Cyberscope

Audit Report

BlockATM V2

April 2025

SHA256

1e42612d69d651f11968baa97b8b15ae96acffd8d60232c7812e5bc79b42af98

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	09 Apr 2025

Source Files

Filename	SHA256
IBlockFee.sol	bde8c080a225093a330b3c94bcc9dc6078af6993d6acad6de6fd050d17bea312
IBlockATMPayout.sol	d0a413c0a22c6a858e1754a309c8eb14ceb05a0add541e225add79509be81f33
IBlockATMCustomer.sol	4f84fdef6ee8747ce195cdbc036631f1a23eb615b906aa16731b313eb004dbc0
BlockUtils.sol	1d33f257f8e4b690926b8eb5594656e875a7bc9ee583797fdb8877fe4fff8197
BlockFee.sol	98d899a8d6db9ec3f7198d6f4a3834ab3867185a4796a815acbc81679cc1b46a
BlockCommon.sol	3d31535c7f1f0a27377acdc5bf9978f4c245ad17ed651a8d7d663716171d14e8
BlockATMProxyPayout.sol	eb23d43d53e368d69d735dbb2de578c497926893416e6204e145423165ef8899
BlockATMPayout.sol	8501f28abd693bf0d754eb268abcf2e018ca52161334c844de06bd928294466a
BlockATMCustomer.sol	ba10a7198b0df2f9a2a00f6fdf6c35cdd5c84b2e3d7a59a84672c95769c867aa
BlockATMCollect.sol	2bf515acea7b89a8df28790fa8a022c520a2f08e6d8f26b51b7403e0275375e2

BaseCustomer.sol	69aedd4835eccc0cfb7cae2c0c854242dbf0c56c7397c997e290239b9adbc3c4

Contract Readability Comment

The assessment of the smart contract has highlighted several areas of concern regarding its readability and adherence to best practices. The codebase appears segmented and does not fully align with fundamental coding principles, which can hinder both readability and the review process. To enhance the contract's stability, security, and long-term viability, it is recommended to undertake a comprehensive code refactor. Simplifying and restructuring the code to better align with best practices and coding standards will be essential for improving maintainability and ensuring the contract is production-ready. As it currently stands, the architecture of the contract is not suitable for production deployment.

Overview

BlockCommon

- **transferFrom:**
  Transfers a specified amount of tokens from the "from" address to the "to" address, provided that the "from" address has approved the necessary allowance for the calling contract.
- **withdrawCommon:**
  Transfers a specified amount of tokens from the contract's balance to the designated withdrawal address.

BlockFee

- **subFeeCommon:**
  Transfers a specified amount of tokens from the "from" address to this contract for the given tokenAddress. The "from" address must have approved the necessary allowance to the contract. Then, it forwards the specified amount of tokens to the feeReceiverAddress.
- **subFee:**
  Calls subFeeCommon for a specified amount and tokenAddress and transfers tokens to a fee receiver.

BlockATMCustomer

- **depositToken:**
  Deposits a specified amount of tokens into the contract from the caller's address. It checks if the contract is not in a burn state and then transfers the tokens to the contract's address.
- **calcFee:**
  Calculates the fee amount based on the number of times a specific token has been deposited. It retrieves the decimals of the token and computes the fee as twice the count of deposits.
- **withdrawToken:**
  Withdraws tokens from the contract to a specified address, supporting both stable

and non-stable coins. It verifies the withdrawal address and the validity of the withdrawal information.

BlockATMPayout

- **safeTransferToProxy:**
  Safely transfers a specified amount of ERC20 tokens to the proxy payout address. This function can only be called by the designated proxy address, as enforced by the `onlyProxy` modifier.
- **transferToProxy:**
  Transfers a specified amount of ERC20 tokens to the proxy payout address without the safety checks of the `safeTransfer` method. Similar to `safeTransferToProxy`, this function is restricted to the proxy address through the `onlyProxy` modifier.

