Competitive Security Assessment

Magpie_CCIP

Jul 22nd, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

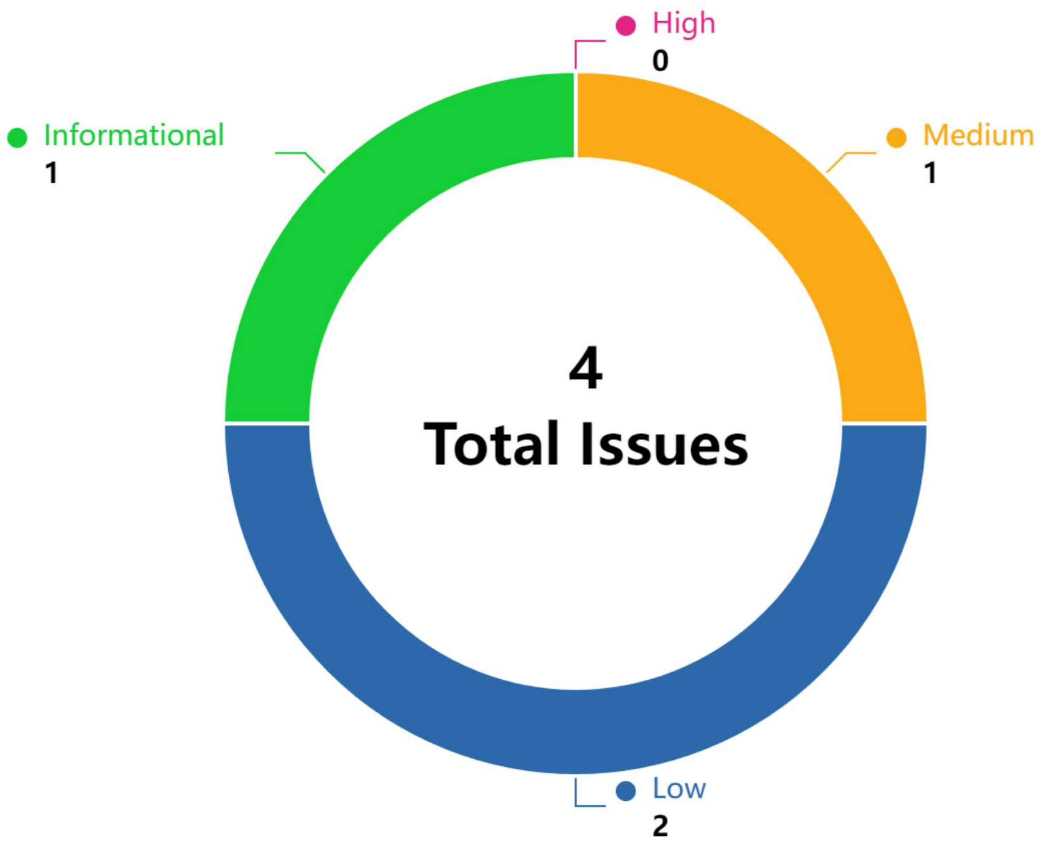
Overview

Project Name	Magpie_CCIP
Language	solidity
Codebase	• https://github.com/magpiexyz/radpie_contracts/ • audit version - c0e50725cb13757d85cea489fa429f3f2efa4ce2 • final version - ?

Audit Scope

File	SHA256 Hash

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
MCP-1	Refund excess funds or set up the withdraw mechanism	Logical	Medium	Fixed	***
MCP-2	Using `burn` instead of `burnFrom` in `RadpieOFTBridge::_debitFrom()`	Logical	Low	Acknowledged	***
MCP-3	Lack whitelist Whether the radpie token check is supported in `RadpieCCIPBridge:: tokenTransfer` function	Logical	Low	Acknowledged	***
MCP-4	Gas Optimization	Code Style	Informational	Fixed	***

