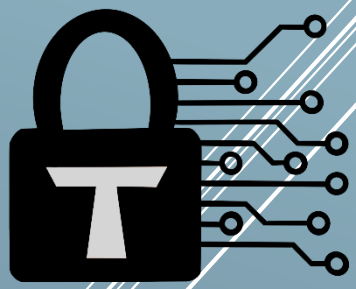
Trust Security

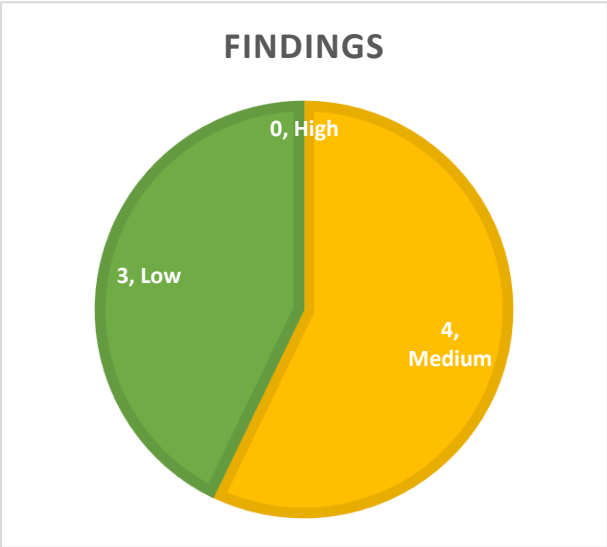


Smart Contract Audit

Boto SmartProxy

06/06/23

Executive summary

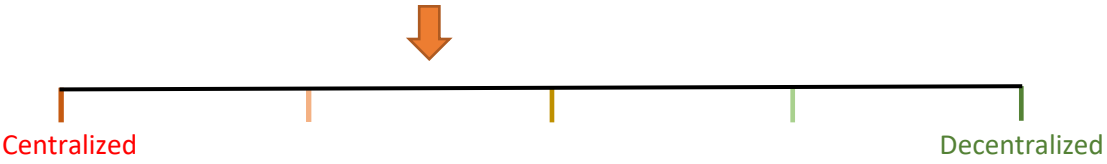


Category	Automation
Audited file count	2
Lines of Code	273
Auditor	Trust
Time period	01-06/06/23

Findings

Severity	Total	Open	Fixed	Acknowledged
High	0	-	-	-
Medium	4	-	4	-
Low	3	-	2	1

Centralization score



Signature

Digitally signed by Trust
Date: 2023-08-15 19:27:02

Document properties

Versioning

Version	Date	Description
0.1	06/06/23	Client report
0.2	13/08/23	Mitigation review
0.3	15/08/23	Mitigation review #2

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- BotoSmartProxy.sol
- dependencies/AllowedOperations.sol

Repository details

- **Repository URL:** https://github.com/botoapp/smart_actions
- **Commit hash:** 87d98492fc86435b4405a7a0772975f23dff0149
- **Mitigation review hash:** fd987c6da3d899d8e6c762bb82955c6df2640e6a
- **Mitigation 2 review hash:** f43d83bc2d2a37f8065fd862da9c41af6d26cb53

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Excellent**	Project has kept code as simple as possible, reducing attack risks
Documentation	**Excellent**	Project is very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	Project introduces some concerning centralization risks.

Findings

Medium severity findings

TRST-M-1 Deployer can backdoor the SmartProxy with another DEFAULT_ADMIN

- **Category:** Initialization flaws
- **Source:** BotoSmartProxy.sol
- **Status:** Fixed

Description

During construction, the deployer is given the privileged roles.

```
constructor() EIP712("BotoProxy", "1.0.0") {
    _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _setupRole(KEEPER_ROLE, msg.sender);
    _setupRole(SUPER_OPERATOR_ROLE, msg.sender);
}
```