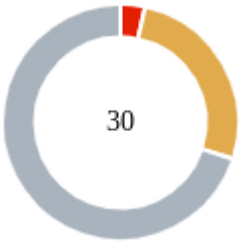
BlockATMProxyPayout

- **payoutByWallet:**
  Initiates a payout directly from the user's wallet. It accepts a `Payout` struct, an array of order numbers, an array of recipient addresses, and an array of amounts to be paid. The function calls `payoutToken` to handle the payout process.
- **payoutByContract:**
  Facilitates a payout from a specified contract address. This function can only be called by addresses that are recognized as financial addresses, as enforced by the `onlyFinancials` modifier. It accepts similar parameters as `payoutByWallet` and calls `payoutToken` to execute the payout.
- **payoutByProxy:**
  Allows a payout to be executed through a proxy address. This function is restricted to recognized user addresses via the `onlyUser` modifier. It also takes a `Payout` struct, order numbers, recipient addresses, and amounts, and calls `payoutToken` to perform the payout.
- **payoutToken:**
  Handles the payout process, including calculating fees and transferring tokens to the contract. It verifies if the token is stable and calls the appropriate transfer functions based on the payout type to transfer tokens to the withdrawing addresses. It processes batch payments to recipients and sends the payout fee to the fee receiver address.

BlockATMCollect

- **bindingRelationship:**
  Establishes a relationship between a set of addresses and a customer address. It binds each address in the provided array to the specified customer address and updates the financial addresses associated with that customer. The function also deducts a fee using the `subFee` method from the `BlockFee` contract and emits a `BindingRelationship` event to log the action.
- **collect:**
  Facilitates the collection of tokens from users associated with a specific customer address. It transfers tokens from users to the contract based on the provided trades and calculates the fees for each currency. The function calls `withdrawToken` to handle the transfer of tokens to the customer address and the fee address. Finally, it emits a `Collect` event with details of the transaction, including the collected amounts and fees.

Findings Breakdown



Critical	1
Medium	8
Minor / Informative	21

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	8	0	0	0
Minor / Informative	21	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	PLOF	Potential Loss Of Funds	Unresolved
Medium	AME	Address Manipulation Exploit	Unresolved
Medium	FCS	Fragmented Code Segments	Unresolved
Medium	ISU	Inconsistent State Update	Unresolved
Medium	IAC	Ineffective Access Control	Unresolved
Medium	PGA	Potential Griefing Attack	Unresolved
Medium	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Medium	TSI	Tokens Sufficiency Insurance	Unresolved
Medium	UAI	Unchecked Array Indices	Unresolved
Minor / Informative	PAI	Payout Amount Inconsistency	Unresolved
Minor / Informative	ALM	Array Length Mismatch	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	HV	Hardcoded Values	Unresolved

●	IDI	Immutable Declaration Improvement	Unresolved
●	MC	Missing Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	IEE	Inaccurate Events Emission	Unresolved
●	PEF	Potentially Excessive Fee	Unresolved
●	UPA	Unchecked Payout Amount	Unresolved
●	UPT	Unchecked Payout Transfer	Unresolved
●	US	Unchecked SubType	Unresolved
●	UTF	Unchecked Transfer Flag	Unresolved
●	UUA	Unsanitized User Arguments	Unresolved
●	UWA	Unsanitized Withdrawal Amount	Unresolved
●	UTPD	Unverified Third Party Dependencies	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

PLOF - Potential Loss Of Funds

Criticality	Critical
Location	BlockATMCollect.sol#L150
Status	Unresolved

Description

The `collect` contract relies on the `transferFrom` method to transfer tokens from addresses included in a `bindingMap` to the `customerAddress`. For the transfer to be processed, users must first approve tokens to the contract address. An attacker can front-run a call to the `bindingRelationship` method to establish a connection with that user as the `customerAddress`. The attacker can then execute the `collect` method to withdraw all approved tokens to the contract, potentially leading to unauthorized access to funds.

```
function collect(bool safe,address[] calldata currency,Trade[] calldata
trade,address customerAddress) onlyFinancials(customerAddress) public returns
(bool){
uint256[] memory total = transferToken(currency,trade,customerAddress);
uint256 fLength = currency.length;
uint256[] memory feeArray = new uint256[](fLength);
address feeAddrees = IBlockFee(feeGateway).feeReceiverAddress();
for (uint256 i = 0; i < fLength; ){
uint256 value = total[i];
uint256 feeAmount =
withdrawToken(safe,currency[i],value,customerAddress,feeAddrees);
feeArray[i] = feeAmount;
unchecked { ++i; }
}
emit Collect(msg.sender,customerAddress,currency,trade,feeArray,feeRate);
return true;
}
```

Recommendation

Approved tokens to the contract should remain inaccessible to users of the system. This can be achieved by implementing proper access controls, particularly by incorporating checks that require users to opt-in to binding with a specific `customerAddress`. By

ensuring that users explicitly consent to the binding relationship, the contract will prevent unauthorized access to user funds.

