MythX

REPORT 5F42F0FC0C10D4001222E9C0

Created	Sun Aug 23 2020 22:43:08 GMT+0000 (Coordinated Universal Time)
Number of analyses	3
User	jonahsparklemobile@gmail.com

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
4115acf1-90aa-4c75-88ad-62199a1a494a	contracts/SparkleTimestamp.sol	0
287859dc-0d53-4bda-863c-021f09f5942b	contracts/SparkleLoyalty.sol	2
b6a808f2-eb5b-401d-934b-8d1d5287fdbd	contracts/SparkleRewardTiers.sol	0

Analysis 4115acf1-90aa-4c75-88ad-62199a1a494a MythX

Started	Sun Aug 23 2020 22:43:18 GMT+0000 (Coordinated Universal Time)
Finished	Sun Aug 23 2020 22:58:30 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Mythx-Cli-0.6.19
Main Source File	Contracts/SparkleTimestamp.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	0

ISSUES

Analysis 287859dc-0d53-4bda-863c-021f09f5942b MythX

Started	Sun Aug 23 2020 22:43:18 GMT+0000 (Coordinated Universal Time)
Finished	Sun Aug 23 2020 22:58:32 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Mythx-Cli-0.6.19
Main Source File	Contracts/SparkleLoyalty.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	2

ISSUES

LOW

SWC-123

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Source file
contracts/SparkleLoyalty.sol
Locations

```
484 returns (uint256, bool, uint256)
485 {
486 (uint256 remaining, bool status, uint256 deposit) = ISparkleTimestamp(timestampAddress).getTimeRemaining(_loyaltyAddress);
487 return (remaining, status, deposit);
488 }
```

Source file
contracts/SparkleLoyalty.sol
Locations

```
484 returns (uint256, bool, uint256)
485 {
486 (uint256 remaining, bool status, uint256 deposit) = ISparkleTimestamp(timestampAddress).getTimeRemaining(_loyaltyAddress);
487 return (remaining, status, deposit);
488 }
```

LOW

SWC-123

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Source file

contracts/SparkleLoyalty.sol

Locations

```
187 require(msg.sender != address(0), 'Invalid {from}');
188 // Validate caller has a timestamp and it has matured
189 require(ISparkleTimestamp(timestampAddress).hasTimestamp(msg.sender), 'No record');
190 require(ISparkleTimestamp(timestampAddress).isRewardReady(msg.sender), 'Not mature');
```