They are then required to select the permanent owner and call *initialize()*:

```
function initialize(
    address newOwner
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(!_isInitialized, "BotoSmartProxy: already initialized");
    _isInitialized = true;
    // Ownership transfer
    _setupRole(DEFAULT_ADMIN_ROLE, newOwner);
    _setupRole(KEEPER_ROLE, newOwner);
    _setupRole(SUPER_OPERATOR_ROLE, newOwner);
    _revokeRole(SUPER_OPERATOR_ROLE, msg.sender);
    _revokeRole(KEEPER_ROLE, msg.sender);
    _revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);
    emit Initialized();
}
```

The function revokes the old roles from the deployer. However, if the deployer has since granted additional users any role, such as DEFAULT_ADMIN, those will remain in place. Therefore, the platform implicitly trusts the deployer whereas the design is for only the new owner to be trusted.

Recommended mitigation

Introduce the DEPLOYER_ROLE. DEFAULT_ADMIN_ROLE should only be unlocked in *initialize()*, which shall only be callable by the DEPLOYER_ROLE.

Team response

Fixed.

Mitigation review

Suggestion has been implemented successfully.

TRST-M-2 Attacker can re-use victim's signatures to allow operations at any time

- **Category:** Signature malleability issues
- **Source:** BotoSmartProxy.sol
- **Status:** Fixed

Description

In order to allow user-specific functionality, the following structure of allowed operations is signed by the user.

```
struct Operation {
    address contractAddress;
    bytes4 functionSelector;
}
struct Operations {
    Operation[] operations;
}
```

The hash signed is returned from the function below, which uses standard EIP712 encoding.

```
function _hashOperations(
    Operations memory _operations
) internal pure returns (bytes32, bytes32[] memory) {
    bytes32[] memory operationHashes = new bytes32[](
        _operations.operations.length
    );
    for (uint256 i = 0; i < _operations.operations.length; i++) {
        operationHashes[i] =
_hashOperation(_operations.operations[i]);
    }
    return (
        keccak256(
            abi.encode(
                OPERATIONS_SCHEMA_HASH,
                keccak256(abi.encodePacked(operationHashes))
            )
        ),
        operationHashes
    );
}
```

The user's signature is checked:

```
// Encode operations as EIP-712 typed data and compute the hash
(bytes32 hash, bytes32[] memory operationHashes) = _hashOperations(
    operations
);
// Verify the signature
address signer = ECDSA.recover(_hashTypedDataV4(hash), signature);
```

Notably, the user does not sign any field that guarantees that the signature cannot be replayed in a future *addAllowedOperationExtendedScope()* call. In fact, the signature can be sent by

other users to the frontend, to re-add operations. The user may not intend for them to be called at that point, if they disallowed the operations at a later point in time.

Recommended mitigation

The signed structure should contain a nonce and a timestamp. When processing a signature, mark the hash as used, to make it not repayable.

Team response

Fixed.

Mitigation review

Adding and removing operations now validate with the user's nonce. This means once a transaction has been executed, the signature can never be used again in the blockchain, fixing the issue.

TRST-M-3 A user can remove user's allowed operations, making the protocol unusable for them

- **Category:** Signature malleability issues
- **Source:** BotoSmartProxy.sol
- **Status:** Fixed

Description

The hashing and signature scheme has been described in TRST-M-2. Additionally, it has been observed that the same structure is used for *removeAllowedOperationExtendedScope()*.

```
// Encode operations as EIP-712 typed data and compute the hash
(bytes32 hash, bytes32[] memory operationHashes) = _hashOperations(
    operations
);
address signer = ECDSA.recover( _hashTypedDataV4(hash), signature);
```

This introduces a cross-function signature replay attack. The signature from the *add()* operation can be used immediately for the *remove()* operation, disallowing the desired access.

Recommended mitigation

The signed struct should have a boolean value, whether to add or remove this operation.

Team response

Fixed.

Mitigation review

The fix introduced a nonce per user, which means the signature can only be used once, fixing the issue.

