Naive Receiver Audit Report

Prepared by Ahmad Faraz

Auditing Protocol: `Naive Receiver`

Date: July 31, 2025

Contents

1 Disclaimer

I, Ahmad Faraz, makes every effort to understand the protocol and identify vulnerabilities within the given timeframe but holds no responsibility for missed issues. This audit is not an endorsement of the protocol's business logic or product and focuses solely on Solidity level vulnerabilities.

2 Risk Classification

2.1 Impact Table

Likelihood	Critical	High	Medium	Low
Critical	Critical	Critical	High	Medium
High	Critical	High	High/Medium	Medium
Medium	High	High/Medium	Medium	Medium/Low
Low	Medium	Medium	Medium/Low	Low

3 Audit Details

3.1 Scope

The audit covered the following files:

- `./src/`
- `BasicForwarder.sol`
- `FlashLoanReceiver.sol`
- `Multicall.sol`
- `NaiveReceiverPool.sol`

4 Protocol Summary

The Naive Receiver protocol is a minimalistic flash loan system built around a pool contract holding 1000 WETH, offering flash loans with a fixed 1 WETH fee per loan, regardless of the amount borrowed. The pool supports meta-transactions via a permissionless forwarder, allowing anyone to relay calls on behalf of users without authentication.
A sample `FlashLoanReceiver` contract, funded with 10 WETH, is capable of initiating flash loans from the pool. However, due to flawed assumptions and lack of safeguards, the system is vulnerable to griefing attacks, fund draining, and access control issues, putting all assets at risk, both from the pool and the user contract.
An emergency response requires recovering and consolidating all WETH into a designated recovery account to prevent malicious exploitation.

4.1 Roles

- **NaiveReceiverPool:** Provides fixed fee flash loans using WETH as the lending asset.
- **FlashLoanReceiver:** Requests and receives flash loans from the pool to perform arbitrary operations.
- **BasicForwarder:** Facilitates meta-transactions by forwarding signed requests to target contracts.

5 Executive Summary

The `Naive Recevier` smart contracts protocol audited by Ahmad Faraz. The audit identified 9 issues in total, classified as follows.

Severity Level	Issue Count
Critical	1
High	2
Low	2
Informational	3
Gas	1
Total	**9**

6 Findings

6.1 Critical Severity

6.1.1 [C-1] Lack of Access Control on `NaiveReceiverPool::withdraw`

- **Description:**The `NaiveReceiverPool` protocol provides flash loans to `FlashLoanReceiver`. When `FlashLoanReceiver` takes a flash loan via the `flashLoan` function, it returns a fee as profit to `NaiveReceiverPool`. The `withdraw` function allows `NaiveReceiverPool` to withdraw its profit or `totalDeposits`. However, the function lacks access control, enabling anyone to call withdraw and steal funds.

```
function withdraw(uint256 amount, address payable receiver)
    external {
    deposits[_msgSender()] -= amount;
    totalDeposits -= amount;
    weth.transfer(receiver, amount);
}
```

- **Impact:** Anyone can withdraw all funds from `NaiveReceiverPool`.
- **Cause:** Lack of ownership.

- **Recommended Mitigation:** Use the `Ownable` contract at deployment and create an `onlyOwner` modifier for the withdraw function to restrict access to the contract owner.

```
+ import "@openzeppelin/contracts/access/Ownable.sol";

+ contract NaiveReceiverPool {
+ contract NaiveReceiverPool is Ownable {
    // existing code...

    function withdraw(uint256 amount, address payable receiver)
        external
+       onlyOwner
    {
        deposits[_msgSender()] -= amount;
        totalDeposits -= amount;
        weth.transfer(receiver, amount);
    }
    // optionally, set the owner during deployment if constructor
        exists
}
```

- **Proof of Concept:**
  1. Anyone can deposit tokens.
  2. `FlashLoanReceiver` takes a loan.
  3. The fee is returned.
  4. Anyone calls withdraw.
  5. Funds are withdrawn.

  Run the code in NaiveReceiverPool.t.sol:

```
function testWithDraw() public {
    console2.log("pool balance before:", weth.balanceOf(address(
        naivePool)));
    naivePool.withdraw(naivePool.totalDeposits(), payable(address(
        feeStealer)));
    console2.log("fee stealer:", weth.balanceOf(feeStealer));
    console2.log("pool balance after:", weth.balanceOf(address(
        naivePool)));
}
```

