Cyberscope

Audit Report

8lends

November 2024

Repository https://github.com/8lnds/tmp_repo

Commit 2172fb6e273347e93ae0ee7552fc3e03d2c4db6b

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/8lnds/tmp_repo
Commit	2172fb6e273347e93ae0ee7552fc3e03d2c4db6b

Test Deploys	https://sepolia.etherscan.io/address/0xd6AA2436b41429e308efcc1A6109846AA4198219 https://sepolia.etherscan.io/address/0x3a9e5cb432897542CE724fd01387373184a17FB5 https://sepolia.etherscan.io/address/0x0D3751F7Bd2f86622B3313D12ad67A4B1BE8D80e https://sepolia.etherscan.io/address/0x116445b0Acc4df18a5A81b39099fb889eA83D0c7 https://sepolia.etherscan.io/address/0x0C2F5AfaDAfBA5D4C84985E33b785f5C7DD34f17 https://sepolia.etherscan.io/address/0x0dB0215D01eC84984E081eB45Cd56218a043fB9E https://sepolia.etherscan.io/address/0xbA843654bFB975E8e8CE6C34ce9d2eF8cc47c44F https://sepolia.etherscan.io/address/0xC95B622D242BDE00e29834E254777C80ac5dEAFE

Audit Updates

Initial Audit	15 Nov 2024

Source Files

Filename	SHA256
TreasuryETH.sol	9fbc50928aafea146bfbc0ceeed57b21c22f9b74c06e7a9a6945d5eca3e77c8a
Sell.sol	34a079d99015895d0597f8f3795e8daeb43b6a1f79173a93a9925f2bb354964e
MockERC20.sol	dfed5cd3ce5a1199acb44ca66a2e0e753eee359592f294b58a715ce994aa15c9
ManagerRegistry.sol	1e46c00b62cab50e8583d7e4b235f449a82bad8eb9b3fa7a9e92c455475af83c
MCLR.sol	c0c00daec89de4e902603c424e1b1568d51f5e3d073a38fb657ee7fea4aab3a4
Fundraise.sol	c5b3c468a87c5e8aa9973672286e04cde7ce316136ffd2e4ba92746cc9b8f4d2
Burn.sol	00c94a282498a9fc4f275195484dfc6dc3e2138a7b03a4457bf9f6c84b3834ec
lib/MerkleProof.sol	74f97c6253565a5e5a697489cd67f6ba415cda0a2a4cfdee04dfb2f248aca28e
interfaces/ITreasury.sol	67344a7f0a0366f3c867ef275f6c700e4bb9f830532a860ba427c02b41a48bda
interfaces/IManagerRegistry.sol	b97b0a96a6c66ba62e6961c8c857d909d4ea147c4ee62c874eb6a567232c40ab
interfaces/IFundraise.sol	e2a054f9c0f2587cfc148f32307551e14bd5c9486932e63355080935622e2366

interfaces/IERC20Mintable.sol	e9a6d0bcc7a87eb434ffe1a54a4d07c81e1f00a9b8a76655ead801ada08b689c

Overview

8lends is an innovative project that has undergone an extensive audit of its core smart contracts. 8lends enables the creation of projects that can be funded by accredited investors. The scope of the audit covers the following smart contracts and functionalities.

Fundraise.sol

This contract manages the state of projects that can be funded by whitelisted investors using a loan token. Projects are initiated by an administrator with parameters such as hard cap, soft cap, total invested amount, start time, duration of pre-funding, interest rate for investors, end of the open stage, loyalty percentage, platform interest rate, price for the MCLR token, and the borrower's address, among others.

Once a project accumulates enough tokens to meet the soft cap, it is marked as pre-funded, and further investments are halted. The borrower can withdraw the invested tokens when the project is pre-funded or when the soft cap is reached. At this point, the project is marked as funded, and an equivalent amount of MCLR tokens is minted to the treasury contract.