AME - Address Manipulation Exploit

Criticality	Medium
Location	BlockCommon.sol#L12,20,26,36 BlockFee.sol#L45,50,58
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.

```
function transferFrom(address tokenAddress,address from,address to,uint256
amount) internal checkTokenAddress(tokenAddress) returns(uint256) {
...
}

function transferFrom(address tokenAddress,address from,address to) internal
checkTokenAddress(tokenAddress) returns(uint256) {
...
}

function transferCommon(address tokenAddress,address to,uint256 amount)
internal checkTokenAddress(tokenAddress) checkAmount(amount) returns(uint256)
{
...
}

function withdrawCommon(bool flag,address tokenAddress,address
withdrawAddress,uint256 amount) internal checkAmount(amount)
checkTokenAddress(tokenAddress) checkWithdrawAddress(withdrawAddress) {
...
}
```

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.

FCS - Fragmented Code Segments

Criticality	Medium
Location	BlockUtils.sol BlockFee.sol BlockCommon.sol BlockATMProxyPayout.sol BlockATMPayout.sol BlockATMCustomer.sol BlockATMCollect.sol BaseCustomer.sol
Status	Unresolved

Description

The contracts are excessively fragmented across multiple files and functions. The current implementation relies on numerous internal functions and excessive repetition of code segments. This architecture significantly impacts code readability and future maintenance, making it challenging to maintain and modify the code effectively.

Recommendation

It is advisable to refactor the contracts to reduce fragmentation and improve code organization. This can be achieved by consolidating related functions into relevant files and minimizing code duplication through the use of shared utility functions or libraries. By optimizing the architecture, code readability and maintainability can be enhanced making future review and maintenance processes more efficient.

ISU - Inconsistent State Update

Criticality	Medium
Location	BlockATMCollect.sol#L50
Status	Unresolved

Description

The `BlockATMCollect` contract updates the `financeMap` of a `customerAddress` with elements from the `newOwnerList`. Then, the `newOwnerList` is stored as the `financeList` of the `customerAddress`. If the `bindingRelationship` function is called a second time, the `newOwnerList` will override the contents of the `financeList`; however, the `financeMap` of the previous `newOwnerList` remains active and is extended with the new elements.

```
function bindingRelationship(bool safe,uint256 id,address[] calldata
array,address[] calldata newOwnerList,address customerAddress) public returns
(bool){
uint256 length = array.length;
for (uint256 i = 0; i < length; ) {
address addr = array[i];
require(relationMap[addr] == address(0), "Address has been bound");
relationMap[addr] = customerAddress;
unchecked { ++i; }
}
uint256 oLenth = newOwnerList.length;
User storage v = userMap[customerAddress];
for (uint256 i = 0; i < oLenth; ) {
address addr = newOwnerList[i];
v.financeMap[addr] = true;
unchecked { ++i; }
}
v.financeList = newOwnerList;
v.owner = msg.sender;
IBlockFee(feeGateway).subFee(safe,msg.sender,id,3);
emit BindingRelationship(msg.sender,array,newOwnerList,customerAddress);
return true;
}
```

Recommendation

It is advisable to implement a mechanism to ensure that updates to the `financeMap` and `financeList` are consistent and reflect the current state of ownership. This could involve clearing or appropriately managing previous entries when new data is introduced.

IAC - Ineffective Access Control

Criticality	Medium
Location	BlockATMCollect.sol#L36
Status	Unresolved

Description

The `collect` function implements the `onlyFinancials` modifier and can only be called by addresses that have been included in the `financeMap` of the `customerAddress`. However, any user can call the `bindingRelationship` method and set themselves as active in the `financeMap`. Therefore, any user can effectively call the `collect` method and initiate the process for any arbitrary `customerAddress`.

```
modifier onlyFinancials(address customerAddress){
require(userMap[customerAddress].financeMap[msg.sender],"Not a financial
address");
_;
}
```

Recommendation

It is advisable to implement additional checks to ensure that only authorized addresses can call the `bindingRelationship` method and modify the `financeMap`. This could involve verifying the identity of the caller or requiring specific permissions before allowing updates. By doing so, you can prevent unauthorized users from gaining access to the collect method and initiating the process.