6.2 High Severity

6.2.1 [H-1] Meta-Transaction Signature Reuse - Forwarder Attack

- **Description:**The `BasicForwarder` implements `EIP712`, allowing off-chain signatures to be used by a relayer. A malicious actor can trick a legitimate user into signing a malicious signature, which the attacker can use to misuse the user's funds and send them to `NaiveReceiverPool` as a fee for a flashLoan.

```
function onFlashLoan(address initiator, address token, uint256
    amount, uint256 fee, bytes calldata) external returns (bytes32)
    {
    if (initiator != address(this) || msg.sender != address(vault)
        || token != address(vault.asset()) || fee != 0) {
        revert UnexpectedFlashLoan();
```

```
    }
    ERC20(token).approve(address(vault), amount);
    return keccak256("IERC3156FlashBorrower.onFlashLoan");
}
```

- **Impact:**The `FlashLoanReceiver` funds will be burned.
- **Cause:** Signature reuse and Replay attacks.
- **Recommended Mitigation:** To prevent signature reuse and replay attacks, implement a strict nonce management and ensure each meta-transaction is processed only once. This can be enforced by maintaining a mapping of used nonces per user and rejecting any meta-transaction with a previously used nonce.

```
+ mapping(address => uint256) public nonces;
+ mapping(bytes32 => bool) public executed;

function execute(address from, address to, uint256 value, bytes
    calldata data, uint256 nonce, bytes calldata signature) external
    {

+   require(nonce == nonces[from], "Invalid nonce");
+   bytes32 txHash = keccak256(abi.encodePacked(from, to, value,
    data, nonce));
+   require(!executed[txHash], "Meta-tx already executed");

    // EIP712 verify signature...

+   executed[txHash] = true;
+   nonces[from]++;

    (bool success, ) = to.call{value: value}(data);
    require(success, "Call failed");
}
```

- **Proof of Concept:**
  1. A user signs an off-chain signature.
  2. A malicious user tricks the user into signing a `BasicForwarder.Request`.
  3. The malicious user signs the signature for themselves.
  4. The signature is used in `FlashLoanReceiver.onFlashLoan`.
  5. All funds are burned to `NaiveReceiverPool` as a fee.

```
function testDrainUserFunds() public {
    // 1. Using dummy key
    uint256 privateKey = 0xA11CE;
    address signer = vm.addr(privateKey);

    // 2. Build calldata
    bytes memory callData = abi.encodeWithSignature("attack()");

    // 3. Build the request
    BasicForwarder.Request memory req = BasicForwarder.Request({
       from: signer, target: address(attacker),  value: 0, gas: 100
       _000, nonce: basicForwarder.nonces(signer), data: callData,
       deadline: block.timestamp + 1 hours });
```

```
11
12      // 4. Hash the struct using EIP712
13      bytes32 typeHash = basicForwarder.getRequestTypehash();
14      bytes32 structHash = keccak256(abi.encode(typeHash, req.from,
            req.target, req.value, req.gas, req.nonce,  keccak256(req.
            data), req.deadline));
15
16      // 5. EIP712 Digest
17      bytes32 digest = keccak256( abi.encodePacked("\x19\x01",
            basicForwarder.domainSeparator(), structHash));
18
19      // 6. Sign with victims private key
20      (uint8 v, bytes32 r, bytes32 s) = vm.sign(privateKey, digest);
21      bytes memory signature = abi.encodePacked(r, s, v);
22
23      // 7. Simulate call to forwarder
24      vm.prank(makeAddr("relayer"));
25      basicForwarder.execute(req, signature);
26
27      console2.log("Signature accepted and forwarded to:", req.target
            );
28      assertEq(weth.balanceOf(signer), 0, "Victim should be drained")
            ;
29      console2.log("Pool WETH balance:", weth.balanceOf(address(
            naivePool)));
30  }
```

6.2.2 [H-2] Underflow Leads to Potential DoS in `NaiveReceiverPool`

- **Description:**At deployment, `NaiveReceiverPool` does not require initial funds. When a user attempts a flash loan exceeding the available deposit amount, the transaction reverts without proper error handling.

```
1  function flashLoan(IERC3156FlashBorrower receiver, address token,
       uint256 amount, bytes calldata data) external returns (bool) {
2      if (token != address(weth)) revert UnsupportedCurrency();
3      weth.transfer(address(receiver), amount);
4
5  @>  totalDeposits -= amount;
6
7      if (receiver.onFlashLoan(msg.sender, address(weth), amount,
           FIXED_FEE, data) != CALLBACK_SUCCESS) {
8          revert CallbackFailed();
9      }
10
11     uint256 amountWithFee = amount + FIXED_FEE;
12     weth.transferFrom(address(receiver), address(this),
           amountWithFee);
13     totalDeposits += amountWithFee;
14     deposits[feeReceiver] += FIXED_FEE;
15
16     return true;
17 }
```

- **Impact:** The revert causes a denial-of-service (DoS) condition, confusing users attempting flash loans. In older Solidity versions, this could lead to an underflow of

funds.