For a funded project, the borrower or administrator can repay the loaned tokens to the contract. The project is marked as repaid only if the total invested amount plus the interest rate is fully repaid. For repaid or partially repaid projects, investors can claim a portion of the tokens proportional to their initial investment. The claimable amount can be withdrawn either in the form of the loaned token or as MCLR tokens from the treasury contract. In the latter case, the treasury receives the loaned tokens in exchange.

ManagerRegistry.sol

This contract defines a list of manager addresses with elevated privileges, allowing them to act as administrators on all other contracts. The isAllowedCall function is used to verify if an address is a manager.

Sell.sol

The contract implements a sell function that accepts a USDT amount from the caller and mints MCLR tokens in exchange. The exchange rate is determined by the MCLR price set in

the latest project. The minted MCLR tokens are owned by the caller's address. It also includes a claim function that allows the manager to withdraw tokens from the contract.

TreasuryETH.sol

This contract is used to store MCLR tokens for funded projects. It includes a claim function that allows the manager to withdraw tokens from the contract.

MCLR.sol

An upgradable, mintable, and burnable ERC20 token named MCLR. The minter role is assigned to a specific address during initialization, allowing it to mint new tokens. Additionally, the upgrader role is capable of upgrading the contract.

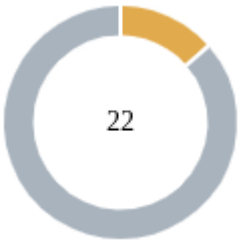
MockERC20.sol

An upgradable, mintable, burnable and ownable ERC20 token. The minter role is assigned to a specific address during initialization, allowing it to mint new tokens. Additionally, the owner is capable of upgrading the contract.

Burn.sol

The contract implements a burn function that allows a manager to burn an amount of MCLR tokens from its balance.

Findings Breakdown



Critical	0
Medium	3
Minor / Informative	19

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	3	0	0	0
Minor / Informative	19	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	DPI	Decimals Precision Inconsistency	Unresolved
Medium	MAC	Missing Access Control	Unresolved
Medium	MCSV	Missing Cross-Chain Signature Verification	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	CFI	Claimed Fee Inconsistency	Unresolved
Minor / Informative	DCE	Dead Code Elimination	Unresolved
Minor / Informative	EFT	Early Funding Termination	Unresolved
Minor / Informative	ERR	Exchange Ratio Risk	Unresolved
Minor / Informative	MPC	Merkle Proof Centralization	Unresolved
Minor / Informative	MT	Mints Tokens	Unresolved
Minor / Informative	MEM	Missing Error Messages	Unresolved
Minor / Informative	MIV	Missing Input Validation	Unresolved
Minor / Informative	MRM	Missing Redemption Mechanism	Unresolved
Minor / Informative	PSRV	Potential Signature Replay Vulnerability	Unresolved

●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	ULP	Uncollateralized Loan Provision	Unresolved
●	UTI	Unimplemented Time-Based Interest	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved

DPI - Decimals Precision Inconsistency

Criticality	Medium
Location	Sell.sol#L61 Fundraise.sol#L236
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in how the decimals field is handled across contracts. Specifically, the `Fundraise` contract mints tokens proportionally to the invested amounts of a `loanToken`, while the `Sell` contract exchanges a `USDT` token using the same price. Since these two tokens may not have the same decimal precision, users might receive significantly fewer minted tokens than expected for their actual deposited value. As a result, this issue could lead to a significant loss of funds for users.

```
function sell(uint256 _usdtAmount) external {
require(!isPaused, "E1");
IERC20(usdt).safeTransferFrom(msg.sender, address(this),
_usdtAmount);
uint256 price = IFundraise(fundraise).lastMCLRPrice();
uint256 mclrAmount = (_usdtAmount / (price + (price / 1000)) *
addedPercentage);
IERC20Mintable(MCLR).mint(msg.sender, mclrAmount);
}
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

MAC - Missing Access Control

Criticality	Medium
Location	ManagerRegistry.sol#L26 TreasuryETH.sol#L92 MCLR.sol#L21 MockERC20.sol#L23 Sell.sol#L37 Burn.sol#L24
Status	Unresolved

Description

The `initialize` functions can be frontrun during deployment, allowing administrative roles to be transferred to third parties not associated with the team. Such third parties would gain access to all the functions of the system.

