Cyberscope

Audit Report

BlaroThings

March 2025

marketplace.sol 3343add12bf40762bc5177cd2fef635c64e31f1b591172033fde751b259a8aec

investmentPool.sol 576f2db4a1bdc8b53529ab60b65da9e8118772d506a5f0aa77e6ea8446507106

FactoryPool.sol 929298e6d10e7fd2741360fc3e5bafe4a3c03315ae8e005cf1d6314d663ceb83

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

Audit Updates

Initial Audit	13 Jan 2025
Corrected Phase 2	23 Jan 2025
Corrected Phase 3	11 Mar 2025

Source Files

Filename	SHA256
marketplace.sol	3343add12bf40762bc5177cd2fef635c64e31f1b591172033fde751b259a8aec
investmentPool.sol	576f2db4a1bdc8b53529ab60b65da9e8118772d506a5f0aa77e6ea8446507106
FactoryPool.sol	929298e6d10e7fd2741360fc3e5bafe4a3c03315ae8e005cf1d6314d663ceb83

Overview

The BlaroThings Ecosystem consists of several interconnected smart contracts. This audit focuses on the InvestmentPool, PoolFactory, and Marketplace that together provide a comprehensive platform for secure investments, pool management, and secondary trading of investment shares. These contracts work seamlessly to enable users to participate in investment opportunities, manage pools, and trade shares in a decentralized and efficient manner.

InvestmentPool

The InvestmentPool contract is at the core of the ecosystem, providing a decentralized mechanism for managing investments in approved stablecoins. Users can deposit stablecoins into a pool with a defined maximum capacity and lifespan, receiving shares proportional to their contributions. The pool closes to new investments once it reaches its capacity or maturity date, after which users can withdraw their funds.

Key features include:

- **Early Withdrawals:** Investors can withdraw their funds before maturity, incurring a predefined penalty.
- **Profit Distribution:** Pool owners can distribute profits to investors based on their share holdings.
- **Administrative Control:** Pool owners can manage approved stablecoins, close pools, and distribute profits securely.

The contract ensures robust security with reentrancy protection and strict access controls, providing a reliable environment for investments.

PoolFactory

The PoolFactory contract acts as a factory for creating and managing multiple instances of InvestmentPool. It streamlines the deployment of investment pools by allowing the contract owner to configure key parameters such as capacity, lifespan, and early withdrawal penalties.

Features include:

- **Pool Creation:** Deploys new pools and tracks them in a structured manner.
- **Pool Management:** Enables verification and interaction with registered pools to ensure integrity.
- **Administrative Control:** Allows the owner to close pools, ensuring lifecycle management.

By simplifying the creation and management of pools, the PoolFactory provides a scalable and secure way to manage multiple investment opportunities.

Marketplace

The Marketplace contract facilitates a secondary market for trading shares in InvestmentPool contracts. Using the native BLR token, users can list, buy, and delist shares, enabling liquidity and flexibility within the ecosystem. The marketplace charges a trading fee, which is accumulated and managed by the contract owner.

Key functionalities:

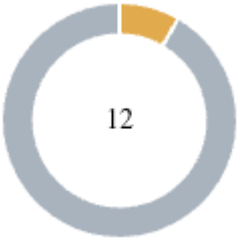
1. **Listing Shares:** Sellers can list shares from specific pools for sale with a specified price per share.
2. **Purchasing Shares:** Buyers can acquire listed shares, with the contract handling payment and fee distribution.
3. **Delisting Shares:** Sellers can partially or fully remove their unsold shares from listings.
4. **Fee Management:** The contract collects a trading fee on each purchase, which is withdrawn by the owner to support operations.

The Marketplace ensures smooth integration with InvestmentPool contracts, though certain aspects (e.g., reliance on specific functions) require careful alignment with the pools' intended behavior.

Summary

The BlaroThings Ecosystem provides a robust framework for decentralized investments and trading. InvestmentPool enables secure and managed investment operations, PoolFactory simplifies the deployment and management of multiple pools, and Marketplace enhances liquidity by offering a secondary trading platform for shares. Together, these contracts deliver a comprehensive and integrated solution for decentralized financial ecosystems.

