OBELISK

Part of Tibereum Group

AUDITING REPORT

Version Notes

Version	No. Pages	Date	Revised By	Notes
1.0	Total:	YYYY-MM-DD	Zapmore, Auditor1	Audit Draft

Audit Notes

Audit Date	YYYY-MM-DD - YYYY-MM-DD
Auditor/Auditors	Auditor1, Auditor2
Auditor/Auditors Contact Information	contact@obeliskauditing.com
Notes	Specified code and contracts are audited for security flaws. UI/UX (website), logic, team, and tokenomics are not audited.
Audit Report Number	OB5XXXXXXXX

Disclaimer

This audit is not financial, investment, or any other kind of advice and is for informational purposes only. This report is not a substitute for doing your own research and due diligence. Obelisk is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Obelisk has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Obelisk is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. The audit is paid by the project but neither the auditors nor Obelisk has any other connection to the project and has no obligations other than to publish an objective report. Obelisk will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Obelisk assumes that the provided information and material were not altered, suppressed, or misleading. This report is published by Obelisk, and Obelisk has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Obelisk.

Obelisk Auditing

Defi is a relatively new concept but has seen exponential growth to a point where there is a multitude of new projects created every day. In a fast-paced world like this, there will also be an enormous amount of scams. The scams have become so elaborate that it's hard for the common investor to trust a project, even though it could be legit. We saw a need for creating high-quality audits at a fast phase to keep up with the constantly expanding market. With the Obelisk stamp of approval, a legitimate project can easily grow its user base exponentially in a world where trust means everything. Obelisk Auditing consists of a group of security experts that specialize in security and structural operations, with previous work experience from among other things, PricewaterhouseCoopers. All our audits will always be conducted by at least two independent auditors for maximum security and professionalism.

As a comprehensive security firm, Obelisk provides all kinds of audits and project assistance.

Audit Information

The auditors always conducted a manual visual inspection of the code to find security flaws that automatic tests would not find. Comprehensive tests are also conducted in a specific test environment that utilizes exact copies of the published contract.

While conducting the audit, the Obelisk security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Obelisk assesses the risks and assigns a risk level to each section together with an explanatory comment. Take note that the comments from the project team are their opinion and not the opinion of Obelisk.

Table of Contents

Project Information

Name	
Description	
Website	
Contact	
Contact information	@XXXX on TG
Token Name(s)	N/A
Token Short	N/A
Contract(s)	See Appendix A
Code Language	Solidity
Chain	Polygon / BSC

Audit of ShadeCash

The main takeaway will be added here after the audit is completed and the final draft is created.

Obelisk was commissioned by XXXX on the XXXX th of XXXX 2021 to conduct a comprehensive audit of XXXX' contracts. The following audit was conducted between the XXXXth of XXXX 2021 and the XXXXth of XXXX 2021. Two of Obelisk's security experts went through the related contracts manually using industry standards to find if any vulnerabilities could be exploited either by the project team or users.

Findings and other relevant info will be updated at audit completion and added here.

The informational findings are good to know while interacting with the project but don't directly damage the project in its current state, hence it's up to the project team if they deem that it's worth solving these issues.

The team has not reviewed the UI/UX, logic, team, or tokenomics of the XXXX project**.**

Please read the full document for a complete understanding of the audit.

Summary Table

Finding	ID	Severity	Status
Withdraw Function Never Withdraws	#0001	High Risk	Closed
SafeTransfer Does Not Require Approval	#0002	High Risk	Closed
Lock Array Recreated Every Time Locks Change	#0003	High Risk	Closed
Unbound Loops On User Locks	#0004	Medium Risk	Closed
Checking Whether The Penalty Receiver Is A Contract Is Done Manually	#0005	Low Risk	Closed
Rounding Precision	#0006	Low Risk	Closed
Inconsistent Logic	#0007	Low Risk	Closed
Global Id For Lock	#0008	Informational	Closed
Blank Elements May Be Left In Array	#0009	Low Risk	Closed
Redundant Division And Multiplication	#0010	Informational	Closed
Loops Can Be Combined	#0011	Informational	Closed
Unbounded Loop	#0012	Informational	Closed
Lock Is Never Removed	#0013	High Risk	Closed
Protocol Values Should Be Public	#0014	Informational	Closed
Mixed Tab and Space Indentation	#0015	Informational	Closed