```
function initialize(address _superManager)
        initializer public
    {
        __AccessControl_init();
        __UUPSUpgradeable_init();

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(SUPER_MANAGER_ROLE, _superManager);

    }
```

Recommendation

The team is advised to implement proper access controls to ensure that only authorized team members can call these functions.

MCSV - Missing Cross-Chain Signature Verification

Criticality	Medium
Location	Fundraise.sol#L130
Status	Unresolved

Description

The contract implements a signature verification mechanism. The signature includes the caller of the function, the project ID, the amount to invest, the root of the current Merkle tree, and a nonce. However, the signature lacks verification against the current `chainId`. Without including the `chainId` in the verification process, the same message could potentially be replicated across different chains in a cross-chain application.

```
bytes32 ethSignedMessageHash = keccak256(abi.encodePacked("\x19Ethereum
Signed Message:\n32", keccak256(abi.encodePacked(msg.sender, _pid, _amount,
_rootHash, _nonce))));
(bytes32 r, bytes32 s, uint8 v) = splitSignature(_sig);
address signer = ecrecover(ethSignedMessageHash, v, r, s);
```

Recommendation

The team is advised to include the current `chainId` as part of the verification process. This will prevent the same signature from being used across different networks.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Fundraise.sol#L185,203,222,247,285,355,373,386,407,437,445,452 TreasuryETH.sol#L44 Sell.sol#L70,77,84 ManagerRegistry.sol#L46 Burn.sol#L41 MCLR.sol#L34,38 MockERC20.sol#L35,39
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function withdrawInvestment(uint256 _projectId, address _investor)
external {}
function cancelProject(uint256 _projectId) external {}
function transferFundsToBorrower(uint256 _projectId) external {}
function makeRepayment(uint256 _projectId, uint256 _amount)
external {}
function claim(uint256 _projectId, bool _claimInMCLR, address
_investor) external {}
function createProject(
        Project memory _project,
        bytes32 _whitelistRoot,
        uint256 _projectHash
    ) external returns (uint256) {}
function setMCLRPrice(uint256 _projectId, uint256 _newMCLRPrice)
external  {}
function moveProjectStage(uint256 _projectId) external  {}
function setProject(uint256 _projectId, Project memory _project)
external  {}
function setWhitelist(bytes32 _whitelistRoot, uint256 _projectId)
external {}
function setLoyalty(bytes32 _loyaltyRoot, uint256 _projectId)
external {}
function setTrustedSigner(address _signer) external {}
function claim(address _token, uint256 _amount, address _recepient)
external {}
function setPercentage(uint256 _addedPercentage) external {}
function setPauseStatus(bool _status) external {}
function setManagerStatus(address _manager, bool _status) external
onlyRole(SUPER_MANAGER_ROLE) {}
function burn(uint256 amount) external  {}
function mint(address to, uint256 amount) external
onlyRole(MINTER_ROLE) {}
function _authorizeUpgrade(address newImplementation)
        internal
        onlyRole(UPGRADER_ROLE)
        override
    {}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

CFI - Claimed Fee Inconsistency

Criticality	Minor / Informative
Location	Fundraise.sol#L329
Status	Unresolved

Description

The contract implements the `_claim` function, which allows project investors to claim their share of the repaid amounts based on their investment allocation. Once the claimable amount is determined from the repaid funds, a fee is deducted from the withdrawal. However, the fee is calculated based on the total invested amount rather than the total repaid amount. This fee is then subtracted from the claimable amount, but the contract does not account for cases where the claimable amount is less than the fee. This oversight can lead to a potential underflow and transaction failure during execution.