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	11

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	11	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	ISTR	Incorrect Stablecoin Tokens Recipient	Unresolved
Minor / Informative	AME	Address Manipulation Exploit	Unresolved
Minor / Informative	ITA	Incorrect Transfer Amount	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	PBV	Percentage Boundaries Validation	Unresolved
Minor / Informative	PRE	Potential Reentrance Exploit	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	RCC	Redundant Condition Checks	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L16	Validate Variable Setters	Unresolved
Minor / Informative	L19	Stable Compiler Version	Unresolved
Minor / Informative	L20	Succeeded Transfer Check	Unresolved

ISTR - Incorrect Stablecoin Tokens Recipient

Criticality	Medium
Location	marketplace.sol#L232
Status	Unresolved

Description

As part of the share delisting process, the contract invokes the `withdrawAfterMaturity` function of the InvestmentPool contract, which transfers the stablecoin payout to the `msg.sender`.However, the `msg.sender` in this context is the Marketplace contract, not the listing seller. As a result, the stablecoin tokens intended for the user are instead transferred to the Marketplace contract, leaving the user without the tokens they listed in the platform.

```
investmentPool.withdrawAfterMaturity();
```

Recommendation

The team is advised to take these segments into consideration and rewrite them to ensure the payout is directed to the correct user. The team should also ensure that only valid calls to `withdrawAfterMaturity` are allowed by adding appropriate permission checks, such as verifying the caller is authorized to withdraw on behalf of the user.

AME - Address Manipulation Exploit

Criticality	Minor / Informative
Location	marketplace.sol#L103,104
Status	Unresolved

Description

The contract's design includes functions that accept external contract addresses as parameters without performing adequate validation or authenticity checks. This lack of verification introduces a significant security risk, as input addresses could be controlled by attackers and point to malicious contracts. Such vulnerabilities could enable attackers to exploit these functions, potentially leading to unauthorized actions or the execution of malicious code under the guise of legitimate operations.

The `listShares` function does not validate the `pool` exists in the `existingPools` variable of the PoolFactory contract.

```
IInvestmentPool investmentPool = IInvestmentPool(pool);
(, , uint256 share, ) = investmentPool.investors(msg.sender);
```

Recommendation

To mitigate this risk and enhance the contract's security posture, it is imperative to incorporate comprehensive validation mechanisms for any external contract addresses passed as parameters to functions. This could include checks against a whitelist of approved addresses, verification that the address implements a specific contract interface, or other methods that confirm the legitimacy and integrity of the external contract. Implementing such validations helps prevent malicious exploits and ensures that only trusted contracts can interact with sensitive functions.

Exploit Scenario: Suppose that a malicious actor takes advantage of the lack of validation for the address parameter in the function of the smart contract. Here's a step-by-step breakdown of the exploit:

1. The attacker deploys a malicious contract that implements the investors function to return fake data when queried.
2. The attacker calls the listShares function on the Marketplace contract, providing the address of their malicious contract as the pool parameter. The function:
    - Treats the malicious contract as a valid pool.
    - Calls the investors function of the malicious contract, which returns a maximum share value to bypass the share >= amount check.
    - Creates a listing for the pool with any specified amount and price.
3. Impact:
    - The Marketplace contract now contains a fraudulent listing associated with the attacker’s malicious pool.
    - Unsuspecting users interacting with the listing may believe it is legitimate and attempt to purchase shares. Since purchase logic relies on additional interactions with the pool, the malicious contract could execute arbitrary code or steal user funds.
    - This could lead to significant losses for users and reputational damage for the platform.