Source file

contracts/SparkleLoyalty.sol

Locations

```
16 * @author SparkleMobile Inc.
17 */
18 contract SparkleLoyalty is Ownable, Pausable, ReentrancyGuard {
19
20 /**
21 * @dev Ensure math safety through SafeMath
22 */
23 using SafeMath for uint256;
24
25 // Gas to send with certain transations that may cost more in the future due to chain growth
26 uint256 private gasToSendWithTX = 25317;
27 // Base rate APR (5%) factored to 365.2422 gregorian days
28 uint256 private baseRate = 0.00013690 * 10e7; // A full year is 365.2422 gregorian days (5%)
29
30 // Account data structure
31 struct Account {
32 address _address; // Loyalty reward address
33 uint256 _balance; // Total tokens deposited
34 uint256 _collected; // Total tokens collected
35 uint256 _claimed; // Total succesfull reward claims
36 uint256 _joined; // Total times address has joined
37 uint256 _tier; // Tier index of reward tier
38 bool _isLocked; // Is the account locked
39 }
40
41 // tokenAddress of erc20 token address
42 address private tokenAddress;
43
44 // timestampAddress of time stamp contract address
45 address private timestampAddress;
46
47 // treasuryAddress of token treeasury address
48 address private treasuryAddress;
49
50 // collectionAddress to receive eth payed for tier upgrades
51 address private collectionAddress;
52
53 // rewardTiersAddress to resolve reward tier specifications
54 address private tiersAddress;
55
56 // minProofRequired to deposit of rewards to be eligibile
57 uint256 private minRequired;
58
59 // maxProofAllowed for deposit to be eligibile
60 uint256 private maxAllowed;
61
```

```
// totalTokensClaimed of all rewards awarded
uint256 private totalTokensClaimed;

// totalTimesClaimed of all successfully claimed rewards
uint256 private totalTimesClaimed;

// totalActiveAccounts count of all currently active addresses
uint256 private totalActiveAccounts;

// Accounts mapping of user loyalty records
mapping(address => Account) private accounts;

/**
* @dev Sparkle Loyalty Rewards Program contract .cTor
* @param _tokenAddress of token used for proof of loyalty rewards
* @param _treasuryAddress of proof of loyalty token reward distribution
* @param _collectionAddress of ethereum account to collect tier upgrade eth
* @param _tiersAddress of the proof of loyalty tier rewards support contract
* @param _timestampAddress of the proof of loyalty timestamp support contract
*/
constructor(address _tokenAddress, address _treasuryAddress, address _collectionAddress, address _tiersAddress, address _timestampAddress)
public
Ownable()
Pausable()
ReentrancyGuard()
{
// Initialize contract internal addresse(s) from params
tokenAddress = _tokenAddress;
treasuryAddress = _treasuryAddress;
collectionAddress = _collectionAddress;
tiersAddress = _tiersAddress;
timestampAddress = _timestampAddress;

// Initialize minimum/maximum allowed deposit limits
minRequired = uint256(1000).mul(10e7);
maxAllowed = uint256(250000).mul(10e7);
}

/**
* @dev Deposit additional tokens to a reward address loyalty balance
* @param _depositAmount of tokens to deposit into a reward address balance
* @return bool indicating the success of the deposit operation (true == success)
*/
function depositLoyalty(uint _depositAmount)
public
whenNotPaused
nonReentrant
returns (bool)
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}1');
// Validate specified value meets minimum requirements
require(_depositAmount >= minRequired, 'Minimum required');

// Determine if caller has approved enough allowance for this deposit
if(IERC20(tokenAddress).allowance(msg.sender, address(this)) < _depositAmount) {
// No, rever informing that deposit amount exceeded allownce amount
revert('Exceeds allowance');
}

// Obtain a storage instsance of callers account record
Account storage loyaltyAccount = accounts[msg.sender];