```
claimable -= fee;
```

Recommendation

The team is advised to implement proper checks to ensure the fee is applied only when it can be deducted from the claimable amount.

DCE - Dead Code Elimination

Criticality	Minor / Informative
Location	Fundraise.sol#L143
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

In this case, the following conditional statement is never executed

```
if (project.totalInvested >= project.hardCap) {
project.openStageEndAt = block.timestamp;
project.innerStruct.stage = Stage.PreFunded;
emit ProjectStatusChanged(_pid, uint8(Stage.PreFunded));
}
```

due to the earlier statement:

```
require(project.totalInvested + _amount <= project.hardCap, "E6");
```

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

EFT - Early Funding Termination

Criticality	Minor / Informative
Location	Fundraise.sol#L173
Status	Unresolved

Description

The contract implements the `_invest` function, which allows whitelisted investors to deposit an amount of `loanToken` for a project. If the current timestamp exceeds the termination timestamp of the open stage and the `totalInvested` amount exceeds the `softCap`, the project is marked as `PreFunded`. At this point, investors cannot contribute additional funds, even if the `hardCap` has not been reached.

```
function _invest(uint256 _pid, uint256 _amount) internal {
Project storage project = projects[_pid];
if (project.innerStruct.stage == Stage.ComingSoon) {
if (block.timestamp >= project.startAt) {
    project.innerStruct.stage = Stage.Open;
    emit ProjectStatusChanged(_pid, uint8(Stage.Open));
}   else {
    return;
    }
}
require(project.innerStruct.stage == Stage.Open, "E5");
require(project.totalInvested + _amount <= project.hardCap, "E6");
if (block.timestamp > project.openStageEndAt) {
if (project.totalInvested > project.softCap) {
    project.innerStruct.stage = Stage.PreFunded;
    project.openStageEndAt = block.timestamp;
    emit ProjectStatusChanged(_pid, uint8(Stage.PreFunded));
    return;
} else {
    project.innerStruct.stage = Stage.Canceled;
    emit ProjectStatusChanged(_pid, uint8(Stage.Canceled));
    return;
    }
}
```

Recommendation

The team is advised to revise the implementation of the `_invest` function to ensure that accredited investors are allowed to invest in projects during the phase between the `softCap` and `hardCap`.

ERR - Exchange Ratio Risk

Criticality	Minor / Informative
Location	Sell.sol#L57 Fundraise.sol#L333
Status	Unresolved

Description

The `Sell` contract allows `USDT` tokens to be exchanged for `MCLR` tokens at a price derived from the most recently created project. Since this price information is publicly available on the blockchain, users can potentially time their exchanges to mint new tokens at favorable rates, thereby affecting the overall price stability of `MCLR`.

Similarly, the Fundraise contract uses the same price to estimate claimable funds in the form of MCLR tokens. If this price does not accurately reflect market conditions, it could lead to price instability for the token.

```
function sell(uint256 _usdtAmount) external {
require(!isPaused, "E1");
IERC20(usdt).safeTransferFrom(msg.sender, address(this), _usdtAmount);
uint256 price = IFundraise(fundraise).lastMCLRPrice();
uint256 mclrAmount = (_usdtAmount / (price + (price / 1000)) *
addedPercentage);
IERC20Mintable(MCLR).mint(msg.sender, mclrAmount);
}


uint256 mclrAmount = (claimable * BASIS_POINTS) /
project.innerStruct.mclrPrice;
```

Recommendation

Relying on the last available price to mint new tokens can pose risks to the token's price stability. Implementing a decentralized approach that processes real-time information would provide a more robust and secure solution.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	Fundraise.sol#L119,277
Status	Unresolved

Description

The contract uses a Merkle Proof mechanism in order to define many applicable addresses. The verification process is based on an off-chain configuration. The contract owner is responsible for updating the in-chain “Merkle Root” in order to validate correctly the provided message.