PGA - Potential Griefing Attack

Criticality	Medium
Location	BlockATMCollect.sol#L66,95
Status	Unresolved

Description

The contract is vulnerable to griefing attacks, where malicious actors can exploit the contract's logic to interfere with legitimate user operations.

In this case, the contract implements the `bindingRelationship` function to set up relationships between users and a `customerAddress`. The caller of the method is then set as the owner of the binding structure. Therefore, third party users can call this method to set themselves as the owner. The latter is then able to delete all binding information through the `deleteRelationship` function.

Such griefing attacks could undermine the contract's usability and obstruct legitimate user operations.

```
v.owner = msg.sender;
```

```
function deleteRelationship(address[] calldata array,address customerAddress)
onlyUserOwner(customerAddress) public returns (bool){
deleteRelationshipCommon(array, customerAddress);
return true;
}
```

Recommendation

The team is advised to review the transfer mechanism to ensure that all legitimate operations are processed as intended. This will help maintain the integrity of user activities and strengthen trust in the system.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Medium
Location	BlockFee.sol#L50
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

```
function subFeeCommon(bool safe,address tokenAddress,address from,uint256
amount,uint256 id,uint256 subType) internal {
super.transferFrom(tokenAddress,from,address(this),amount);
if (amount > 0){
super.withdrawCommon(safe, tokenAddress, feeReceiverAddress, amount);
}
emit SubFee(from,
tokenAddress,feeReceiverAddress,amount,id,subType,msg.sender);
}
```

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
```

TSI - Tokens Sufficiency Insurance

Criticality	Medium
Location	BlockATMPayout.sol
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks. In particular, the `from` address is expected to hold the necessary funds however no such functionality is implemented to deposit funds to the `BlockATMPayout` contract.

```
function _transferStableTokens(Payout calldata payout,address from,uint256
payType, uint256 feeAmount) private {
...
else if (payType == 1 || payType == 3) {
uint256 all = payout.total + feeAmount + payout.gasAmount;
if (payout.safe) {
IBlockATMPayout(from).safeTransferToProxy(payout.tokenAddress, all);
} else {
IBlockATMPayout(from).transferToProxy(payout.tokenAddress, all);
}
}
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

UAI - Unchecked Array Indices

Criticality	Medium
Location	BlockATMCollect.sol#L131
Status	Unresolved

Description

The transferToken method is updating the elements of a `total` array using user provided indices. The `collect` is the accessing the `total` array assuming its elements span the `[0,fLength]` range. However, this may not be true if the indices passed by the user are not properly sanitized.

```
total[t.index] += sendAmount;
```

```
for (uint256 i = 0; i < fLength; ){
uint256 value = total[i];
...
}
```

Recommendation

It is advisable to implement input validation and sanitization for the indices provided by the user. This should include checks to ensure that the indices are within the expected range. Thereby preventing out-of-bounds access and ensuring the integrity of the contract.

PAI - Payout Amount Inconsistency

Criticality	Minor / Informative
Location	BlockATMProxyPayout.sol#L127,134,136,140
Status	Unresolved

Description

The `payoutToken` method uses the `Payout` method to calculate and transfer an amount of tokens to the `BlockATMProxyPayout`. It then processes the payments using the `amount` array. If the tokens transferred to the contract are not equal to `amount` array a portion of the amount of tokens will remain in the contract.

```
function payoutToken(Payout calldata payout,address from,uint256
payType,string[] calldata orderNo, address[] calldata array, uint256[]
calldata amount, uint256 id) internal {
...
if (stable){
_transferStableTokens(payout, from, payType, feeAmount);
} else {
_transferTokens(payout, from, payType, feeAmount);
require(IBlockFee(feeGateway).isSupportedFeeToken(payout.feeTokenAddress),"fee
TokenAddress is not Support");
}
_processBatchPayments(payout, array, amount, length);
_sendPayoutFee(payout,feeAmount,stable);
...
}
```

Recommendation

It is recommended to ensure that the amount of tokens received is equal to the total amount to be transferred to the receivers.