MCP-1:Refund excess funds or set up the withdraw mechanism

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/contracts/crosschain/RadpieCCIPBridge.sol#L113-L141
- code/contracts/crosschain/RadpieCCIPBridge.sol#L113-151
- code/contracts/crosschain/RadpieCCIPBridge.sol#L113-L151
- code/contracts/crosschain/RadpieCCIPBridge.sol#L151

```
113: function tokenTransfer(
114:         uint64 destinationChainSelector,
115:         address _receiver,
116:         uint256 _amount
117:     ) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
118:         if (_receiver == address(0)) revert InvalidAddress();
119:
120:         if (_amount == 0 || msg.value == 0) revert InvalidAmount();
121:
122:         IERC20(radpie).safeTransferFrom(msg.sender, address(this), _amount);
123:         IERC20(radpie).safeIncreaseAllowance(chainlinkRouter, _amount);
124:
125:         (Client.EVM2AnyMessage memory evm2AnyMessage, uint256 fee) = _estimateGasFee(
126:             destinationChainSelector,
127:             _receiver,
128:             radpie,
129:             _amount,
130:             address(0)
131:         );
132:
```

```
133:         if (fee > msg.value) revert NotEnoughBalance(msg.value, fee);
134:
135:         bytes32 messageId;
136:
137:         messageId = IRouterClient(chainlinkRouter).ccipSend{ value: fee }(
138:             destinationChainSelector,
139:             evm2AnyMessage
140:         );
```

```
function tokenTransfer(
        uint64 destinationChainSelector,
        address _receiver,
        uint256 _amount
    ) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
        if (_receiver == address(0)) revert InvalidAddress();

        if (_amount == 0 || msg.value == 0) revert InvalidAmount();

        IERC20(radpie).safeTransferFrom(msg.sender, address(this), _amount);
        IERC20(radpie).safeIncreaseAllowance(chainlinkRouter, _amount);

        (Client.EVM2AnyMessage memory evm2AnyMessage, uint256 fee) = _estimateGasFee(
            destinationChainSelector,
            _receiver,
            radpie,
            _amount,
            address(0)
        );

        if (fee > msg.value) revert NotEnoughBalance(msg.value, fee);

        bytes32 messageId;

        messageId = IRouterClient(chainlinkRouter).ccipSend{ value: fee }(
            destinationChainSelector,
            evm2AnyMessage
        );

        emit TokensTransferred(
            messageId,
            destinationChainSelector,
            _receiver,
            radpie,
            _amount,
            address(0),
            fee
        );
    }
```

```
function tokenTransfer(
        uint64 destinationChainSelector,
        address _receiver,
        uint256 _amount
    ) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
        if (_receiver == address(0)) revert InvalidAddress();

        if (_amount == 0 || msg.value == 0) revert InvalidAmount();

        IERC20(radpie).safeTransferFrom(msg.sender, address(this), _amount);
        IERC20(radpie).safeIncreaseAllowance(chainlinkRouter, _amount);

        (Client.EVM2AnyMessage memory evm2AnyMessage, uint256 fee) = _estimateGasFee(
            destinationChainSelector,
            _receiver,
            radpie,
            _amount,
            address(0)
        );

```

```
        if (fee > msg.value) revert NotEnoughBalance(msg.value, fee);

        bytes32 messageId;

        messageId = IRouterClient(chainlinkRouter).ccipSend{ value: fee }(
            destinationChainSelector,
            evm2AnyMessage
        );

        emit TokensTransferred(
            messageId,
            destinationChainSelector,
            _receiver,
            radpie,
            _amount,
            address(0),
            fee
        );
    }
```

```
}
```