```
require(MerkleProof.verifyCalldata(_proof, whitelistRoots[_pid],
leaf), "E1");
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly. The team should carefully manage the private keys of the owner’s account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

MT - Mints Tokens

Criticality	Minor / Informative
Location	MCLR.sol#L34 MockERC20.sol#L35 Fundraise.sol#L237 Sell.sol#L63
Status	Unresolved

Description

The `MINTER_ROLE` has the authority to mint tokens. The minter may take advantage of it by calling the `mint` function. As a result, the contract tokens may be highly inflated.

```
function mint(address to, uint256 amount) external
onlyRole(MINTER_ROLE) {
        _mint(to, amount);
}
```

In addition, the `Fundraise` and `Sell` contracts have the ability to mint tokens through calls of external functions.

```
IERC20Mintable(address(mclrToken)).mint(treasury, mclrAmount);

IERC20Mintable(MCLR).mint(msg.sender, mclrAmount);
```

Recommendation

The team should carefully manage the private keys of the minter's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the `MINTER_ROLE`, which will eliminate the threats but it is non-reversible.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	Fundraise.sol#L119,130,137,153,154,187,190,191,193,205,213,225,227,230,249,252,277,291,294,295,298,360,374,375,376,379,387,408,438,446,453,514
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
if (project.totalInvested < project.softCap) revert("E11");
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MIV - Missing Input Validation

Criticality	Minor / Informative
Location	Fundraise.sol#L323
Status	Unresolved

Description

The contract implements the `createProject` function, which initializes a new project based on user-provided arguments. However, the contract does not verify that the input arguments are sanitized and properly structured. Specifically, the borrower address is not checked against the zero address, the project duration is not validated against the provided start and end timestamps, the interest rates are not checked against a maximum allowable value, the `loyalityPercent` is not checked against the `platformInterestRate` and the price is not verified to have the expected value. These discrepancies can lead to significant execution problems, resulting in issues such as halted transactions due to underflows and division by zero errors.

```
function createProject(
Project memory _project,
bytes32 _whitelistRoot,
uint256 _projectHash) external returns (uint256) {

require(IManagerRegistry(managerRegistry).isAllowedCall(msg.sender)
, "E9");
uint256 projectId = projectCount++;
projects[projectId] = _project;
whitelistRoots[projectId] = _whitelistRoot;
emit ProjectCreated(projectId, _project.innerStruct.borrower,
_projectHash);
return projectId;
}
```

Recommendation

The team is advised to ensure that all user-provided variables are properly sanitized to conform to the expected format. Incorporating these steps at the creation stage ensures the contract performs as intended and enhances user trust.

MRM - Missing Redemption Mechanism

Criticality	Minor / Informative
Location	MCLR.sol
Status	Unresolved

Description

The contract processes the borrowing of loaned amounts by minting MCLR tokens to the treasury address. It also burns tokens when investors claim the loan tokens, thereby removing the loaned tokens from the system. However, the contract lacks a mechanism that allows users to exchange MCLR tokens for the deposited amounts. Implementing such a functionality could be beneficial, as it would peg the minted tokens to the withheld reserves.

Recommendation

The team is advised to consider implementing a mechanism that allows users to exchange minted tokens for stored reserves. To ensure proper functionality, it is recommended to mint a separate instance of the MCLR token for each project.

PSRV - Potential Signature Replay Vulnerability

Criticality	Minor / Informative
Location	Fundraise.sol#L129
Status	Unresolved

Description

The contract implements the `investUpdate` function to update the Merkle root for investors of a specific project ID. As part of the verification process, the function confirms the validity of a signature against a `trustedSigner`. To prevent signature replay attacks, the contract uses a nonce in the signing process. Specifically, before signature verification, the contract checks if the received nonce is higher than the current nonce, and at the end of the verification process, it increments the nonce by 1. However, this setup allows for a signature with a nonce value greater than `nonce + 1` to be accepted. In such cases, the signature can be replayed multiple times until the contract state reaches the provided nonce. This vulnerability allows third parties to access the function if a nonce value greater than `nonce + 1` is used.

```
function investUpdate(uint256 _pid, uint256 _amount, bytes32 _rootHash,
uint256 _nonce, bytes memory _sig) external {
require(_nonce > nonce, "E3");
bytes32 ethSignedMessageHash = keccak256(abi.encodePacked("\x19Ethereum
Signed Message:\n32", keccak256(abi.encodePacked(msg.sender, _pid, _amount,
_rootHash, _nonce))));
(bytes32 r, bytes32 s, uint8 v) = splitSignature(_sig);
address signer = ecrecover(ethSignedMessageHash, v, r, s);
require(signer == trustedSigner, "E4");
whitelistRoots[_pid] = _rootHash;
_invest(_pid, _amount);
nonce++;
}
```