Findings

Manual Analysis

Withdraw Function Never Withdraws

FINDING ID	#0001
SEVERITY	High Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 312-326

```
1     function withdraw(uint256 amount) public nonReentrant {
2         require(amount != 0, "Can't withdraw 0");
3
4         _updateUserLocks(msg.sender);
5         _updateReward(msg.sender);
6         _claimReward(msg.sender);
7
8         Balances storage bal = balances[msg.sender];
9         require(amount <= bal.total - bal.locked, "Not enough
  unlocked tokens to withdraw");
10
11        bal.total -= amount;
12
13        _sendTokensAndPenalty(amount, 0);
14
15        emit Withdrawn(msg.sender, amount);
16    }
```

DESCRIPTION	The function is missing the actual transfer. This would wipe user's balance without sending the user any money.
RECOMMENDATION	Add a *safeTransfer()*.
RESOLUTION	A transfer call was added.

SafeTransfer Does Not Require Approval

FINDING ID	#0002
SEVERITY	High Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 511-525

```
1     function _sendTokensAndPenalty(uint256 tokensAmount, uint256
  penaltyAmount) internal {
2         if (penaltyAmount != 0 && address(penaltyReceiver) !=
  address(0)) {
3             if (penaltyReceiverIsContract) {
4                 stakingToken.approve(address(penaltyReceiver),
  penaltyAmount);
5                 penaltyReceiver.notifyReward(penaltyAmount);

6             } else {
7                 stakingToken.safeTransfer(address(penaltyReceiver),
  penaltyAmount);
8             }
9             emit PenaltyPaid(msg.sender, penaltyAmount);
10            stakingToken.safeTransfer(msg.sender, tokensAmount);
11        } else {
12            stakingToken.safeTransfer(msg.sender, tokensAmount +
  penaltyAmount);
13        }
14        totalSupply -= (tokensAmount + penaltyAmount);
15    }
```

DESCRIPTION	An unnecessary approval is dangerous and should not be in the code. Approving the same amount as the transfer amount will allow the recipient to withdraw the amount themselves again.
RECOMMENDATION	Do not approve for a *safeTransfer*.
RESOLUTION	The call to *approve()* was removed.

Lock Array Recreated Every Time Locks Change

FINDING ID	#0003
SEVERITY	High Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol

DESCRIPTION	Multiple function will update the lock array by creating a new array, copying old values into this array, then replacing the old array. This is a very error prone operation, given how frequently it occurs. Furthermore, the gas costs of updating storage memory can be significant.
RECOMMENDATION	Simplify the logic and save gas by just shifting every value to the left and pop the top of the array. Since random gaps are created only in withdrawLock(uint256 id) shift values there instead as no random gaps have yet to be created. Then you can simply shift without any random gaps in the update function. Due to the complexity withdrawLock(uint256 id) introduces, implementing another datastructure such as OrderedSet, RenounceableQueue or a Linked List could be more beneficial.
RESOLUTION	Locks are not updated consistently. The new implementation uses mappings to avoid re-creating the entire array.

Unbound Loops On User Locks

FINDING ID	#0004
SEVERITY	Medium Risk
STATUS	Closed
LOCATION	• Rev 2 - ShadeStaker.sol -> 167-179: *for (uint i = 0; i < length; i++) {* • Rev 2 - ShadeStaker.sol -> 233-237: *for (uint i = 0; i < locks.length; i++) {* • Rev 2 - ShadeStaker.sol -> 241-247: *for (uint i = 0; i < locks.length; i++) {* • Rev 2 - ShadeStaker.sol -> 371-371: *for (uint i = 0; i < locks.length; i++) {* • Rev 2 - ShadeStaker.sol -> 383-385: *for (uint i = 0; i < lockedCount; i++) {* • Rev 2 - ShadeStaker.sol -> 458-470: *for (uint i = 0; i < locks.length; i++) {* • Rev 2 - ShadeStaker.sol -> 476-478: *for (uint i = 0; i < lockedCount; i++) {*

DESCRIPTION	Certain contracts can add an unlimited number of locks for a user. Iterating over an unbounded array can cause transactions to revert due to the gas limit.
RECOMMENDATION	Provide a limit to the size of the array.
RESOLUTION	A hard limit of 14 locks via the *lockDuration* limits the number of locks.

