10AP17
Auditss

Chakra Audit

Prepared by: 10ap17

Contents

1 Executive Summary 2
1.1 Overview . . . 2
1.2 Audit Scope . . . 2
1.3 Summary of Findings . . . 3

2 Detailed Findings 4
2.1 [High] Duplicate Validator signatures bypass multi-signature validation . . . 4
2.1.1 Vulnerability Detail . . . 4
2.1.2 Proof of Concept . . . 5
2.1.3 Impact . . . 7
2.1.4 Recommendations . . . 8
2.2 [Low] `receive_cross_chain_callback` function always returns true . . . 9
2.2.1 Vulnerability Detail . . . 9
2.2.2 Impact . . . 10
2.2.3 Recommendations . . . 10
2.3 [Low] Incorrect parameter order in `CrossChainHandleResult` event emission . . . 11
2.3.1 Vulnerability Detail . . . 11
2.3.2 Impact . . . 11
2.3.3 Recommendations . . . 11
2.4 [Info] Redundant Check in `receive_cross_chain_msg` . . . 12
2.4.1 Vulnerability Detail . . . 12
2.4.2 Impact . . . 12
2.4.3 Recommendations . . . 12
2.5 [Info] Typo in `deocde_transfer()` function name . . . 13
2.5.1 Vulnerability Detail . . . 13
2.5.2 Impact . . . 13
2.5.3 Recommendations . . . 13

3 Conclusion 14

1 Executive Summary

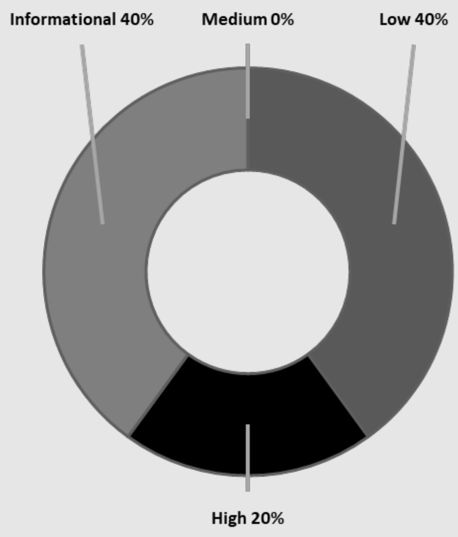
1.1 Overview

The Chakra Network is a modular settlement layer designed to unlock Bitcoin's liquidity across blockchain ecosystems. By acting as a shared settlement layer, Chakra enables execution layers like Ethereum and Base to focus on transaction execution without managing settlement complexities. This approach fosters interoperability and enhances economic security, creating a dynamic DeFi environment. The audit reviewed key components of the protocol, including contracts for cross-chain settlement, signature verification, and ERC20 token handling. Particular focus was given to ensuring secure transaction processing,and robust cross-chain message encoding.

1.2 Audit Scope

- solidity/handler/contracts/ChakraSettlementHandler.sol
- solidity/settlement/contracts/ChakraSettlement.sol
- solidity/settlement/contracts/SettlementSignatureVerifier.sol
- solidity/handler/contracts/SettlementSignatureVerifier.sol
- solidity/handler/contracts/BaseSettlementHandler.sol
- solidity/settlement/contracts/BaseSettlement.sol
- solidity/handler/contracts/libraries/MessageV1Codec.sol
- solidity/settlement/contracts/libraries/MessageV1Codec.sol
- solidity/handler/contracts/libraries/AddressCast.sol
- solidity/settlement/contracts/libraries/AddressCast.sol
- solidity/handler/contracts/ChakraToken.sol
- solidity/handler/contracts/ERC20CodecV1.sol
- solidity/handler/contracts/TokenRoles.sol
- solidity/handler/contracts/libraries/Message.sol
- solidity/settlement/contracts/libraries/Message.sol
- solidity/handler/contracts/libraries/ERC20Payload.sol
- cairo/handler/src/utils.cairo
- cairo/handler/src/settlement.cairo
- cairo/handler/src/handler_erc20.cairo
- cairo/handler/src/codec.cairo
- cairo/handler/src/ckr_btc.cairo
- cairo/handler/src/interfaces.cairo
- cairo/handler/src/constant.cairo
- cairo/handler/src/lib.cairo