ITA - Incorrect Transfer Amount

Criticality	Minor / Informative
Location	marketplace.sol#L155,162
Status	Unresolved

Description

The `purchaseShares` function transfers the `sellerAmount` (representing the total value of the purchased shares) directly to the seller. However, while the contract calculates a fee for each purchase and adds it to the `accumulatedFees` variable, it does not transfer the fee amount to the contract's address. As a result, the `accumulatedFees` variable holds a misleading value since the actual fees are not physically present in the contract balance. This issue undermines the fee collection mechanism and could impact the contract's revenue model or other operations reliant on accurate fee accumulation.

```
// Transfer the total price from buyer to seller
IBLRToken(blrToken).transferFrom(
    msg.sender,
    listing.seller,
    sellerAmount
);

// Accumulate the fee in the contract
accumulatedFees += fee;
```

Recommendation

The team is recommended to modify the logic to ensure that the fee amount is transferred to the contract's address separately from the seller's payment. Addtionally, the fee accounting logic should be retained to reflect the correct accumulated fees. By implementing these changes, the contract will maintain an accurate fee accumulation mechanism, ensuring its revenue model operates as intended.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	investmentPool.sol#L79
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
approvedStablecoins[_stablecoin] = _status;
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	investmentPool.sol#L70
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
earlyWithdrawalPenaltyPercent = _earlyWithdrawalPenaltyPercent;
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

PRE - Potential Reentrance Exploit

Criticality	Minor / Informative
Location	investmentPool.sol#L145,164
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the transfer method of the staclecoin token. The staclecoin contract could have a malicious transfer method that makes a callback to the `earlyWithdraw` or the `withdrawAfterMaturity` method. The re-entrance exploit could be used by a malicious user to drain the contract's funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

```
function earlyWithdraw() external poolIsActive nonReentrant {
    Investor storage investor = investors[msg.sender];
    require(investor.depositedAmount > 0, "No deposit found");
    require(!investor.hasWithdrawn, "Already withdrawn");

    uint256 penalty = (investor.depositedAmount *
        earlyWithdrawalPenaltyPercent) / 100;
    uint256 payout = investor.depositedAmount - penalty;

    IERC20(investor.stablecoin).transfer(msg.sender, payout);

    investor.share = 0; // "Burn" the shares
    investor.hasWithdrawn = true;
    totalDeposits -= investor.depositedAmount; // Reduce total pool deposits

    emit EarlyWithdrawal(msg.sender, penalty, payout);
}
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Do Not allow contract addresses to receive funds.
- Proceed with the external call as the last statement of the method, so that the state will have been updated properly during the re-entrance phase.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	investmentPool.sol#L94,97,100
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is 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

```
IERC20(_stablecoin).transferFrom(msg.sender, address(this), _amount);

uint256 shares = calculateShares(_amount, _stablecoin);
investors[msg.sender] = Investor(_amount, _stablecoin, shares, false);
investorList.push(msg.sender);

totalDeposits += _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
```

RCC - Redundant Condition Checks

Criticality	Minor / Informative
Location	marketplace.sol#L203,228,229
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

As part of the share delisting process, the contract checks for certain conditions redundantly. For instance, the contract verifies if `block.timestamp >= poolEndTime(pool)` before allowing withdrawal. This check is already performed within the `withdrawAfterMaturity` function, which is subsequently called. This redundancy increases gas consumption without adding security benefits.

```
require(
    block.timestamp >= poolEndTime(pool),
    "Pool has not matured yet"
);
require(share > 0, "No shares to withdraw");
require(!hasWithdrawn, "Already withdrawn");
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	investmentPool.sol#L75,82 FactoryPool.sol#L21,22,23,24,67,68,69
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.

```
bool _status
address _stablecoin
uint256 _amount
uint256 _maxCapacity
uint256 _poolLifespan
uint256 _earlyWithdrawalPenaltyPercent
address[] memory _approvedStablecoins
address _poolAddress
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	marketplace.sol#L94
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.