- **Cause:** Underflow of `totalDeposits -= amount;`
- **Recommended Mitigation:**
  1. Require a minimum deposit at deployment to ensure flash loans are available.
  2. Add a require statement to check liquidity before calculations.

```
function flashLoan(IERC3156FlashBorrower receiver, address token,
    uint256 amount, bytes calldata data) external returns (bool) {
     if (token != address(weth)) revert UnsupportedCurrency();
     weth.transfer(address(receiver), amount);

+    if (amount > totalDeposits) revert NotEnoughLiquidity();
     totalDeposits -= amount;

     if (receiver.onFlashLoan(msg.sender, address(weth), amount,
         FIXED_FEE, data) != CALLBACK_SUCCESS) {
          revert CallbackFailed();
     }

     uint256 amountWithFee = amount + FIXED_FEE;
     weth.transferFrom(address(receiver), address(this),
         amountWithFee);
     totalDeposits += amountWithFee;
     deposits[feeReceiver] += FIXED_FEE;

     return true;
}

+ error NotEnoughLiquidity();
```

- **Proof of Concept:**
  1. No funds are in the pool.
  2. The transaction always reverts.
  3. This causes a DoS condition.

6.3 Low Severity

6.3.1 [L-1] `NaiveReceiverPool::flashLoan` Not Following CEI

- **Description:** The `flashLoan` function does not follow the Checks-Effects-Interactions (CEI) pattern. While not currently reentrant, future upgrades could introduce high-severity vulnerabilities, potentially leading to loss of funds.
- **Impact:** Skipping CEI introduces risks for future upgrades or code extensions.
- **Recommended Mitigation:**

```
     function flashLoan(IERC3156FlashBorrower receiver, address
         token, uint256 amount, bytes calldata data) external returns
          (bool) {
     if (token != address(weth)) revert UnsupportedCurrency();

-    weth.transfer(address(receiver), amount);
```

```
-    totalDeposits -= amount;

+    totalDeposits -= amount;                    // Effect
+    weth.transfer(address(receiver), amount); // Interaction

     if (receiver.onFlashLoan(msg.sender, address(weth), amount,
        FIXED_FEE, data) != CALLBACK_SUCCESS) {
         revert CallbackFailed();
     }

     uint256 amountWithFee = amount + FIXED_FEE;
     weth.transferFrom(address(receiver), address(this),
        amountWithFee);
     totalDeposits += amountWithFee;
     deposits[feeReceiver] += FIXED_FEE;

     return true;
}
```

6.3.2 [L-2] `NaiveReceiverPool::flashLoan` Lacks Event Emission

- **Description:** The `flashLoan` function does not emit events, which is crucial for protocol accounting, monitoring, and integration with subgraph/indexer tools.

- **Recommended Mitigation:**

```
  function flashLoan(IERC3156FlashBorrower receiver, address token,
      uint256 amount, bytes calldata data) external returns (bool)
     {
     if (token != address(weth)) revert UnsupportedCurrency();
     weth.transfer(address(receiver), amount);

     totalDeposits -= amount;

     if (receiver.onFlashLoan(msg.sender, address(weth), amount,
        FIXED_FEE, data) != CALLBACK_SUCCESS) {
         revert CallbackFailed();
     }

     uint256 amountWithFee = amount + FIXED_FEE;
     weth.transferFrom(address(receiver), address(this),
        amountWithFee);
     totalDeposits += amountWithFee;
     deposits[feeReceiver] += FIXED_FEE;

+    emit FlashLoanRepaid(address(receiver), amountWithFee);
+    emit FlashLoanFeeCollected(feeReceiver, FIXED_FEE);

     return true;
}

+ event FlashLoanRepaid(address indexed receiver, uint256
    amountWithFee);
+ event FlashLoanFeeCollected(address indexed feeReceiver, uint256
    feeAmount);
```

6.4 Informational

6.4.1 [I-1] `NaiveReceiverPool::withdraw` Lacks Condition Check on totalDeposits Withdrawal

- **Description:** The withdraw function does not verify whether the withdrawal amount exceeds totalDeposits, potentially underflow error.

6.4.2 [I-2] `NaiveReceiverPool::withdraw` Lacks Event Emission

- **Description:** The withdraw function does not emit events, reducing transparency and traceability for monitoring and accounting purposes.

6.4.3 [I-3] `FlashLoanReceiver::pool` Should Be Immutable

- **Description:** The pool variable in `FlashLoanReceiver` is not marked as immutable, despite being set only once during deployment. Marking it as immutable would optimize gas costs and improve code clarity.

6.5 Gas Issue

6.5.1 [G-1]`flashLoanReceiver::_executeActionDuringFlashLoan` Not Used Anywhere

- **Description:** The `_executeActionDuringFlashLoan` function in FlashLoanReceiver is defined but not used anywhere in the codebase, leading to unnecessary code bloat.