TRST-M-4 An attacker can prevent users from adding or removing allowed operations indefinitely

- **Category:** Signature malleability issues
- **Source:** BotoSmartProxy.sol
- **Status:** Fixed

Description

After the nonce introduction in the audit fix commit, addition and removal from extended scope checks the user's nonce.

```
(
    bytes32 hash,
    bytes32[] memory operationHashes
) = _hashOperationsWithNonce(operations, _nonces[user]);
// (
//     bytes32 hash,
//     bytes32[] memory operationHashes
// ) = _hashOperations(operations);
// Verify the signature
address signer = ECDSA.recover(_hashTypedDataV4(hash), signature);
require(signer == user, "Signer does not match user");
```

This stops signature re-use, but does not prevent signature frontrunning attacks. An attacker can inspect the mempool to find a TX that uses the nonce, and replicate it (Sender does not need to be signer in the architecture). Note that the original transaction will revert, because the nonce will have been advanced.

Typically, it wouldn't be a major issue, as the user's intention would be fulfilled, add/remove by a request of a different party. However, the attacker can copy the signature and use it with the opposite function. For example, they may see a "remove operation" request and send an "add operation" request, with the same signature. Note that both functions accept the same Operation[] array and construct the signed hash identically. Indeed, the frontrunning transaction would execute successfully, because adding an already-added operation (or vice-versa) is permitted in the AllowedOperations contract.

```
function _addAllowedOperation(
    bytes32 operationHash,
    Operation memory operation
) internal {
    if (!allowedOperations[operationHash]) {
        allowedOperations[operationHash] = true;
        emit OperationAdded(operation);
    }
}
```

Therefore, an attacker that is snooping on the public mempool can deny user requests from ever being fulfilled. If user is trying to disallow a sensitive operation, there is now an opportunity for it to be abused.

Recommended mitigation

The root cause of the issue is that operations with a different semantic meaning (i.e. add/remove) have the same structure. We recommend a unique identifier to be used for each intention.

Team response

Fixed.

Mitigation review

An "intention" string was added to the Operation structures. It is different for addition and removal of operations. A malicious user that copies a signature from the mempool will only be able to perform the same intent. Therefore, the original transaction could revert but the intention will be fulfilled. This can still be seen as a UI inconvenience, but is absolutely safe from a data integrity perspective.

Low severity findings

TRST-L-1 No separation of pause and unpause privileges

- **Category:** Access control issues
- **Source:** BotoSmartProxy.sol
- **Status:** Acknowledged

Description

By design, the KEEPER role is able to pause and unpause the SmartProxy.

```
function pause() external onlyRole(KEEPER_ROLE) {
    _pause();
}
function unpause() external onlyRole(KEEPER_ROLE) whenNotShutdown {
    _unpause();
}
```

Logically, unpausing is a much more sensitive operation, as if the issue which causes pausing has not been dealt with, the project may face serious risks. Additionally, it is not a time-critical action contrary to pausing.

Recommended mitigation

Introduce a new role, or make DEFAULT_ADMIN be required for unpausing.

Team response

Acknowledged. Due to time-sensitivity of both *pause()* and *unpause(),* the team has decided to keep both under the same role.

TRST-L-2 Incorrect emission of events when executing with executeWithSuperOperator()

- **Category:** Event-related issues
- **Source:** BotoSmartProxy.sol
- **Status:** Fixed

Description

The *Executed* event is emitted in the *execute()* and *executeWithSuperOperator()* functions.

```
/// @notice Emitted when Operations are executed.
/// @param operations the operations executed
/// @param functionCallData the data of the function calls executed
/// @param user the user on behalf of whom the operations were
executed
event Executed(
    Operation[] operations,
    bytes[] functionCallData,
    address user
);
```

In *executeWithSuperOperator()*, the **user** parameter is set to the operator itself.

```
emit Executed(operations, functionCallData, msg.sender);
```

As all operations should be on behalf of a certain user, the event is misleading.

Recommended mitigation

Add a *SuperExecuted* event for SuperOperator executions, without a **user** parameter.

Team response

Fixed.

Mitigation review

Fix applied correctly.

