- It was verified that the Zap contract properly pulls the tokens from the TokenKeeper contract when receiving tokens from Stargate.

In the following example, tokens were swapped and staked through the Zap contract:

```
>>> pool.balances(alice)
0
>>> tx = zap.swapAndStake(swapParams, pool, {'from': alice})
Transaction sent: 0x822bedf10f58358c1a5d51ca2046375cb3eaec4dd18dc2b3764d46e89ed1862d
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 1
  Zap.swapAndStake confirmed   Block: 16731656   Gas used: 657198 (2.19%)
>>> printDictionary(tx.events['Staked'])
account: 0xe3855324B960Fe3Da8c3764B5DEbB1ce1280f098
amount : 441298210681499322546
>>> pool.balances(alice)
441298210681499322546
```

AUTOMATED TESTING

7.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

Slither did not list any vulnerabilities from the contracts, as the Crystalize team previously reviewed the results and disabled the false positive findings by using indicators in the source code.

7.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

pool/Registry.sol

Report for src/pool/Registry.sol
https://dashboard.mythx.io/#/console/analyses/23fa450f-dc42-4df8-bcf5-f70bf14730bd

Line	SWC Title	Severity	Short Description
76	(SWC-107) Reentrancy	Low	A call to a user-supplied address is executed.

pool/PoolFactory.sol

MythX did not identify any vulnerabilities in the contract.

pool/Pool.sol

MythX did not identify any vulnerabilities in the contract.

librairies/ERC20Utils.sol

MythX did not identify any vulnerabilities in the contract.

librairies/Error.sol

MythX did not identify any vulnerabilities in the contract.

lens/Lens.sol

MythX did not identify any vulnerabilities in the contract.

swapper/AsyncSwapper.sol

MythX did not identify any vulnerabilities in the contract.

zap/TokenKeeper.sol

Report for src/zap/TokenKeeper.sol
https://dashboard.mythx.io/#/console/analyses/4f25c478-1607-4c32-bbb0-ce5af8e5b94c

Line	SWC Title	Severity	Short Description
12	(SWC-123) Requirement Violation	Low	Requirement violation.

zap/Zap.sol

MythX did not identify any vulnerabilities in the contract.

stargate/StargateReceiver.sol

MythX did not identify any vulnerabilities in the contract.

The findings obtained as a result of the MythX scan were examined, and they were not included in the report because they were determined false positives.

THANK YOU FOR CHOOSING

// HALBORN