```

```
// Determine if there is an upper deposit cap
if(maxAllowed > 0) {
// Yes, determine if the deposit amount + current balance exceed max deposit cap
if(loyaltyAccount._balance.add(_depositAmount) > maxAllowed || _depositAmount > maxAllowed) {
// Yes, revert informing that the maximum deposit cap has been exceeded
revert('Exceeds cap');
}

}

// Determine if the tier selected is enabled
if(!ISparkleRewardTiers(tiersAddress).getEnabled(loyaltyAccount._tier)) {
// No, then this tier cannot be selected
revert('Invalid tier');
}

// Determine of transfer from caller has succeeded
if(IERC20(tokenAddress).transferFrom(msg.sender, address(this), _depositAmount)) {
// Yes, thend determine if the specified address has a timestamp record
if(ISparkleTimestamp(timestampAddress).hasTimestamp(msg.sender)) {
// Yes, update callers account balance by deposit amount
loyaltyAccount._balance = loyaltyAccount._balance.add(_depositAmount);
// Reset the callers reward timestamp
_resetTimestamp(msg.sender);
//
emit DepositLoyaltyEvent(msg.sender, _depositAmount, true);
// Return success
return true;
}

// Determine if a timestamp has been added for caller
if(!ISparkleTimestamp(timestampAddress).addTimestamp(msg.sender)) {
// No, revert indicating there was some kind of error
revert('No timestamp created');
}

// Prepare loyalty account record
loyaltyAccount._address = address(msg.sender);
loyaltyAccount._balance = _depositAmount;
loyaltyAccount._joined = loyaltyAccount._joined.add(1);
// Update global account counter
totalActiveAccounts = totalActiveAccounts.add(1);
//
emit DepositLoyaltyEvent(msg.sender, _depositAmount, false);
// Return success
return true;
}

// Return failure
return false;
}

/**
* @dev Claim Sparkle Loyalty reward
*/
function claimLoyaltyReward()
public
whenNotPaused
nonReentrant
returns(bool)
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
```

```
// Validate caller has a timestamp and it has matured
require(ISparkleTimestamp(timestampAddress).hasTimestamp(msg.sender), 'No record');
require(ISparkleTimestamp(timestampAddress).isRewardReady(msg.sender), 'Not mature');

// Obtain the current state of the callers timestamp
(uint256 timeRemaining, bool isReady, uint256 rewardDate) = ISparkleTimestamp(timestampAddress).getTimeRemaining(msg.sender);
// Determine if the callers reward has matured
if(isReady) {
// Value not used but throw unused var warning (cleanup)
rewardDate = 0;
// Yes, then obtain a storage instance of callers account record
Account storage loyaltyAccount = accounts[msg.sender];
// Obtain values required for caculations
uint256 dayCount = (timeRemaining.div(ISparkleTimestamp(timestampAddress).getTimePeriod())).add(1);
uint256 tokenBalance = loyaltyAccount._balance.add(loyaltyAccount._collected);
uint256 rewardRate = ISparkleRewardTiers(tiersAddress).getRate(loyaltyAccount._tier);
uint256 rewardTotal = baseRate.mul(tokenBalance).mul(rewardRate).mul(dayCount).div(10e7).div(10e7);
// Increment collected by reward total
loyaltyAccount._collected = loyaltyAccount._collected.add(rewardTotal);
// Increment total number of times a reward has been claimed
loyaltyAccount._claimed = loyaltyAccount._claimed.add(1);
// Incrememtn total number of times rewards have been collected by all
totalTimesClaimed = totalTimesClaimed.add(1);
// Increment total number of tokens claimed
totalTokensClaimed += rewardTotal;
// Reset the callers timestamp record
_resetTimestamp(msg.sender);
// Emit event log to the block chain for future web3 use
emit RewardClaimedEvent(msg.sender, rewardTotal);
// Return success
return true;
}

// Revert opposed to returning boolean (May or may not return a txreceipt)
revert('Failed claim');
}

/**
* @dev Withdraw the current deposit balance + any earned loyalty rewards
*/
function withdrawLoyalty()
public
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
// validate that caller has a loyalty timestamp
require(ISparkleTimestamp(timestampAddress).hasTimestamp(msg.sender), 'No timestamp2');

// Determine if the account has been locked
if(accounts[msg.sender]._isLocked) {
// Yes, revert informing that this loyalty account has been locked
revert('Locked');
}

// Obtain values needed from account record before zeroing
uint256 joinCount = accounts[msg.sender]._joined;
uint256 collected = accounts[msg.sender]._collected;
uint256 deposit = accounts[msg.sender]._balance;
bool isLocked = accounts[msg.sender]._isLocked;
// Zero out the callers account record
Account storage account = accounts[msg.sender];
```

```
account._address = address(0x0);
account._balance = 0x0;
account._collected = 0x0;
account._joined = joinCount;
account._claimed = 0x0;
account._tier = 0x0;
// Preserve account lock even after withdraw (account always locked)