ALM - Array Length Mismatch

Criticality	Minor / Informative
Location	BlockATMCollect.sol#L150
Status	Unresolved

Description

The contract is designed to handle the process of elements from arrays through functions that accept multiple arrays as input parameters. These functions are intended to iterate over the arrays, processing elements from each array in a coordinated manner. However, there are no explicit checks to verify that the lengths of these input arrays are equal. This lack of validation could lead to scenarios where the arrays have differing lengths, potentially causing out-of-bounds access if the function attempts to process beyond the end of the shorter array. Such situations could result in unexpected behavior or errors during the contract's execution, compromising its reliability and security.

```
function collect(bool safe,address[] calldata currency,Trade[] calldata
trade,address customerAddress) onlyFinancials(customerAddress) public returns
(bool){
uint256[] memory total = transferToken(currency,trade,customerAddress);
uint256 fLength = currency.length;
uint256[] memory feeArray = new uint256[](fLength);
address feeAddrees = IBlockFee(feeGateway).feeReceiverAddress();
for (uint256 i = 0; i < fLength; ){
uint256 value = total[i];
uint256 feeAmount =
withdrawToken(safe,currency[i],value,customerAddress,feeAddrees);
feeArray[i] = feeAmount;
unchecked { ++i; }
}
emit Collect(msg.sender,customerAddress,currency,trade,feeArray,feeRate);
return true;
}
```

Recommendation

To mitigate this, it is recommended to incorporate a validation check at the beginning of the function that accepts multiple arrays to ensure that the lengths of these arrays are identical. This can be achieved by implementing a conditional statement that compares the lengths of the arrays, and reverts the transaction if the lengths do not match. Such a validation step will prevent out-of-bounds errors and ensure that elements from each array are processed in a paired and coordinated manner, thus preserving the integrity and intended functionality of the contract.

CR - Code Repetition

Criticality	Minor / Informative
Location	BlockCommon.sol#L12,20,26,36 BlockFee.sol#L45,50,58
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is strongly advisable to minimize code repetition where possible.

```
function transferFrom(address tokenAddress,address from,address to,uint256
amount) internal checkTokenAddress(tokenAddress) returns(uint256) {
...
}

function transferFrom(address tokenAddress,address from,address to) internal
checkTokenAddress(tokenAddress) returns(uint256) {
...
}

function transferCommon(address tokenAddress,address to,uint256 amount)
internal checkTokenAddress(tokenAddress) checkAmount(amount) returns(uint256)
{
...
}

function withdrawCommon(bool flag,address tokenAddress,address
withdrawAddress,uint256 amount) internal checkAmount(amount)
checkTokenAddress(tokenAddress) checkWithdrawAddress(withdrawAddress) {
...
}
```

```
function subFee(bool safe,address tokenAddress,address from,uint256
amount,uint256 id,uint256 subType) onlyBlockUser(from) public returns (bool) {
...
}

function subFeeCommon(bool safe,address tokenAddress,address from,uint256
amount,uint256 id,uint256 subType) internal {
...
}


function subFee(bool safe,address from,uint256 id,uint256 subType)
onlyBlockUser(from) public returns (bool) {
...
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	BlockATMCustomer.sol#L67
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 withdrawToken(bool safe,Withdraw[] calldata withdrawInfo,address
withdrawAddress,address feeTokenAddress) public onlyFinance returns (bool) {
...
}
```

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.

HV - Hardcoded Values

Criticality	Minor / Informative
Location	BlockATMCollect.sol#L67 BlockATMCustomer.sol#L38 BlockATMPayout.sol#L18
Status	Unresolved

Description

The contract contains multiple instances where numeric values are directly hardcoded into the code logic rather than being assigned to constant variables with descriptive names. Hardcoding such values can lead to several issues, including reduced code readability, increased risk of errors during updates or maintenance, and difficulty in consistently managing values throughout the contract. Hardcoded values can obscure the intent behind the numbers, making it challenging for developers to modify or for users to understand the contract effectively.

```
IBlockFee(feeGateway).subFee(safe,msg.sender,id,2);
```

Recommendation

It is recommended to replace hardcoded numeric values with variables that have meaningful names. This practice improves code readability and maintainability by clearly indicating the purpose of each value, reducing the likelihood of errors during future modifications. Additionally, consider using constant variables which provide a reliable way to centralize and manage values, improving gas optimization throughout the contract.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	BlockATMProxyPayout.sol#L22 BlockATMPayout.sol#L13 BlockATMCollect.sol#L33
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
feeGateway
proxyPayoutAddress
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MC - Missing Check

Criticality	Minor / Informative
Location	BlockATMCollect.sol#L55
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. Specifically, the contract does not verify that the `customerAddress` is the non-zero address.

```
relationMap[array[i]] = customerAddress;
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	BlockCommon.sol
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.

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.

IEE - Inaccurate Events Emission

Criticality	Minor / Informative
Location	BlockATMProxyPayout.sol#L144
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that may result in the emission of misleading events. Emitting accurate 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. In this case, the `orderNo` is passed by the user and may result in the emission of misleading events.