TRST-L-3 If the deployer of SmartProxy is designated to be the owner, the contract will be unusable

- **Category:** Initialization flaws
- **Source:** BotoSmartProxy.sol
- **Status:** Fixed

Description

The *initialize()* function never checks that the **newOwner** is not the deployer.

```
function initialize(
    address newOwner
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(!_isInitialized, "BotoSmartProxy: already initialized");
    _isInitialized = true;
```

```
    // Ownership transfer
    _setupRole(DEFAULT_ADMIN_ROLE, newOwner);
    _setupRole(KEEPER_ROLE, newOwner);
    _setupRole(SUPER_OPERATOR_ROLE, newOwner);
    _revokeRole(SUPER_OPERATOR_ROLE, msg.sender);
    _revokeRole(KEEPER_ROLE, msg.sender);
    _revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);
    emit Initialized();
}
```

If this is the case, the contract will revoke the deployer's roles after granting them. Therefore, the contract will not be maintainable.

Recommended mitigation

If the contract wishes to keep the functionality of **newOwner == deployer**, do not revoke the roles. Otherwise, verify that the comparison is false.

Team response

Fixed

Mitigation review

The issue has been resolved as the *initialize()* code was refactored.

Additional recommendations

State changes should emit an event

The function below changes a security-critical state variable:

```
function toggleUnsafeAllowAll()
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
    whenNotPaused
{
    _unsafeAllowAll = !_unsafeAllowAll;
}
```

We recommend to emit an event for the sake of transparency.

Expose visibility to important state variables

Some important variables are listed below:

```
bool private _unsafeAllowAll = false;
bool private _isShutdown = false;
bool private _isInitialized = false;
```

They are marked private and do not have an accompanying getter function, so their value cannot be retrieved easily by a user.

Misleading fallback function

The fallback function supposedly disables transferring native tokens to the contract.

```
fallback() external {
    revert("This contract does not accept Ether transfers.");
}
```

However, the fallback function is not marked as payable. Therefore, this code will only be reached when calling the contract without a **msg.value**. There is no impact, because lack of a payable fallback (receive function) would make the contract revert when receiving value.

Missing NATSPEC documentation

The function below does not document the **user** parameter.

```
/// @notice Function to execute a set of operations.
/// Requirements:
/// - The caller must have the `EXECUTER_ROLE` role.
/// - The contract must not be paused.
/// @param operations The operations to execute.
/// @param arguments The arguments for each operation.
function execute(
    Operation[] memory operations,
    bytes[] memory arguments,
    address user
) external onlyRole(EXECUTOR_ROLE) whenNotPaused {
```

Centralization risks

Privileged roles can take over funds approved to SmartProxy

There are various roles that are trusted not to mishandle user's approved funds.

1. BASIC_SCOPE_MANAGER – Role is able to approve arbitrary operations for all users.
2. EXTENDED_SCOPE_MANAGER_ROLE – Role is able to approve arbitrary operations for a specific user. However, the fact a victim **user** cannot be passed to *execute()* does not necessary protect them, because what counts is the contract/selector/calldata actually invoked.
3. EXECUTOR – Role has complete control of the calldata passed to an approved contract/selector duo.
4. SUPER_OPERATOR – Role can call any function without any previous approvals.
5. DEFAULT_ADMIN_ROLE – Role can nominate any other role and therefore can perform all of the actions above.

Privileged actors can pause the contract

It should be noted that at any point an admin can pause the contract. Therefore, a contingency plan should be made be users in the event that the automatic action will not be executed by the platform.

Systemic risks

Off-chain risks

Various aspects of the Boto platform are performed off-chain. When executing calls on user's behalf, calldata is ultimately packaged by the platform. If the off-chain process allows users to have total control of calldata, a user may be able to bypass privilege boundaries and interact with assets of other users.

Similarly, an off-chain procedure determines which operations are available to be confirmed by the user for *extended scope* interactions. If that procedure allows users to interact with contract of other users, or interact mutual contracts in an unsafe way, it may be compromised to perform privileged actions.

We have recommended that the platform should sandbox user's approvals in a safer way, that can be fully verified on-chain.