Recommendation

Verifying that the provided nonce value is exactly the expected value will prevent signatures from being replayed. The team is advised to implement this check to ensure that each signature is used only once and in the correct sequence.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	TreasuryETH.sol#L169,255
Status	Unresolved

Description

The contract operates on transfered amounts and updates its state without accounting for potential fees. The fees or taxes of a token are an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
project.innerStruct.loanToken.safeTransferFrom(msg.sender,
address(this), _amount);
project.totalInvested += _amount;
```

```
project.innerStruct.loanToken.safeTransferFrom(msg.sender,
address(this), _amount);
project.innerStruct.totalRepaid += _amount;
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

ULP - Uncollateralized Loan Provision

Criticality	Minor / Informative
Location	Fundraise.sol#L
Status	Unresolved

Description

The `Fundraise` contract is designed to distribute undercollateralized loans to project borrowers. The current implementation lacks safeguards to ensure that tokens are properly repaid. If a borrower chooses not to repay a funded project, investors cannot claim their investment, either in the form of the loaned token or the minted `MCLR`. For claims to be processed, a manager must transfer the equivalent of the total invested amount in the loaned token by calling the `makeRepayment` function.

```
uint256 totalShare = (project.innerStruct.totalRepaid *
investorShare) / BASIS_POINTS;
```

Recommendation

It is recommended to introduce mechanisms to ensure borrowers repay loans, enabling investors to claim their investments. Consider automating the `makeRepayment` process or requiring collateral to secure repayments.

UTI - Unimplemented Time-Based Interest

Criticality	Minor / Informative
Location	TreasuryETH.sol#L260
Status	Unresolved

Description

The contract implements an interest mechanism applied to the borrower of loaned assets during the repayment stage. The interest is currently applied as a fixed percentage, regardless of the loan's duration. Implementing a time-based interest could incentivize borrowers to repay early.

```
if (project.innerStruct.totalRepaid >= project.totalInvested
    + ((project.totalInvested *
(project.innerStruct.platformInterestRate +
project.investorInterestRate)) / BASIS_POINTS)) {
    project.innerStruct.stage = Stage.Repaid;
    emit PrincipalRepayment(_projectId, _amount);
    emit ProjectStatusChanged(_projectId, uint8(Stage.Repaid));
}
```

Recommendation

The team is advised to consider implementing a time-based interest system. This approach would adjust the interest based on the duration of the loan, thereby incentivizing borrowers to repay their loans early and potentially reducing the overall interest burden.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	TreasuryETH.sol#L17,25,44 Sell.sol#L21,37,57,70,77,84 ManagerRegistry.sol#L26,46,54 Fundraise.sol#L93,94,95,96,97,117,129,185,203,222,247,275,285,356,357,358,373,386,407,437,445,452,465 Burn.sol#L16,24
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address public MCLR
address _mclr
address _managerRegistry
address _recepient
uint256 _amount
address _token
address _usdt
uint256 _addedPercentage
address _fundraise
uint256 _usdtAmount
bool _status
address _superManager
address _manager
address _sender