```
function payoutToken(Payout calldata payout,address from,uint256
payType,string[] calldata orderNo, address[] calldata array, uint256[]
calldata amount, uint256 id) internal {
...
emit PayoutToken(from, payout, payType, feeAmount, orderNo, array, amount,
msg.sender, id);
}
```

Recommendation

It is recommended to implement representative events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details for 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.

PEF - Potentially Excessive Fee

Criticality	Minor / Informative
Location	BlockATMProxyPayout.sol#L58
Status	Unresolved

Description

The contract reserves fees based on the length of the receivers array, assuming that `1` token should be withheld for each receiver. However, this calculation neglects the relative value of the fee token. As a result, a single token may have excessive value, leading to potential over-estimation of fees.

```
function calcFee(uint256 length,ICustomizeERC20 erc20) internal view returns
(uint256 feeAmount) {
uint256 decimals = erc20.decimals();
return length * (10**(decimals));
}

function getSendAmount(address[] calldata array,address tokenAddress) internal
view returns (uint256 length,uint256 feeAmount){
length = array.length;
feeAmount = calcFee(length, ICustomizeERC20(tokenAddress));
return (length,feeAmount);
}
```

Recommendation

It is advisable to consider the relative value of the fee token when reserving fees. Alternatively, the contract could reserve fees as a reasonable portion of the transferred amount, ensuring that the fees are proportional to the value being processed and preventing excessive reservations.

UPA - Unchecked Payout Amount

Criticality	Minor / Informative
Location	BlockATMPayout.sol#L115
Status	Unresolved

Description

The `BlockATMProxyPayout.sol` contract implements the `payoutToken` function. Through this function, the contract receives tokens specified by the `payout` structure and forwards tokens as indicated by the `amount` array. However, the contract does not verify that the total of the amounts sent is covered by the incoming balance. As a result, the contract may transfer tokens from its own balance.

```
function _processBatchPayments(Payout calldata payout, address[] calldata
receivers, uint256[] calldata amounts, uint256 length) private {
for (uint256 i = 0; i < length; ) {
super.withdrawCommon(payout.safe, payout.tokenAddress, receivers[i],
amounts[i]);
unchecked { ++i; }
}
}
```

Recommendation

It is advisable to implement a check to ensure that the total of the amounts being sent is covered by the incoming balance before executing the token transfers. This will prevent the contract from transferring tokens from its own balance, thereby enhancing the security and integrity of the payout process.

UPT - Unchecked Payout Transfer

Criticality	Minor / Informative
Location	BlockATMProxyPayout.sol#L85,98
Status	Unresolved

Description

The contract implements the `payoutByContract` and `payoutByProxy` functions. In both cases, the user provides a `payoutAddress`, which the contract calls to execute the `safeTransferToProxy` method. If this address is a malicious contract, it may not transfer any value. However, the contract does not verify whether it received any tokens and proceeds to transfer the requested amount. This could lead to an inconsistency between the actual balance and the transferred amount.

```
if (payout.safe) {
IBlockATMPayout(from).safeTransferToProxy(payout.tokenAddress,
all);
} else {
IBlockATMPayout(from).transferToProxy(payout.tokenAddress, all);
}
```

Recommendation

It is advisable to implement the necessary checks to verify that the contract has received the expected tokens before proceeding with the transfer of the requested amount. This way, the contract can prevent inconsistencies between the actual balance and the transferred amount, ensuring the integrity of the payout process and protecting against potential malicious contracts.

US - Unchecked SubType

Criticality	Minor / Informative
Location	BlockFee.sol#L46
Status	Unresolved

Description

The contract implements a `subType` variable passed as an argument. However the contract does not verify the provided argument is a valid value.