1.3 Summary of Findings



	Issue Title	Severity
1	Duplicate Validator signatures bypass multi-signature validation	High
2	`receive_cross_chain_callback` function always returns true	Low
3	Incorrect parameter order in `CrossChainHandleResult` event emission	Low
4	Redundant check in `receive_cross_chain_msg`	Info
5	Typo in `deocde_transfer()` function name	Info

2 Detailed Findings

2.1 [High] Duplicate Validator signatures bypass multi-signature validation

2.1.1 Vulnerability Detail

The `verifyECDSA` function within the signature verification system fails to check if the signatures in the provided signature set are unique (not duplicates). As a result, the same validator's signature can be duplicated in the signature array, allowing it to bypass the minimum number of required validators.

The vulnerability arises because the `verifyECDSA` function verifies signatures in a loop, but does not check whether each signature is unique. Instead, it counts every valid signature, even if the same signature from a single `validator` is repeated. This behavior undermines the purpose of requiring multiple validators to sign a message for it to be considered valid.

```
function verify(
    bytes32 msgHash,
    bytes calldata signatures,
    uint8 sign_type
) external view returns (bool) {
    if (sign_type == 0) {
        return verifyECDSA(msgHash, signatures);
    } else {
        return false;
    }
}

function verifyECDSA(
    bytes32 msgHash,
    bytes calldata signatures
) internal view returns (bool) {
    require(
        signatures.length % 65 == 0,
        "Signature␣length␣must␣be␣a␣multiple␣of␣65"
    );

    uint256 len = signatures.length;
    uint256 m = 0;
    for (uint256 i = 0; i < len; i += 65) {
        bytes memory sig = signatures[i:i + 65];
        if (
            validators[msgHash.recover(sig)] && ++m >=
                required_validators
        ) {
            return true;
        }
    }

    return false;
}
```

2.1.2 Proof of Concept

In this test , we simulate a scenario with two validators (validator1 and validator2), both of whom sign the same message. The contract is configured to require signatures from 2 validators for successful verification (required_validators is 2).
Here's how the test works:
- We first generate valid signatures for the message from both validator1 and validator2.
- Instead of passing both valid signatures to the verify function, we intentionally pass two identical signatures from validator1.
- The verifyECDSA function will validate the duplicate signatures because it does not check whether the signatures are from different validators.
- As a result, the test will pass successfully, even though we only provided one valid signature repeated twice. This highlights the vulnerability, as the system incorrectly considers the message as having been signed by two validators.

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import "../src/SettlementSignatureVerifier.sol";

contract VulnerabilityTest is Test {

SettlementSignatureVerifier verifier;
address owner;
address validator1;
uint256 validator1PK;
address validator2;
```

```
uint256 validator2PK;

function setUp() public {
    // Initialize addresses
    owner = makeAddr("owner");

    (validator1, validator1PK) = makeAddrAndKey("validator1
       ");
    (validator2, validator2PK) = makeAddrAndKey("validator2
       ");

    // Deploy and initialize the
       SettlementSignatureVerifier contract
    verifier = new SettlementSignatureVerifier();
    verifier.initialize(owner, 2); // Set required
       validators to 2

    // Set Aleksa as manager and owner
    vm.prank(owner); // Simulate that Aleksa is calling the
        contract
    verifier.add_manager(owner);

    // Add validator1 and validator2
    vm.prank(owner);
    verifier.add_validator(validator1);
    vm.prank(owner);
    verifier.add_validator(validator2);
}

 function testDuplicateSignatureVulnerability() public {
    // Prepare message and signatures
    bytes32 msgHash = keccak256("Test␣message");

    // Simulate validator1 signing the message
    (uint8 v1, bytes32 r1, bytes32 s1)= vm.sign(
       validator1PK, msgHash);
    // Generate the signature
    bytes memory sigValidator1 = abi.encodePacked(r1, s1,
       v1);