Description

*****:** If the sender mistakenly sends more Ether than required for the fee calculation, the tokenTransfer() function doesn't return this excess Ether. The code lacks a mechanism to refund or handle surplus Ether sent beyond the required fee. Without handling this excess Ether, it remains trapped within the contract, leading to a loss of funds for the sender.
*****:** When calling `RadpieCCIPBridge.tokenTransfer()` , if the user sends msg.value greater than the fee required for the transaction, the excess ether will remain in the contract and will not be returned. This may result in the funds being locked up in the contract.
*****:** Suppose a user sent more ETH to the RadpieCCIPBridge contract via tokenTransfer(). Currently there is no way to withdraw the excessive ETH so those ETH will be stuck in the contract forever.
*****:** The tokenTransfer function in the provided smart contract lacks a refund mechanism for excess funds when msg.value is greater than the calculated gas fee (fee). This can lead to a scenario where a portion of the user's

funds (i.e., msg.value - fee) is locked in the contract, and there is no provision for returning these excess funds to the user.

```
function tokenTransfer(
        uint64 destinationChainSelector,
        address _receiver,
        uint256 _amount
    ) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
        if (_receiver == address(0)) revert InvalidAddress();

        if (_amount == 0 || msg.value == 0) revert InvalidAmount();

        IERC20(radpie).safeTransferFrom(msg.sender, address(this), _amount);
        IERC20(radpie).safeIncreaseAllowance(chainlinkRouter, _amount);

        (Client.EVM2AnyMessage memory evm2AnyMessage, uint256 fee) = _estimateGasFee(
            destinationChainSelector,
            _receiver,
            radpie,
            _amount,
            address(0)
        );

        if (fee > msg.value) revert NotEnoughBalance(msg.value, fee);

        bytes32 messageId;

        messageId = IRouterClient(chainlinkRouter).ccipSend{ value: fee }(
            destinationChainSelector,
            evm2AnyMessage
        );

        emit TokensTransferred(
            messageId,
            destinationChainSelector,
            _receiver,
            radpie,
            _amount,
            address(0),
            fee
        );
    }
```

It's important to note that the lack of a refund mechanism might result in users losing funds unnecessarily. This issue could impact the overall user experience and trust in the contract.
*****:** The RadpieCCIPBridge contract accepts native tokens (like ETH in Ethereum) as msg.value in the tokenTransfer function. This design is intended to handle transaction fees for cross-chain transfers. However, the contract currently lacks a mechanism to withdraw these native tokens from the contract. This absence poses a

risk of funds getting locked within the contract without a secure way for the owner or an authorized entity to retrieve them.

Recommendation

*****: If the user sends too much ether, return the excess amount back
*****: To solve this problem and improve the security and user-friendliness of the contract, consider returning excess funds:
Calculate the difference between msg.value - fee and return this part of the funds to msg.sender. This can be achieved with simple `payable(msg.sender).transfer(msg.value - fee)`.

```
if (msg.value > fee) {
    payable(msg.sender).transfer(msg.value - fee);
}
```

*****: Compute excess ETH sent to the contract:

```
msg.value - actualCost
```

and store this value inside a local accounting system. Implement a separate withdraw() function to let users withdraw themselves. This follows the pull over push pattern.
*****: To fix this issue, it is recommended to implement a refund mechanism to return any excess funds to the sender. Here's a suggested modification:

```solidity
// Add a state variable to track excess funds
uint256 private excessFunds;

// ...

function tokenTransfer(
    uint64 destinationChainSelector,
    address _receiver,
    uint256 _amount
) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
    // ... (existing code)



    bytes32 messageId;

    messageId = IRouterClient(chainlinkRouter).ccipSend{ value: fee }(
        destinationChainSelector,
        evm2AnyMessage
    );

 if (0 > msg.value - fee ) {
        // Calculate excess funds
        excessFunds = msg.value - fee;
        // Refund excess funds to the sender
        payable(msg.sender).transfer(excessFunds);
    }



    emit TokensTransferred(
        messageId,
        destinationChainSelector,
        _receiver,
        radpie,
        _amount,
        address(0),
        fee
    );
}
```