account._isLocked = isLocked;
// Decement the total number of active accounts
totalActiveAccounts = totalActiveAccounts.sub(1);

// Delete the callers timestamp record
_deleteTimestamp(msg.sender);

// Determine if transfer from treasury address is a success
if(!IERC20(tokenAddress).transferFrom(treasuryAddress, msg.sender, collected)) {
// No, revert indicating that the transfer and wisthdraw has failed
revert('Withdraw failed');
}

// Determine if transfer from contract address is a sucess
if(!IERC20(tokenAddress).transfer(msg.sender, deposit)) {
// No, revert indicating that the treansfer and withdraw has failed
revert('Withdraw failed');
}

// Emit event log to the block chain for future web3 use
emit LoyaltyWithdrawnEvent(msg.sender, deposit.add(collected));
}

function returnLoyaltyDeposit(address _rewardAddress)
public
whenNotPaused
onlyOwner
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
// validate that caller has a loyalty timestamp
require(ISparkleTimestamp(timestampAddress).hasTimestamp(_rewardAddress), 'No timestamp2');
// Validate that reward address is locked
require(accounts[_rewardAddress]._isLocked, 'Lock account first');
uint256 deposit = accounts[_rewardAddress]._balance;
Account storage account = accounts[_rewardAddress];
account._balance = 0x0;
// Determine if transfer from contract address is a sucess
if(!IERC20(tokenAddress).transfer(_rewardAddress, deposit)) {
// No, revert indicating that the treansfer and withdraw has failed
revert('Withdraw failed');
}

// Emit event log to the block chain for future web3 use
emit LoyaltyDepositWithdrawnEvent(_rewardAddress, deposit);
}

function returnLoyaltyCollected(address _rewardAddress)
public
whenNotPaused
onlyOwner
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
```

```
// validate that caller has a loyalty timestamp
require(ISparkleTimestamp(timestampAddress).hasTimestamp(_rewardAddress), 'No timestamp2b');
// Validate that reward address is locked
require(accounts[_rewardAddress]._isLocked, 'Lock account first');
uint256 collected = accounts[_rewardAddress]._collected;
Account storage account = accounts[_rewardAddress];
account._collected = 0x0;
// Determine if transfer from treasury address is a success
if(!IERC20(tokenAddress).transferFrom(treasuryAddress, _rewardAddress, collected)) {
// No, revert indicating that the transfer and wisthdraw has failed
revert('Withdraw failed');
}

// Emit event log to the block chain for future web3 use
emit LoyaltyCollectedWithdrawnEvent(_rewardAddress, collected);
}

function removeLoyaltyAccount(address _rewardAddress)
public
whenNotPaused
onlyOwner
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
// validate that caller has a loyalty timestamp
require(ISparkleTimestamp(timestampAddress).hasTimestamp(_rewardAddress), 'No timestamp2b');
// Validate that reward address is locked
require(accounts[_rewardAddress]._isLocked, 'Lock account first');
uint256 joinCount = accounts[_rewardAddress]._joined;
Account storage account = accounts[_rewardAddress];
account._address = address(0x0);
account._balance = 0x0;
account._collected = 0x0;
account._joined = joinCount;
account._claimed = 0x0;
account._tier = 0x0;
account._isLocked = false;
// Decement the total number of active accounts
totalActiveAccounts = totalActiveAccounts.sub(1);

// Delete the callers timestamp record
_deleteTimestamp(_rewardAddress);

emit LoyaltyAccountRemovedEvent(_rewardAddress);
}

/**
* @dev Gets the locked status of the specified address
* @param _loyaltyAddress of account
* @return (bool) indicating locked status
*/
function isLocked(address _loyaltyAddress)
public
view
whenNotPaused
returns (bool)
{
return accounts[_loyaltyAddress]._isLocked;
}

function lockAccount(address _rewardAddress, bool _value)
public
```

```
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {from}');
require(_rewardAddress != address(0x0), 'Invalid {reward}');
// Validate specified address has timestamp
require(ISparkleTimestamp(timestampAddress).hasTimestamp(_rewardAddress), 'No timstamp');
// Set the specified address' locked status
accounts[_rewardAddress]._isLocked = _value;
// Emit event log to the block chain for future web3 use
emit LockedAccountEvent(_rewardAddress, _value);
}

/**
* @dev Gets the storage address value of the specified address
* @param _loyaltyAddress of account
* @return (address) indicating the address stored calls account record
*/
function getLoyaltyAddress(address _loyaltyAddress)
public
view
whenNotPaused
returns(address)
{
return accounts[_loyaltyAddress]._address;
}

/**
* @dev Get the deposit balance value of specified address
* @param _loyaltyAddress of account
* @return (uint256) indicating the balance value
*/
function getDepositBalance(address _loyaltyAddress)