Checking Whether The Penalty Receiver Is A Contract Is Done Manually

FINDING ID	#0005
SEVERITY	Low Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 513-518

```
            if (penaltyReceiverIsContract) {
                stakingToken.approve(address(penaltyReceiver), penaltyAmount);
                penaltyReceiver.notifyReward(penaltyAmount);

            } else {
                stakingToken.safeTransfer(address(penaltyReceiver), penaltyAmount);
            }
```

LOCATION	Rev 2 - ShadeStaker.sol -> 104-108

```
    function setPenaltyReceiver(IPenaltyReceiver newPenaltyReceiver, bool isContract) public onlyOwner {
        penaltyReceiver = newPenaltyReceiver;
        penaltyReceiverIsContract = isContract;
        emit SetPenaltyReceiver(address(newPenaltyReceiver), isContract);
    }
```

DESCRIPTION	A boolean parameter is used to determine if the *penaltyReceiver* is a contract.
RECOMMENDATION	Use *isContract* to check if it's a contract. Do note the dangers with this function as described in the implementation comments: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol#L36
RESOLUTION	The project team has implemented the recommended change.

Rounding Precision

FINDING ID	#0006
SEVERITY	Low Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol ->286

```
1            uint256 unlockTime = (block.timestamp / rewardsDuration *
  rewardsDuration) + lockDuration;
```

LOCATION	Rev 2 - ShadeStaker.sol -> 348-349 Rev 2 - ShadeStaker.sol -> 389-390

```
1        uint256 penalty = amount / 2;
2        amount -= penalty;
```

DESCRIPTION	A number of locations in the contract, including the ones noted above, have potential rounding errors. For example, at line 286, the math might cause the following rounding: (123 /10 * 10) +2 = 122
RECOMMENDATION	Ensure that the correct precision is used in all arithmetic operations.
RESOLUTION	Project team has stated that the rounding behaviour is intentional.

Inconsistent Logic

FINDING ID	#0007
SEVERITY	Low Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 168

```
1                   if (locks[i].unlockTime > block.timestamp) {
```

LOCATION	Rev 2 - ShadeStaker.sol -> 288

```
1             if (locksLength == 0 || userLocks[account][locksLength-
  1].unlockTime < unlockTime) {
```

LOCATION	Rev 2 - ShadeStaker.sol -> 448

```
1         if (locks[length-1].unlockTime <= block.timestamp) {
```

DESCRIPTION	The noted conditionals are almost the same, but subtly different. In contracts with significant branching, it is important that the conditionals used are clear and easy to understand. In particular, the third one noted (line 288) uses the < operator as opposed to the <= operator.
RECOMMENDATION	Use the same conditional statement for consistency, wherever possible. In this case, also confirm whether the discrepancies are intentional.
RESOLUTION	Operators have been changed to be more consistent.

Global Id For Lock

FINDING ID	#0008
SEVERITY	Informational
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 290-294

```
1                userLocks[account].push(LockedBalance({
2                    amount: amount,
3                    unlockTime: unlockTime,
4                    id: lockIds
5                }));
```

DESCRIPTION	Every deposit is assigned a unique lock. Yet this lockid is never used to fetch the individual lock. Currently, there is no way to check what funds are in a given lock on chain.
RECOMMENDATION	Remove the lock id mechanism as they add unnecessary complexity.
RESOLUTION	Withdrawing on a per-lock basis has been removed.

Blank Elements May Be Left In Array

FINDING ID	#0009
SEVERITY	Low Risk
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 458-482

```
for (uint i = 0; i < length; i++) {
        if (locks[i].unlockTime > block.timestamp) {
            // if lock not expired adding amount to total locked
            lockedAmount = lockedAmount + locks[i].amount;
            if (length > lockDurationMultiplier) {
                newLocks[lockedCount] = locks[i];
                lockedCount ++;
            }
        } else {
            // if expired delete it
            delete locks[i];
        }
    }

    // let's get rid of empty locks (gaps) on the beginning of array if they are
    // the reason is to not allow array to grow
    if (length > lockDurationMultiplier && lockedCount != 0 && length > lockedCount) {
        delete userLocks[account];
        for (uint i = 0; i < lockedCount; i++) {
            userLocks[account].push(newLocks[i]);
        }
    }

    bal.locked = lockedAmount;
```

DESCRIPTION	The if statement checks the *lockDurationMultiplier* and compares it to the number of locks at the start of the function. There is a likelihood that the user locks will not be correctly updated to use the new locks.
RECOMMENDATION	Clarify the logic of updating the locks.
RESOLUTION	Project team simplified the check logic such that it will always remove blank elements.