```
function subFee(bool safe,address tokenAddress,address from,uint256
amount,uint256 id,uint256 subType) onlyBlockUser(from) public returns (bool) {
subFeeCommon(safe,tokenAddress, from, amount,id,subType);
return true;
}
```

Recommendation

It is advisable to validate all variables from the proper shape to ensure consistency of operations according to the intended design of the contract.

UTF - Unchecked Transfer Flag

Criticality	Minor / Informative
Location	BlockCommon.sol#L40
Status	Unresolved

Description

The `withdrawCommon` function includes a boolean flag to control the execution of either a transfer or a `safeTransfer` method. `withdrawCommon` is called by the `subFee` method, where the respective flag is passed. The `safeTransfer` method is intended for use with ERC20 tokens that do not return a boolean value on transfer, such as some stablecoins. However, the `subFee` function does not verify that the correct flag is set when such a token address is provided. As a result, inconsistencies in the transfer may occur.

```
function withdrawCommon(bool flag,address tokenAddress,address
withdrawAddress,uint256 amount) internal checkAmount(amount)
checkTokenAddress(tokenAddress) checkWithdrawAddress(withdrawAddress) {
IERC20 erc20 = IERC20(tokenAddress);
uint256 balance = erc20.balanceOf(address(this));
require(balance >= amount, "Insufficient balance");
if(flag){
erc20.safeTransfer(withdrawAddress, amount);
} else {
erc20.transfer(withdrawAddress, amount);
}
}
```

Recommendation

It is advisable to implement the proper checks before the transfer method is invoked, preventing inconsistencies in the transfer process and enhancing the reliability of the contract.

UUA - Unsanitized User Arguments

Criticality	Minor / Informative
Location	BlockATMCollect.sol#L50
Status	Unresolved

Description

The contract processes variables that may not form the proper shape. In particular, the array and newOwnerList may be empty arrays or contain zero values. The lack of checks can lead to inconsistencies.

```
function bindingRelationship(bool safe,uint256 id,address[] calldata
array,address[] calldata newOwnerList,address customerAddress) public returns
(bool){
...
}
```

Recommendation

The contract should implement the necessary checks to ensure that variables form the proper shape, thereby ensuring the consistency of operations.

UWA - Unsanitized Withdrawal Amount

Criticality	Minor / Informative
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. Specifically, the contract does not verify that the amount to be withdrawn is non-zero and less than the contract balance, including potential fees.

```
uint256 receiveAmount = info.amount;
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	BlockATMProxyPayout.sol#L75,80
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
function payoutByContract(Payout calldata payout,address
payoutAddress,string[] calldata orderNo,address[] calldata array,uint256[]
calldata amount) public onlyFinancials(payoutAddress) returns (bool){
payoutToken(payout, payoutAddress, 1,orderNo, array, amount,0);
return true;
}

function payoutByProxy(Payout calldata payout,address payoutAddress,string[]
calldata orderNo,address[] calldata array,uint256[] calldata amount) public
onlyUser() returns (bool){
payoutToken(payout, payoutAddress, 3, orderNo, array, amount,0);
return true;
}
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	BaseCustomer.sol#L8,10
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
address public feeGateway
address public owner
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	BlockCommon.sol#L13,22,30,40 BaseCustomer.sol#L28
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).

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.

```
function transferFrom(address tokenAddress,address from,address to,uint256
amount) internal checkTokenAddress(tokenAddress) returns(uint256) {
        if (amount > 0){
            IERC20 erc20 = IERC20(tokenAddress);
            erc20.safeTransferFrom(from, to, amount);
        }
        return amount;
    }

function transferFrom(address tokenAddress,address from,address to) internal
checkTokenAddress(tokenAddress) returns(uint256) {
        IERC20 erc20 = IERC20(tokenAddress);
        uint256 beforeAmount = erc20.balanceOf(from);
        return transferFrom(tokenAddress,from,to,beforeAmount);
    }

...
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	BlockFee.sol#L24,25 BlockATMProxyPayout.sol#L22 BlockATMPayout.sol#L13,17 BlockATMCustomer.sol#L37 BlockATMCollect.sol#L33
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.

```
feeReceiverAddress = newFeeReceiverAddress
feePaymentTokenAddress = newFeePaymentTokenAddress
feeGateway = newFeeGateway
proxyPayoutAddress = newProxyPayoutAddress
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	BlockCommon.sol#L47
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.

```
erc20.transfer(withdrawAddress, amount)
```

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**
IBlockFee	Interface			
	isSupportedFeeToken	External		-
	isStableCoin	External		-
	subFee	External	✓	-
	subFee	External	✓	-
	feeReceiverAddress	External	✓	-
IBlockATMPayout	Interface			
	transferToProxy	External	✓	-
	safeTransferToProxy	External	✓	-
	getOwnerAddressFlag	External	✓	-
IBlockATMCustomer	Interface			
	getOwnerAddressFlag	External	✓	-
BlockUtils	Implementation			
BlockFee	Implementation	Ownable, BlockCommon		