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Sell.sol#L61 Fundraise.sol#L301,302,315,317,477,478,490,492
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 mclrAmount = (_usdtAmount / (price + (price / 1000)) *
addedPercentage)

uint256 investorShare = (investor.investedAmount * BASIS_POINTS) /
project.totalInvested
uint256 totalShare = (project.innerStruct.totalRepaid *
investorShare) / BASIS_POINTS
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	TreasuryETH.sol#L30,31 Sell.sol#L42,43,44,46 Fundraise.sol#L103,104,105,106,454 Burn.sol#L29,30
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
MCLR = _mclr
managerRegistry = _managerRegistry
fundraise = _fundraise
usdt = _usdt
treasury = _treasury
burn = _burn
trustedSigner = _trustedSigner
trustedSigner = _signer
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	Fundraise.sol#L516
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            /*
            First 32 bytes stores the length of the signature

            add(sig, 32) = pointer of sig + 32
            effectively, skips first 32 bytes of signature
...
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
            // second 32 bytes
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	MCLR.sol#L3
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.23;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Treasury	Implementation	Initializable, OwnableUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	claim	External	✓	-
Sell	Implementation	Initializable, OwnableUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	sell	External	✓	-
	claim	External	✓	-
	setPercentage	External	✓	-
	setPauseStatus	External	✓	-

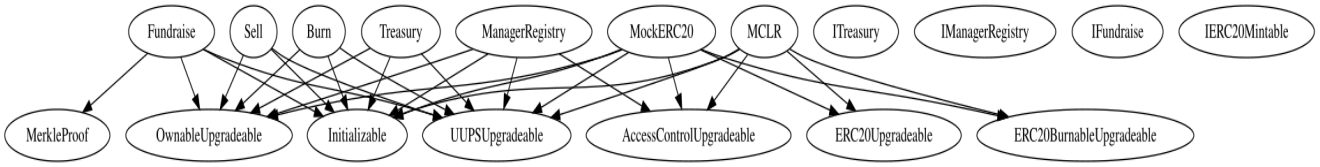
MockERC20	Implementation	Initializable, ERC20Upgradeable, ERC20BurnableUpgradeable, AccessControlUpgradeable, OwnableUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	mint	Public	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyOwner
ManagerRegistry	Implementation	Initializable, AccessControlUpgradeable, OwnableUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	setManagerStatus	External	✓	onlyRole
	isAllowedCall	Public		-
MCLR	Implementation	Initializable, ERC20Upgradeable, ERC20BurnableUpgradeable, AccessControlUpgradeable, UUPSUpgradeable		

		Public	✓	-
	initialize	Public	✓	initializer
	mint	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole
Fundraise	Implementation	Initializable, UUPSUpgradeable, OwnableUpgradeable, MerkleProof		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	invest	External	✓	-
	investUpdate	External	✓	-
	_invest	Internal	✓	
	withdrawInvestment	External	✓	-
	cancelProject	External	✓	-
	transferFundsToBorrower	External	✓	-
	makeRepayment	External	✓	-
	claimLoyalty	External	✓	-
	claim	External	✓	-
	_claim	Internal	✓	
	createProject	External	✓	-
	setMCLRPrice	External	✓	-
	moveProjectStage	External	✓	-
	setProject	External	✓	-