```
blrToken = _blrToken
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	marketplace.sol#L2 investmentPool.sol#L2
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 ^0.8.0;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	marketplace.sol#L155,259 investmentPool.sol#L94,154,171,195
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IBLRToken(blrToken).transferFrom(
            msg.sender,
            listing.seller,
            sellerAmount
        )
IBLRToken(blrToken).transfer(owner(), amount)
IERC20(_stablecoin).transferFrom(msg.sender, address(this), _amount)
IERC20(investor.stablecoin).transfer(msg.sender, payout)
IERC20(investor.stablecoin).transfer(investorAddress, profitShare)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

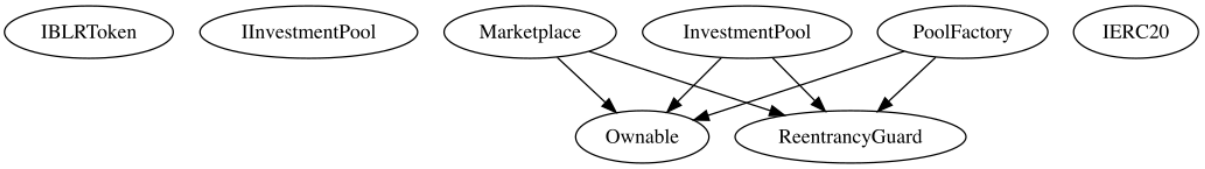
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IBLRToken	Interface			
	balanceOf	External		-
	allowance	External		-
	transferFrom	External	✓	-
	transfer	External	✓	-
IInvestmentPool	Interface			
	deposit	External	✓	-
	earlyWithdraw	External	✓	-
	withdrawAfterMaturity	External	✓	-
	modifyApprovedStablecoin	External	✓	-
	isPoolFullySubscribed	External		-
	totalDeposits	External		-
	poolActive	External		-
	poolEndTime	External		-
	investors	External		-
Marketplace	Implementation	Ownable, ReentrancyGuard		

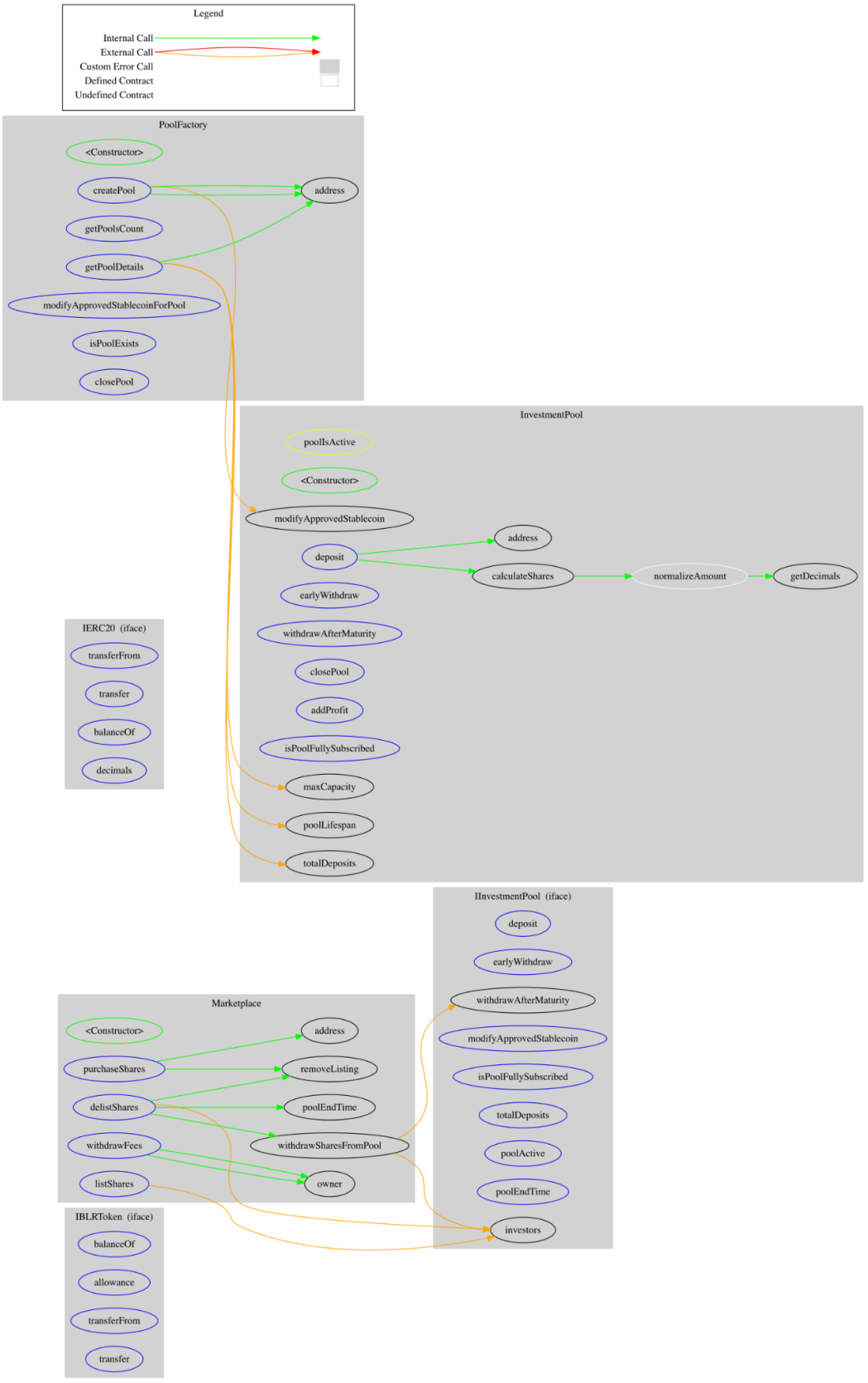
		Public	✓	Ownable
	listShares	External	✓	nonReentrant
	purchaseShares	External	✓	nonReentrant
	delistShares	External	✓	nonReentrant
	withdrawSharesFromPool	Internal	✓	
	poolEndTime	Internal		
	removeListing	Internal	✓	
	withdrawFees	External	✓	onlyOwner nonReentrant
IERC20	Interface			
	transferFrom	External	✓	-
	transfer	External	✓	-
	balanceOf	External		-
	decimals	External		-
InvestmentPool	Implementation	ReentrancyGuard, Ownable		
		Public	✓	Ownable
	modifyApprovedStablecoin	External	✓	onlyOwner
	deposit	External	✓	poolIsActive nonReentrant
	getDecimals	Internal		
	normalizeAmount	Internal		
	calculateShares	Internal		