    // Simulate validator2 singing the message
    (uint8 v2, bytes32 r2, bytes32 s2)= vm.sign(
       validator2PK, msgHash);
    // Generate the signature
    bytes memory sigValidator2 = abi.encodePacked(r2, s2,
       v2);

    // Test vulnerability: Duplicate the same signature and
        try to pass verification
    bytes memory duplicateSignatures = abi.encodePacked(
       sigValidator1, sigValidator1);

    bool duplicatePass = verifier.verify(msgHash,
       duplicateSignatures, 0);
    assertTrue(duplicatePass, "Duplicate␣signatures␣should␣
       pass");
}
}
```

When we run this test, the following output confirms the presence of the vulnerability, showing that the test passed despite only one distinct validator's signature being provided:

```
forge test
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Counter.t.sol:VulnerabilityTest
[PASS] testDuplicateSignatureVulnerability() (gas: 29499)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished
   in 32.95ms (10.44ms CPU time)

Ran 1 test suite in 112.34ms (32.95ms CPU time): 1 tests
   passed, 0 failed, 0 skipped (1 total tests)
```

This output validates that the vulnerability exists, as the duplicate signatures are improperly accepted, bypassing the multi-signature requirement.

2.1.3 Impact

This vulnerability significantly weakens the security of the multi-signature verification process. The minimum number of required validators is intended to provide additional security by ensuring that multiple validators have signed a transaction. However, due to this vulnerability, this security mechanism could be bypassed by submitting duplicate signatures from the same validator.

2.1.4 Recommendations

Consider implementing logic inside the `verifyECDSA` function to check whether there are duplicate signatures in the provided signatures array. This can be done by keeping track of the validators whose signatures have already been processed. This ensures that each validator's signature is only counted once, preventing the bypass of the minimum number of required validators.

2.2 [Low] `receive_cross_chain_callback` function always returns true

2.2.1 Vulnerability Detail

The function `receive_cross_chain_callback` of the `ChakraSettlementHandler` contract always returns `true`, even when the status of the cross-chain message is `Failed`. Although the current implementation of the `settlement/ChakraSettlement` contract does not rely heavily on this return value and does not present a major vulnerability, future implementations could introduce logic that depends on accurate success or failure signals from this function. In such cases, this behavior could lead to more severe issues,

```
function receive_cross_chain_callback(
    uint256 txid,
    string memory from_chain,
    uint256 from_handler,
    CrossChainMsgStatus status,
    uint8 sign_type,
    bytes calldata signatures
) external onlySettlement returns (bool) {
    if (is_valid_handler(from_chain, from_handler) == false)
        {
        return false;
    }

    require(create_cross_txs[txid].status ==
        CrossChainTxStatus.Pending, "Invalid␣
        CrossChainTxStatus");

    if (status == CrossChainMsgStatus.Success) {
        if (mode == SettlementMode.MintBurn) {
            _erc20_burn(address(this), create_cross_txs[txid
                ].amount);
        }
        create_cross_txs[txid].status = CrossChainTxStatus.
            Settled;
    }

    if (status == CrossChainMsgStatus.Failed) {
        create_cross_txs[txid].status = CrossChainTxStatus.
            Failed;
    }
    return true;  // This always returns true, even on
        failure.
}
```

This behavior is unintended, as indicated by the comments for the relevant function:

```
/**
* @dev Receives a cross-chain callback
* @param txid The transaction ID
* @param from_chain The source chain
* @param from_handler The source handler
* @param status The status of the cross-chain message
* @return bool True if successful, false otherwise
*/
```

2.2.2 Impact

No substantial impact on the current implementation.

2.2.3 Recommendations

The function should return `False` if the message status is `Failed` (`status == CrossChainMsgStatus.Failed`). This issue does not represent any major problem for the the current implementation of the `ChakraSettle-ement` contract, the contract which is using this function. Consider updating the return value to reflect the actual success or failure of the message.

2.3 [Low] Incorrect parameter order in `CrossChainHandleResult` event emission

2.3.1 Vulnerability Detail

The function `receive_cross_chain_msg` emits the `CrossChainHandleResult` event with the wrong order of parameters, mixing up `from_chain` and `to_chain`. This can lead to inaccurate data being emitted, potentially causing issues for off-chain services or systems relying on these events to track cross-chain transactions. In particular, the `from_chain` and `to_chain` values are switched in the emitted event.

The event is expected to emit the following parameters in this order:

```
event CrossChainHandleResult(
    uint256 indexed txid,
    CrossChainMsgStatus status,
    string from_chain,
    string to_chain,
    address from_handler,
    uint256 to_handler,
    PayloadType payload_type
);
```

However, it is currently emitting in the following incorrect order:

```
emit CrossChainHandleResult(
    txid,
    status,
    contract_chain_name, // Should be 'from_chain'
    from_chain,          // Should be 'to_chain` in this
        case `contract_chain_name`
    address(to_handler), // Should be 'from_handler'
    from_handler,        // Should be 'to_handler'
    payload_type
);
```

2.3.2 Impact

No substantial impact on the current implementation.

2.3.3 Recommendations

Correct the order of the parameters in the emit `CrossChainHandleResult` statement to align with the event definition.

2.4 [Info] Redundant Check in `receive_cross_chain_msg`

2.4.1 Vulnerability Detail

The function `receive_cross_chain_msg` has two lines that check the same condition `(payload_type == PayloadType.ERC20)`. The require statement ensures the `payload_type` is valid (`ERC20`), making the subsequent if statement redundant. This duplication results in unnecessary checks that may add confusion during code maintenance and auditing.

The redundant check can be seen in the following code of the `ChakraSettlementHandler` contract:

```
require(isValidPayloadType(payload_type), "Invalid␣payload␣
   type");

if (payload_type == PayloadType.ERC20) {  // Redundant
   check
   // Logic for handling ERC20 cross-chain transfers...
}
```
Where the `isValidPayloadType()` function logic is :
```solidity
function isValidPayloadType(
        PayloadType payload_type
   ) internal pure returns (bool) {
        return (payload_type == PayloadType.ERC20);
   }
```

The `require(isValidPayloadType(payload_type))` line ensures that `payload_type` is valid (`ERC20`), making the if check for `payload_type == PayloadType.ERC20` unnecessary.

2.4.2 Impact

No impact.

2.4.3 Recommendations

Consider removing one of the statements checking the `payload_type` since it is already validated two times. This will simplify the code and reduce redundancy.

2.5 [Info] Typo in `deocde_transfer()` function name

2.5.1 Vulnerability Detail

In the contract `ERC20CodecV1`, the function `deocde_transfer()` contains a typo in its name. This should be `decode_transfer()` to maintain consistency with standard naming conventions and prevent confusion during code use or maintenance. Misnaming functions can lead to harder-to-read code and potential integration issues.

The following mistyped function can be found bellow:

```
function deocde_transfer(
    bytes calldata _payload
) external pure returns (ERC20TransferPayload memory
    transferPayload) {
    // Decoding logic here...
}
```

2.5.2 Impact

No impact.

2.5.3 Recommendations

Rename the function to `decode_transfer()` for proper spelling and to adhere to common naming conventions, which will improve readability and maintainability.

3 Conclusion

The audit of the Chakra Network identified several vulnerabilities, including a high-severity issue in the `verifyECDSA` function. This function failed to validate the uniqueness of signatures, allowing duplicate validator signatures to bypass the multi-signature requirement, which could compromise the system's integrity.

Other findings included minor issues such as an always-true return in the `receive_cross_chain_callback` function and parameter misordering in an event emission. These issues were either resolved or noted for improvement, enhancing the system's consistency and resilience.

With these corrections and optimizations, the Chakra Network is better equipped to provide a secure and efficient settlement layer for cross-chain interactions. Continued vigilance and adherence to best practices are encouraged as the protocol evolves.