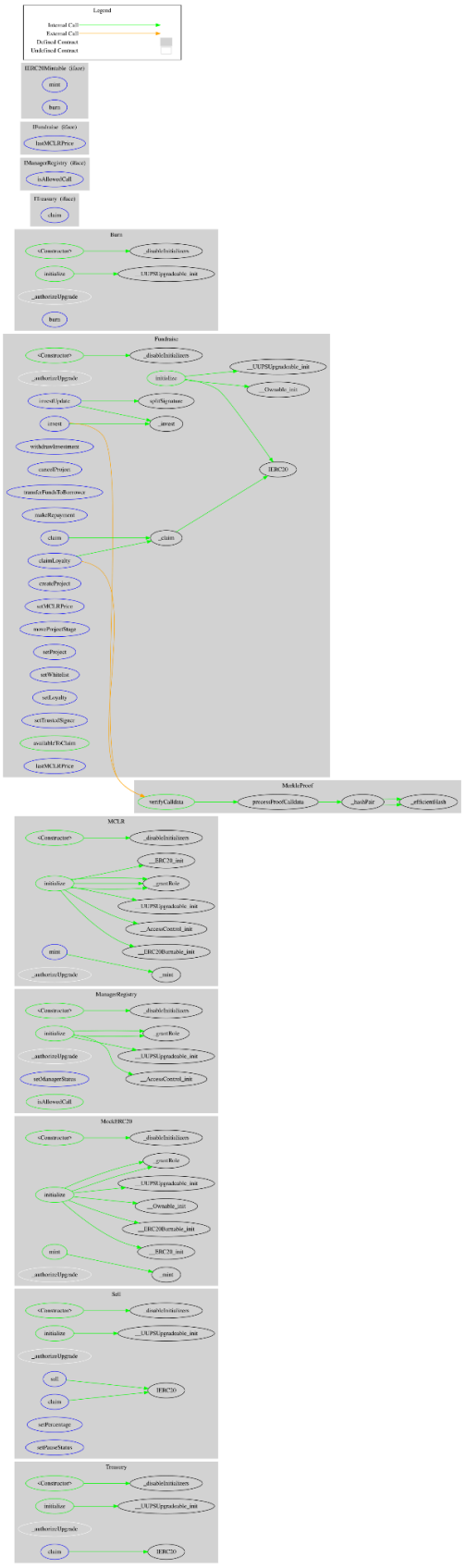
	setWhitelist	External	✓	-
	setLoyalty	External	✓	-
	setTrustedSigner	External	✓	-
	availableToClaim	Public		-
	lastMCLRPrice	External		-
	splitSignature	Public		-
Burn	Implementation	Initializable, OwnableUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	burn	External	✓	-
MerkleProof	Implementation			
	verifyCalldata	Public		-
	processProofCalldata	Internal		
	_hashPair	Private		
	_efficientHash	Private		
ITreasury	Interface			
	claim	External	✓	-
IManagerRegistry	Interface			

	isAllowedCall	External		-
IFundraise	Interface			
	lastMCLRPrice	External		-
IERC20Mintable	Interface			
	mint	External	✓	-
	burn	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IERC20Mintable (Iface)
mint
burn
IFundraise (Iface)
lastMCLRPrice
IManagerRegistry (Iface)
isAllowedCall
ITreasury (Iface)
claim
Burn
<Constructor>
_disableInitializers
initialize
_UUPSUpgradeable_init
_authorizeUpgrade
burn
Fundraise
<Constructor>
_disableInitializers
_authorizeUpgrade
initialize
_UUPSUpgradeable_init
_Ownable_init
createUpdate
updSignature
invest
_invest
withdrawInvestment
cancelProject
transferFundsToBeneficiary
makeRepayment
claim
_claim
IERC20
claimLoyalty
createProject
setMCLRPrice
moveProjectStage
setProject
setWhitelist
setLoyalty
setTrustedSigner
availableToClaim
lastMCLRPrice
MerkleProof
verifyCalldata
processProofCalldata
_hashPair
_efficientHash
MCLR
<Constructor>
_disableInitializers
initialize
__ERC20_init
_grantRole
_UUPSUpgradeable_init
__AccessControl_init
__ERC20Burnable_init
mint
_mint
_authorizeUpgrade
ManagerRegistry
<Constructor>
_disableInitializers
initialize
_grantRole
_UUPSUpgradeable_init
__AccessControl_init
_authorizeUpgrade
setManagerStatus
isAllowedCall
MockERC20
<Constructor>
_disableInitializers
initialize
_grantRole
_UUPSUpgradeable_init
_Ownable_init
__ERC20Burnable_init
__ERC20_init
mint
_mint
_authorizeUpgrade
Sell
<Constructor>
_disableInitializers
initialize
_UUPSUpgradeable_init
_authorizeUpgrade
sell
IERC20
claim
setPercentage
setPauseStatus
Treasury
<Constructor>
_disableInitializers
initialize
_UUPSUpgradeable_init
_authorizeUpgrade
claim
IERC20

Summary

8lends is an interesting project that has a friendly and growing community. Its contracts implement mechanisms for the funding of projects from whitelisted investors and the borrowing of the invested assets. The Smart Contract analysis reported no compiler error or critical issues. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io