public
view
whenNotPaused
returns(uint256)
{
return accounts[_loyaltyAddress]._balance;
}

/**
* @dev Get the tokens collected by the specified address
* @param _loyaltyAddress of account
* @return (uint256) indicating the tokens collected
*/
function getTokensCollected(address _loyaltyAddress)
public
view
whenNotPaused
returns(uint256)
{
return accounts[_loyaltyAddress]._collected;
}

/**
* @dev Get the total balance (deposit + collected) of tokens
* @param _loyaltyAddress of account
* @return (uint256) indicating total balance
*/
function getTotalBalance(address _loyaltyAddress)
```

```
public
view
whenNotPaused
returns(uint256)
{
return accounts[_loyaltyAddress]._balance.add(accounts[_loyaltyAddress]._collected);
}

/**
* @dev Get the times loyalty has been claimed
* @param _loyaltyAddress of account
* @return (uint256) indicating total time claimed
*/
function getTimesClaimed(address _loyaltyAddress)
public
view
whenNotPaused
returns(uint256)
{
return accounts[_loyaltyAddress]._claimed;
}

/**
* @dev Get total number of times joined
* @param _loyaltyAddress of account
* @return (uint256)
*/
function getTimesJoined(address _loyaltyAddress)
public
view
whenNotPaused
returns(uint256)
{
return accounts[_loyaltyAddress]._joined;
}

/**
* @dev Get time remaining before reward maturity
* @param _loyaltyAddress of account
* @return (uint256, bool) Indicating time remaining/past and boolean indicating maturity
*/
function getTimeRemaining(address _loyaltyAddress)
public
whenNotPaused
returns (uint256, bool, uint256)
{
(uint256 remaining, bool status, uint256 deposit) = ISparkleTimestamp(timestampAddress).getTimeRemaining(_loyaltyAddress);
return (remaining, status, deposit);
}

/**
* @dev Withdraw any ether that has been sent directly to the contract
* @param _loyaltyAddress of account
* @return Total number of tokens that have been claimed by users
* @notice Test(s) Not written
*/
function getRewardTier(address _loyaltyAddress)
public
view whenNotPaused
returns(uint256)
{
return accounts[_loyaltyAddress]._tier;
}
```

```
/**
* @dev Select reward tier for msg.sender
* @param _tierSelected id of the reward tier interested in purchasing
* @return (bool) indicating failure/success
*/
function selectRewardTier(uint256 _tierSelected)
public
payable
whenNotPaused
nonReentrant
returns(bool)
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {From}');
// Validate specified address has a timestamp
require(accounts[msg.sender]._address == address(msg.sender), 'No timestamp3');
// Validate tier selection
require(accounts[msg.sender]._tier != _tierSelected, 'Already selected');
// Validate that ether was sent with the call
require(msg.value > 0, 'No ether');

// Determine if the specified rate is > than existing rate
if(ISparkleRewardTiers(tiersAddress).getRate(accounts[msg.sender]._tier) >= ISparkleRewardTiers(tiersAddress).getRate(_tierSelected)) {
// No, revert indicating failure
revert('Invalid tier');
}

// Determine if ether transfer for tier upgrade has completed successfully
(bool success, ) = address(collectionAddress).call{value: ISparkleRewardTiers(tiersAddress).getPrice(_tierSelected), gas: gasToSendWithTX}('');
require(success, 'Rate unchanged');

// Update callers rate with the new selected rate
accounts[msg.sender]._tier = _tierSelected;
emit TierSelectedEvent(msg.sender, _tierSelected);
// Return success
return true;
}

function getRewardTiersAddress()
public
view
whenNotPaused
returns(address)
{
return tiersAddress;
}

/**
* @dev Set tier collectionm address
* @param _newAddress of new collection address
* @notice Test(s) not written
*/
function setRewardTiersAddress(address _newAddress)
public
whenNotPaused
onlyOwner
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {From}');
// Validate specified address is valid
require(_newAddress != address(0), 'Invalid {reward}');
```

```
// Set tier rewards contract address
tiersAddress = _newAddress;
emit TiersAddressChanged(_newAddress);
}

function getCollectionAddress()
public
view
whenNotPaused
returns(address)
{
return collectionAddress;
}

/** @notice Test(s) passed
* @dev Set tier collectionm address
* @param _newAddress of new collection address
*/
function setCollectionAddress(address _newAddress)
public
whenNotPaused
onlyOwner
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {From}');
// Validate specified address is valid
require(_newAddress != address(0), 'Invalid {collection}');
// Set tier collection address
collectionAddress = _newAddress;
emit CollectionAddressChanged(_newAddress);
}

function getTreasuryAddress()
public
view
whenNotPaused
returns(address)
{
return treasuryAddress;
}

/**
* @dev Set treasury address
* @param _newAddress of the treasury address