	earlyWithdraw	External	✓	poolIsActive nonReentrant
	withdrawAfterMaturity	External	✓	nonReentrant
	closePool	External	✓	onlyOwner
	addProfit	External	✓	onlyOwner nonReentrant
	isPoolFullySubscribed	External		-
PoolFactory	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	createPool	External	✓	onlyOwner nonReentrant
	getPoolsCount	External		-
	getPoolDetails	External		-
	modifyApprovedStablecoinForPool	External	✓	onlyOwner nonReentrant
	isPoolExists	External		-
	closePool	External	✓	onlyOwner nonReentrant

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
PoolFactory
<Constructor>
createPool
address
getPoolsCount
getPoolDetails
modifyApprovedStablecoinForPool
isPoolExists
closePool
InvestmentPool
poolIsActive
<Constructor>
modifyApprovedStablecoin
deposit
address
calculateShares
normalizeAmount
getDecimals
earlyWithdraw
withdrawAfterMaturity
closePool
addProfit
isPoolFullySubscribed
maxCapacity
poolLifespan
totalDeposits
IERC20 (iface)
transferFrom
transfer
balanceOf
decimals
IInvestmentPool (iface)
deposit
earlyWithdraw
withdrawAfterMaturity
modifyApprovedStablecoin
isPoolFullySubscribed
totalDeposits
poolActive
poolEndTime
investors
Marketplace
<Constructor>
address
purchaseShares
removeListing
delistShares
poolEndTime
withdrawFees
withdrawSharesFromPool
listShares
owner
IBLRToken (iface)
balanceOf
allowance
transferFrom
transfer

Summary

BlaroThings contract implements a financial mechanism. 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