Redundant Division And Multiplication

FINDING ID	#0010
SEVERITY	Informational
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 286

```
1            uint256 unlockTime = (block.timestamp / rewardsDuration *
  rewardsDuration) + lockDuration;
```

DESCRIPTION	The division is cancelled out by the multiplication.
RECOMMENDATION	Remove the redundant operations or confirm whether this behaviour is intended.
RESOLUTION	The project team has confirmed that this behaviour is intended.

Loops Can Be Combined

FINDING ID	#0011
SEVERITY	Informational
STATUS	Closed
LOCATION	Rev 2 - ShadeStaker.sol -> 233-248

```
1             for (uint i = 0; i < locks.length; i++) {
2                 if (locks[i].amount != 0 && locks[i].unlockTime >
  block.timestamp) {
3                     locksCount ++;
4                 }
5             }
6             LockedBalance[] memory _userLocks = new LockedBalance[]
  (locksCount);
7             if (locksCount != 0) {
8                 uint256 idx;
9                 for (uint i = 0; i < locks.length; i++) {
10                    if (locks[i].amount != 0 && locks[i].unlockTime >
  block.timestamp) {
11                        _userLocks[idx] = locks[i];
12                        _balances.locked += locks[i].amount;
13                        idx ++;
14                    }
15                }
16            }
```

DESCRIPTION	These loops are nearly identical.
RECOMMENDATION	Combine the functionality of the loops. Before the array is looped through the first time, create a tempArray. Instead of incrementing *locksCount*, push to that tempArray and you will end up with a *tempArray* of equal length without having to loop through it again.
RESOLUTION	The loops were combined.

Unbounded Loop

FINDING ID	#0012
SEVERITY	Informational
STATUS	Closed
LOCATION	• Rev 2 - ShadeStaker.sol -> 145-148: *for (uint256 i = 0; i < rewardsAvailable.length; i++) {* • Rev 2 - ShadeStaker.sol -> 201-205: *for (uint i; i < rewardTokens.length; i++) {* • Rev 2 - ShadeStaker.sol -> 499-507: *for (uint i; i < rewardTokens.length; i++) {*

DESCRIPTION	Iterating over an unbounded array can cause transactions to revert due to the gas limit.
RECOMMENDATION	Provide a limit to the size of the array. Alternatively pass a lower and upper index as parameters and iterate over a range.
RESOLUTION	An upper bound of 10 tokens was added.

Lock Is Never Removed

FINDING ID	#0013
SEVERITY	High Risk
STATUS	Closed
LOCATION	Rev 3 - ShadeStaker.sol -> 442-453

```
1         LockedBalance[] memory locks = userLocks[msg.sender];
2
3         uint256 amount;
4
5         // AUDIT Finding Id: 4
6         // length can't be more than lockDurationMultiplier (13) + 1
7         for (uint i = 0; i < locks.length; i++) {
8             if (locks[i].id == id) {
9                 amount = locks[i].amount;
10                delete locks[i];
11            }
12        }
```

DESCRIPTION	A local copy of *locks* is made (*userLocks[msg.sender]*). Then a lock in the local copy is deleted. (*delete locks[i];*) The storage lock is thus never deleted.
RECOMMENDATION	Change the reference type from memory to storage.
RESOLUTION	The lock mechanism was changed to use mapping, resolving this issue.

Protocol Values Should Be Public

FINDING ID	#0014
SEVERITY	Informational
STATUS	Closed
LOCATION	• Rev 4 - ShadeStaker_3_mapping.sol -> 442-453: mapping(address=> mapping(address => bool)) public rewardDistributors • Rev 4 - ShadeStaker_3_mapping.sol -> 442-453: mapping(address => bool) public lockStakers

DESCRIPTION	Variables critical to the operation of the protocol should be public or have an associated view function. Mappings are not iterable and therefore it may be challenging to identify all the distributors and lockStakers.
RECOMMENDATION	Add an array which mirrors the contents of the mapping.
RESOLUTION	Arrays was added which tracks all addresses which ever received the elevated permissions.