* @notice Test(s) passed
*/
function setTreasuryAddress(address _newAddress)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), "Invalid {from}");
// Validate specified address
require(_newAddress != address(0), "Invalid {treasury}");
// Set current treasury contract address
treasuryAddress = _newAddress;
emit TreasuryAddressChanged(_newAddress);
}

function getTimestampAddress()
```

```
public
view
whenNotPaused
returns(address)
{
return timestampAddress;
}

/**
* @dev Set the timestamp address
* @param _newAddress of timestamp address
* @notice Test(s) passed
*/
function setTimestampAddress(address _newAddress)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), "Invalid {from}");
// Set current timestamp contract address
timestampAddress = _newAddress;
emit TimestampAddressChanged(_newAddress);
}

function getTokenAddress()
public
view
whenNotPaused
returns(address)
{
return tokenAddress;
}

/**
* @dev Set the loyalty token address
* @param _newAddress of the new token address
* @notice Test(s) passed
*/
function setTokenAddress(address _newAddress)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), "Invalid {from}");
// Set current token contract address
tokenAddress = _newAddress;
emit TokenAddressChangedEvent(_newAddress);
}

function getSentGasAmount()
public
view
whenNotPaused
returns(uint256)
{
return gasToSendWithTX;
}

function setSentGasAmount(uint256 _amount)
```

```
public
onlyOwner
whenNotPaused
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
// Set the current minimum deposit allowed
gasToSendWithTX = _amount;
emit GasSentChanged(_amount);
}

/**
* @dev Set the minimum Proof Of Loyalty amount allowed for deposit
* @param _minProof amount for new minimum accepted loyalty reward deposit
* @notice _minProof value is multiplied internally by 10e7. Do not multiply before calling!
*/
function setMinProof(uint256 _minProof)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
// Validate specified minimum is not lower than 1000 tokens
require(_minProof >= 1000, 'Invalid amount');
// Set the current minimum deposit allowed
minRequired = _minProof.mul(10e7);
emit MinProofChanged(minRequired);
}

event MinProofChanged(uint256);
/**
* @dev Get the minimum Proof Of Loyalty amount allowed for deposit
* @return Amount of tokens required for Proof Of Loyalty Rewards
* @notice Test(s) passed
*/
function getMinProof()
public
view
whenNotPaused
returns(uint256)
{
// Return indicating minimum deposit allowed
return minRequired;
}

/**
* @dev Set the maximum Proof Of Loyalty amount allowed for deposit
* @param _maxProof amount for new maximum loyalty reward deposit
* @notice _maxProof value is multiplied internally by 10e7. Do not multiply before calling!
* @notice Smallest maximum value is 1000 + _minProof amount. (Ex: If _minProof == 1000 then smallest _maxProof possible is 2000)
*/
function setMaxProof(uint256 _maxProof)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0), 'Invalid {from}');
require(_maxProof >= 2000, 'Invalid amount');
// Set allow maximum deposit
```

```
maxAllowed = _maxProof.mul(10e7);
}

/**
* @dev Get the maximum Proof Of Loyalty amount allowed for deposit
* @return Maximum amount of tokens allowed for Proof Of Loyalty deposit
* @notice Test(s) passed
*/
function getMaxProof()
public
view
whenNotPaused
returns(uint256)
{
// Return indicating current allowed maximum deposit
return maxAllowed;
}

/**
* @dev Get the total number of tokens claimed by all users
* @return Total number of tokens that have been claimed by users
* @notice Test(s) Not written
*/
function getTotalTokensClaimed()
public
view
whenNotPaused
returns(uint256)
{
// Return indicating total number of tokens that have been claimed by all
return totalTokensClaimed;
}

/**
* @dev Get total number of times rewards have been claimed for all users
* @return Total number of times rewards have been claimed
*/
function getTotalTimesClaimed()
public
view
whenNotPaused
returns(uint256)
{
// Return indicating total number of tokens that have been claimed by all
return totalTimesClaimed;
}

/**
* @dev Withdraw any ether that has been sent directly to the contract
*/
function withdrawEth(address _toAddress)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {from}');
// Validate specified address
require(_toAddress != address(0x0), 'Invalid {to}');
// Validate there is ether to withdraw
require(address(this).balance > 0, 'No ether');
// Determine if ether transfer of stored ether has completed successfully
```

```
// require(address(_toAddress).call.value(address(this).balance).gas(gasToSendWithTX)(), 'Withdraw failed');
(bool success, ) = address(_toAddress).call{value:address(this).balance, gas: gasToSendWithTX}('');
require(success, 'Withdraw failed');
}

/**