This modification ensures that any excess funds are refunded to the sender before the main transaction proceeds. Additionally, it is advisable to thoroughly test the modified code to ensure its correctness and security.
***: Add a secure withdrawal function that allows the contract owner or an authorized entity to withdraw the native tokens held by the contract

```
function withdraw(address payable recipient, uint256 amount) external onlyOwner {
    require(address(this).balance >= amount, "Insufficient balance");
    recipient.transfer(amount);
    emit Withdrawn(recipient, amount);
}
```

Client Response

Fixed,Fixed in this branch: https://github.com/magpiexyz/radpie_contracts/pull/90
Fixed,Fixed in this branch: https://github.com/magpiexyz/radpie_contracts/pull/90
Acknowledged,We intend to facilitate the direct transfer of user funds if they surpass the fee, as suggested Issue 2, thereby eliminating the necessity to implement this.
Fixed,Fixed in this branch: https://github.com/magpiexyz/radpie_contracts/pull/90
Acknowledged.We intend to facilitate the direct transfer of user funds if they surpass the fee, as suggested, thereby eliminating the necessity to implement this.

MCP-2:Using `burn` instead of `burnFrom` in `RadpieOFTBridge::_debitFrom()`

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/contracts/crosschain/RadpieOFTBridge.sol#L47-L56
- code/contracts/crosschain/RadpieOFTBridge.sol#L54

```
47: function _debitFrom(
48:         address _from,
49:         uint16,
50:         bytes memory,
51:         uint _amount
52:     ) internal virtual override returns (uint) {
53:         require(_from == _msgSender(), "IndirectOFT: owner is not send caller");
54:         RADPIE.burnFrom(_from, _amount);
55:         return _amount;
56:     }
```

```
54: RADPIE.burnFrom(_from, _amount);
```

Description

***: After checking that _from and _msgSender() must be equal in the _debitFrom function, should design the burn function that calls RADPIE instead of the burnFrom function, which not only reduces gas consumption, but also ensures that the transaction does not fail due to unauthorized.
***: RadpieOFTBridge._debitFrom() uses burnFrom() from OpenZeppelin ERC20BurnableUpgradeable.sol:

```
    function _debitFrom(
        address _from,
        uint16,
        bytes memory,
        uint _amount
    ) internal virtual override returns (uint) {
        require(_from == _msgSender(), "IndirectOFT: owner is not send caller");
        RADPIE.burnFrom(_from, _amount);
        return _amount;
    }
```

```
    function burnFrom(address account, uint256 value) public virtual {
        _spendAllowance(account, _msgSender(), value);
        _burn(account, value);
    }
```

burnFrom() computes allowance. In other words, msg.sender has to approve himself/herself to get allowance before burning. This is an anti-pattern and it could lead to mistakes.

Recommendation

***: Use RADPIE's burn function in the _debitFrom function.
***: Use burn() instead of burnFrom():

```
RADPIE.burnFrom(_amount);
```

Client Response

Acknowledged,The issue is acknowledged as valid; however, we have decided not to address it.
The reason being, if we were to fix this issue, it would necessitate the addition of a
burn role in the RADPIE contract and potentially require more changes in
the implementation. Given that we are transitioning to CCIP for token transfers within
a month, our plan is to revoke the mint role from the OFTbridge contract
and exclusively utilize CCIP. This strategic shift leads us to believe that the issue will
no longer be applicable in our future implementation.
Acknowledged,The issue is acknowledged as valid; however, we have decided not to address it.
The reason being, if we were to fix this issue, it would necessitate the addition of a
burn role in the RADPIE contract and potentially require more changes in
the implementation. Given that we are transitioning to CCIP for token transfers within
a month, our plan is to revoke the mint role from the OFTbridge contract
and exclusively utilize CCIP. This strategic shift leads us to believe that the issue will
no longer be applicable in our future implementation.