Mixed Tab and Space Indentation

FINDING ID	#0015
SEVERITY	Informational
STATUS	Closed
LOCATION	Rev 4 - ShadeStaker_3_mapping.sol

DESCRIPTION	The contract uses mixed tab and space indentation. This can cause the contract indentation to appear “incorrect” (for example on github).
RECOMMENDATION	Use a consistent indentation method.
RESOLUTION	The project team has implemented the recommended change.

Static Analysis

No Findings

On-Chain Analysis

Not Analyzed Yet

External Addresses

Externally Owned Accounts

Owner

ACCOUNT	Address
USAGE	0x... *LPStaker.owner* - Variable
IMPACT	• receives elevated permissions as owner, operator, or other

External Contracts

These contracts are not part of the audit scope.

Staking Token

ADDRESS	
USAGE	0x... *ShadeStaker.stakingToken* - Immutable
IMPACT	• ERC20 Token

WETH

ADDRESS	
USAGE	0x... *ShadeStaker.WETH* - Immutable
IMPACT	• ERC20 Token

Reward Tokens

ADDRESS	
USAGE	0x... *ShadeStaker.rewardTokens* - Variable
IMPACT	• ERC20 Token

Penalty Receiver

ADDRESS	0x8b7bcce67d2566D26393A6b81cAE010762C196B2
USAGE	0x... *ShadeStaker.penaltyReceiver* - Variable
IMPACT	• receives transfer of tokens deposited or minted by project

Reward Distributors

ADDRESS	
USAGE	0x... *ShadeStaker.rewardDistributors* - Variable
IMPACT	• receives elevated permissions as owner, operator, or other

Lock Stakers

ADDRESS	
USAGE	0x... *ShadeStaker.lockStakers* - Variable
IMPACT	• receives elevated permissions as owner, operator, or other

Appendix A - Reviewed Documents

Document	**Address**
ShadeStaker.sol	N/A

Revisions

Revision 1	Zip file
Revision 2	96048adc10c1d304feaef3bb5d01960dcb0f7a5f
Revision 3	0a27b3adb9b30f5aa9a713899ad71c3729579e81
Revision 4	9b413a875788ca04f5c08f7a03397eabe76228f2
Revision 5	b402df61e6ecf03c31238e24067bf1966d683f5c

Imported Contracts

Contracts	Version

Appendix B - Risk Ratings

Risk	Description
High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
Low Risk	A vulnerability which can cause the loss of protocol functionality.
Informational	Non-security issues such as functionality, style, and convention.

Appendix C - Finding Statuses

Closed	Contracts were modified to permanently resolve the finding.
Mitigated	The finding was resolved by other methods such as revoking contract ownership. The issue may require monitoring, for example in the case of a time lock.
Partially Closed	Contracts were updated to fix the issue in some parts of the code.
Partially Mitigated	Fixed by project specific methods which cannot be verified on chain. Examples include compounding at a given frequency.
Open	The finding was not addressed.

Appendix D - Audit Procedure

A typical Obelisk audit uses a combination of the three following methods:

Manual analysis consists of a direct inspection of the contracts to identify any security issues. Obelisk auditors use their experience in software development to spot vulnerabilities. Their familiarity with common contracts allows them to identify a wide range of issues in both forked contracts as well as original code.

Static analysis is software analysis of the contracts. Such analysis is called “static” as it examines the code outside of a runtime environment. Static analysis is a powerful tool used by auditors to identify subtle issues and to verify the results of manual analysis.

On-chain analysis is the audit of the contracts as they are deployed on the block-chain. This procedure verifies that:

- deployed contracts match those which were audited in manual/static analysis;
- contract values are set to reasonable values;
- contracts are connected so that interdependent contract function correctly;
- and the ability to modify contract values is restricted via a timelock or DAO mechanism. (We recommend a timelock value of at least 72 hours)

Each obelisk audit is performed by at least two independent auditors who perform their analysis separately.

After the analysis is complete, the auditors will make recommendations for each issue based on best practice and industry standards. The project team can then resolve the issues, and the auditors will verify that the issues have been resolved with no new issues introduced.

Our auditing method lays a particular focus on the following important concepts:

- Quality code and the use of best practices, industry standards, and thoroughly tested libraries.
- Testing the contract from different angles to ensure that it works under a multitude of circumstances.
- Referencing the contracts through databases of common security flaws.

Follow Obelisk Auditing for the Latest Information

 ObeliskOrg

 ObeliskOrg

OBELISK

Part of Tibereum Group