* @dev Withdraw any ether that has been sent directly to the contract
* @param _toAddress to receive any stored token balance
*/
function withdrawTokens(address _toAddress)
public
onlyOwner
whenNotPaused
nonReentrant
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {from}');
// Validate specified address
require(_toAddress != address(0), "Invalid {to}");
// Validate there are tokens to withdraw
uint256 balance = IERC20(tokenAddress).balanceOf(address(this));
require(balance != 0, "No tokens");

// Validate the transfer of tokens completed successfully
if(IERC20(tokenAddress).transfer(_toAddress, balance)) {
emit TokensWithdrawn(_toAddress, balance);
}
}

/**
* @dev Override loyalty account tier by contract owner
* @param _loyaltyAccount loyalty account address to tier override
* @param _tierSelected reward tier to override current tier value
* @return (bool) indicating success status
*/
function overrideRewardTier(address _loyaltyAccount, uint256 _tierSelected)
public
whenNotPaused
onlyOwner
nonReentrant
returns(bool)
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {from}');
require(_loyaltyAccount != address(0x0), 'Invalid {account}');
// Validate specified address has a timestamp
require(accounts[_loyaltyAccount]._address == address(_loyaltyAccount), 'No timestamp4');
// Update the specified loyalty address tier reward index
accounts[_loyaltyAccount]._tier = _tierSelected;
emit RewardTierChanged(_loyaltyAccount, _tierSelected);
}

/**
* @dev Reset the specified loyalty account timestamp
* @param _rewardAddress of the loyalty account to performm a reset
*/
function _resetTimestamp(address _rewardAddress)
internal
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {from}');
// Validate specified address
require(_rewardAddress != address(0), "Invalid {reward}");
```

```
// Reset callers timestamp for specified address
require(ISparkleTimestamp(timestampAddress).resetTimestamp(_rewardAddress), 'Reset failed');
emit ResetTimestampEvent(_rewardAddress);
}

/**
* @dev Delete the specified loyalty account timestamp
* @param _rewardAddress of the loyalty account to performm the delete
*/
function _deleteTimestamp(address _rewardAddress)
internal
{
// Validate calling address (msg.sender)
require(msg.sender != address(0x0), 'Invalid {from}16');
// Validate specified address
require(_rewardAddress != address(0), "Invalid {reward}");
// Delete callers timestamp for specified address
require(ISparkleTimestamp(timestampAddress).deleteTimestamp(_rewardAddress), 'Delete failed');
emit DeleteTimestampEvent(_rewardAddress);
}

/**
* @dev Event signal: Treasury address updated
*/
event TreasuryAddressChanged(address);

/**
* @dev Event signal: Timestamp address updated
*/
event TimestampAddressChanged(address);

/**
* @dev Event signal: Token address updated
*/
event TokenAddressChangedEvent(address);

/**
* @dev Event signal: Timestamp reset
*/
event ResetTimestampEvent(address _rewardAddress);

/**
* @dev Event signal: Timestamp deleted
*/
event DeleteTimestampEvent(address _rewardAddress);

/**
* @dev Event signal: Loyalty deposited event
*/
event DepositLoyaltyEvent(address, uint256, bool);

/**
* @dev Event signal: Reward claimed successfully for address
*/
event RewardClaimedEvent(address, uint256);

/**
* @dev Event signal: Loyalty withdrawn
*/
event LoyaltyWithdrawnEvent(address, uint256);

/**
* @dev Event signal: Account locked/unlocked
```

```
*/
event LockedAccountEvent(address _rewardAddress, bool _locked);

/**
* @dev Event signal: Loyalty deposit balance withdrawn
*/
event LoyaltyDepositWithdrawnEvent(address, uint256);

/**
* @dev Event signal: Loyalty collected balance withdrawn
*/
event LoyaltyCollectedWithdrawnEvent(address, uint256);

/**
* @dev Event signal: Loyalty account removed
*/
event LoyaltyAccountRemovedEvent(address);

/**
* @dev Event signal: Gas sent with call.value amount updated
*/
event GasSentChanged(uint256);
/**
* @dev Event signal: Reward tiers address updated
*/
event TierSelectedEvent(address, uint256);

/**
* @dev Event signal: Reward tiers address updated
*/
event TiersAddressChanged(address);

/**
* @dev Event signal: Reward tier has been updated
*/
event RewardTierChanged(address, uint256);

/**
* @dev Event signal: Collection address updated
*/
event CollectionAddressChanged(address);

/**
* @dev Event signal: All stored tokens have been removed
*/
event TokensWithdrawn(address, uint256);
}
```

Analysis b6a808f2-eb5b-401d-934b-8d1d5287fdbd MythX

Started	Sun Aug 23 2020 22:43:18 GMT+0000 (Coordinated Universal Time)
Finished	Sun Aug 23 2020 22:58:29 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Mythx-Cli-0.6.19
Main Source File	Contracts/SparkleRewardTiers.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	0

ISSUES