MCP-3:Lack whitelist Whether the radpie token check is supported in `RadpieCCIPBridge:: tokenTransfer` function

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/contracts/crosschain/RadpieCCIPBridge.sol#L113-L151

```
113: function tokenTransfer(
114:         uint64 destinationChainSelector,
115:         address _receiver,
116:         uint256 _amount
117:     ) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
118:         if (_receiver == address(0)) revert InvalidAddress();
119:
120:         if (_amount == 0 || msg.value == 0) revert InvalidAmount();
121:
122:         IERC20(radpie).safeTransferFrom(msg.sender, address(this), _amount);
123:         IERC20(radpie).safeIncreaseAllowance(chainlinkRouter, _amount);
124:
125:         (Client.EVM2AnyMessage memory evm2AnyMessage, uint256 fee) = _estimateGasFee(
126:             destinationChainSelector,
127:             _receiver,
128:             radpie,
129:             _amount,
130:             address(0)
131:         );
132:
```

```
133:         if (fee > msg.value) revert NotEnoughBalance(msg.value, fee);
134:
135:         bytes32 messageId;
136:
137:         messageId = IRouterClient(chainlinkRouter).ccipSend{ value: fee }(
138:             destinationChainSelector,
139:             evm2AnyMessage
140:         );
141:
142:         emit TokensTransferred(
143:             messageId,
144:             destinationChainSelector,
145:             _receiver,
146:             radpie,
147:             _amount,
148:             address(0),
149:             fee
150:         );
151:     }
```

Description

*****:** There is a logical loophole in the tokenTransfer function. This function allows users to transfer radpie tokens to addresses on other chains. However, this function does not verify that the destination chain (represented by the destinationChainSelector) supports radpie tokens until the tokens are actually sent. If the target chain does not support radpie tokens, this will result in funds being locked on the chain as there is no mechanism in place to recover or redirect those funds.

Recommendation

*****: Add a check to the tokenTransfer function to ensure that the target chain supports the tokens being transferred.

```
function tokenTransfer(
    uint64 destinationChainSelector,
    address _receiver,
    uint256 _amount
) external payable nonReentrant whenNotPaused onlyWhitelistedChain(destinationChainSelector) {
    if (_receiver == address(0)) revert InvalidAddress();
    if (_amount == 0 || msg.value == 0) revert InvalidAmount();

    address[] memory supportedTokens = getSupportedTokens(destinationChainSelector);
    bool isTokenSupported = false;
    for (uint i = 0; i < supportedTokens.length; i++) {
        if (supportedTokens[i] == radpie) {
            isTokenSupported = true;
            break;
        }
    }
    if (!isTokenSupported) revert InvalidToken();

    IERC20(radpie).safeTransferFrom(msg.sender, address(this), _amount);
    IERC20(radpie).safeIncreaseAllowance(chainlinkRouter, _amount);
    // ...
}
```

Client Response

Acknowledged,We have a onlyWhitelistedChain(destinationChainSelector) modifier that verifies the destination chain before transferring tokens. This destination is added by the contract owner, and we will utilize a multisig wallet to add supported destination chains for our Radpie token.

MCP-4:Gas Optimization

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/contracts/crosschain/RadpieCCIPBridge.sol#L24
- code/contracts/crosschain/RadpieCCIPBridge.sol#L51
- code/contracts/crosschain/RadpieCCIPBridge.sol#L52

```
24: address public chainlinkRouter;
```

```
51: error AlreadyRegsitered();
```

```
52: error InvalidToken();
```

Description

***: Unused code should be deleted.
***: The chainlinkRouter address is set during the initialization phase and does not appear to change throughout the contract's lifecycle. Since this address remains constant, it is a suitable candidate for being declared as an immutable variable

Recommendation

***: Delete unused code.
***: Add immutable variable for `chainlinkRouter` :

```
address public immutable chainlinkRouter;
```

Once set during contract deployment, the chainlinkRouter address will remain constant and cannot be altered.

Client Response

Fixed,Fixed in this Branch: https://github.com/magpiexyz/radpie_contracts/pull/90
Acknowledged,We are unable to make the Chainlink Router immutable because Chainlink may upgrade its CCIP implementation in the future.

Disclaimer