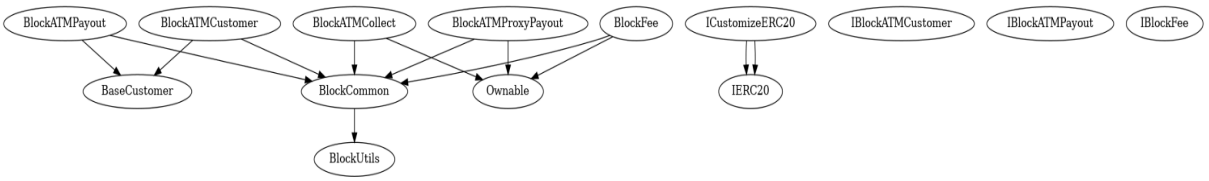
		Public	✓	-
	subFee	Public	✓	onlyBlockUser
	subFeeCommon	Internal	✓	
	subFee	Public	✓	onlyBlockUser
	setFeeAddress	Public	✓	onlyOwner
	setFeeTokenAddress	Public	✓	onlyOwner
	setFeeAmount	Public	✓	onlyOwner
	getSupportedFeeTokens	Public		-
	addSupportedFeeToken	External	✓	onlyOwner
	_addSupportedFeeToken	Internal	✓	
	removeSupportedFeeToken	External	✓	onlyOwner
	addStableCoin	External	✓	onlyOwner
	removeStableCoin	External	✓	onlyOwner
BlockCommon	Implementation	BlockUtils		
	transferFrom	Internal	✓	checkTokenAddress
	transferFrom	Internal	✓	checkTokenAddress
	transferCommon	Internal	✓	checkTokenAddress checkAmount
	withdrawCommon	Internal	✓	checkAmount checkTokenAddress checkWithdrawAddress
ICustomizeERC20	Interface	IERC20		
	decimals	External		-

BlockATMProxyPayout	Implementation	Ownable, BlockCommon		
		Public	✓	-
	addUser	Public	✓	onlyOwner
	deleteUser	Public	✓	onlyOwner
	calcFee	Internal		
	getSendAmount	Internal		
	payoutByWallet	Public	✓	-
	payoutByContract	Public	✓	onlyFinancials
	payoutByProxy	Public	✓	onlyUser
	_transferStableTokens	Private	✓	
	_transferTokens	Private	✓	
	_processBatchPayments	Private	✓	
	_sendPayoutFee	Private	✓	
	payoutToken	Internal	✓	
BlockATMPayout	Implementation	BlockCommon, BaseCustomer		
		Public	✓	-
	safeTransferToProxy	Public	✓	onlyProxy
	transferToProxy	Public	✓	onlyProxy
ICustomizeERC20	Interface	IERC20		
	decimals	External		-

BlockATMCustomer	Implementation	BlockCommon, BaseCustomer		
		Public	✓	-
	depositToken	Public	✓	checkTokenAddress
	calcFee	Internal		
	withdrawToken	Public	✓	onlyFinance
	getWithdrawAddressList	Public		-
	getWithdrawAddressFlag	Public		-
BlockATMCollect	Implementation	Ownable, BlockCommon		
		Public	✓	-
	bindingRelationship	Public	✓	-
	addRelationship	Public	✓	onlyOwner
	recoverRelationship	Public	✓	onlyOwner
	deleteRelationship	Public	✓	onlyUserOwner
	deleteRelationshipCommon	Internal	✓	
	calcFee	Internal		
	transferToken	Internal	✓	
	withdrawToken	Internal	✓	
	collect	Public	✓	onlyFinancials
	getOwnerUserFlag	Public		-
	getOwnerUserList	Public		-
	getOwner	Public		-

BaseCustomer	Implementation			
	processList	Internal	✓	
	burn	Public	✓	onlyOwner
	getOwnerAddressFlag	Public		-
	getOwnerAddressList	Public		-
	getBurnFlag	Public		-

Inheritance Graph



Summary

BlockATM V2 contract implements a utility and financial mechanism. This audit investigates security issues, business logic concerns, and potential improvements. Throughout the audit, a number of high and moderate critical issues were identified. The team is strongly advised to take these findings into consideration to improve the security and consistency of the protocol.

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