// Public Smart Contract Security Assessment 03.25.2024 - 04.08.2024

Limit Orders

Odos

HALBORN

Limit Orders - Odos

Prepared by: HALBORN

Last Updated 07/03/2024

Date of Engagement by: March 25th, 2024 - April 8th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
15	0	0	0	4	11

TABLE OF CONTENTS

1. Introduction

`The Odos team` engaged Halborn to conduct a security assessment on their smart contracts beginning on *03/25/2024* and ending on *04/08/2024*. The security assessment was scoped to the smart contracts provided in the GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 2 weeks for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some minor security issues and recommendations that were mostly addressed by the `Odos team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

Out-Of-Scope

- External libraries and financial-related attacks. Specifically, the Odos Limit Orders project was interacting with the `Odos Router V2`, `Odos Executor` contracts and `Uniswap's Permit2` which were considered out of scope.
- New features/implementations after/with the **remediation commit IDs**.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: odos-limit-orders

(b) Assessed Commit ID: 76e9b84

(c) Items in scope:

- contracts/OdosLimitOrderRouter.sol
- contracts/SignatureValidator.sol
- contracts/UniversalSigValidator.sol

Out-of-Scope:

FILES AND REPOSITORY

(a) Repository: odos-limit-orders

(b) Assessed Commit ID: a6ccad1

(c) Items in scope:

- contracts/OdosLimitOrderRouter.sol
- contracts/SignatureValidator.sol
- contracts/UniversalSigValidator.sol

Out-of-Scope:

FILES AND REPOSITORY

(a) Repository: odos-limit-orders

(b) Assessed Commit ID: 6dac1ec

(c) Items in scope:

- contracts/OdosLimitOrderRouter.sol
- contracts/SignatureValidator.sol
- contracts/UniversalSigValidator.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- 7f115b87f115b8
- 0607f130607f13
- 46a128046a1280
- f826a02f826a02
- 20bfc1f20bfc1f
- 6de3fe06de3fe0
- 4e18d0b4e18d0b
- f30748ff30748f
- d2a9f92d2a9f92
- 170b000170b000

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	4	11

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CENTRALIZATION RISKS	LOW	RISK ACCEPTED
OUTDATED USE OF THIRD-PARTY CODE	LOW	SOLVED - 04/15/2024
SINGLE-STEP OWNERSHIP TRANSFER PROCESS	LOW	SOLVED - 04/15/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PRE-SIGNATURES CAN BE USED TO SCAM EXTERNAL USERS	LOW	RISK ACCEPTED
USE OF ECRECOVER`IS SUSCEPTIBLE TO SIGNATURE MALLEABILITY	INFORMATIONAL	SOLVED - 04/15/2024
OWNER CAN RENOUNCE OWNERSHIP	INFORMATIONAL	SOLVED - 04/16/2024
USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS MAY HELP REDUCE GAS USAGE	INFORMATIONAL	SOLVED - 04/15/2024
NOT ALL EVM-COMPATIBLE CHAINS SUPPORT SOLIDITY 0.8.20+	INFORMATIONAL	SOLVED - 04/15/2024
LACK OF REENTRANCY PROTECTION IN THE FILL FUNCTIONS	INFORMATIONAL	ACKNOWLEDGED
UNSAFE ERC20 OPERATION	INFORMATIONAL	SOLVED - 04/15/2024
MULTIPLE UNLOCKED SOLIDITY VERSIONS IN USE	INFORMATIONAL	SOLVED - 04/15/2024
CACHE ARRAY LENGTH OUTSIDE OF LOOP	INFORMATIONAL	NOT APPLICABLE
LACK OF ZERO ADDRESS CHECK	INFORMATIONAL	SOLVED - 04/15/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNUSED CONSTANT	INFORMATIONAL	SOLVED - 04/15/2024
TYPO IN COMMENTS	INFORMATIONAL	SOLVED - 04/15/2024

7. FINDINGS & TECH DETAILS

7.1 CENTRALIZATION RISKS

// LOW

Description

The `OdosLimitOrderRouter` contract has two roles, `owner` and `filler`, with privileged rights to perform admin tasks and need to be trusted.

- The `owner` role can add and remove addresses to and from the `filler` role, swap tokens held by the `OdosLimitOrderRouter` contract and withdraw funds. It is important to mention that such tokens or funds are not user funds, but router's revenue.
- The `filler` role allows such privileged addresses to fill limit orders of other users using any `context` of their choice. Notice that orders contain arbitrary addresses of supposedly ERC20 contracts, and that `context` is a struct that contains critical data like the address of the `Odos Executor` to interact with.

This can pose a centralization risk in case the private keys for any of the privileged accounts are compromised. Having said that, it is also true that the risk of interacting with external contracts with unexpected behavior is limited due to Odos being the only actor able to fulfil limit orders.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:C/A:C/I:C/D:C/Y:C (4.0)

Recommendation

Several remedial strategies can be employed, including but not limited to: transitioning control to a multi-signature wallet setup, establishing community-driven governance for decision-making on fund management, integrating time locks and/or setting withdrawal limits to prevent abrupt fund depletion.

Remediation Plan

RISK ACCEPTED: The **Odos team** accepted the risk of this finding. It is worth mentioning that, while working on the fixes, Odos added a new liquidator role (`liquidatorAddress`) used to assign the special privilege for an address, together with the owner, to swap router's funds.

7.2 OUTDATED USE OF THIRD-PARTY CODE

// LOW

Description

During the security review, it was noted that the code of the contract `UniversalSigValidator` was extracted from the `AmbireTech` GitHub repository:

```
// Copy-paste from https://github.com/AmbireTech/signature-
validator/blob/main/contracts/EIP6492.sol
// with minimal modifications
```

As part of the security tests, the past audit of the reference code was checked, see: https://github.com/AmbireTech/signature-validator/blob/main/ERC6492-Hunter-Security-Audit-Report-V1.0.pdf
While there were two medium-risk findings that were marked as solved, it was noted that both Ambire's and Odos' implementations were still being affected. After further investigation and contacting the Ambire team (also author of the ERC-6492), it was noted that the vulnerabilities were reintroduced mistakenly. See the following commit: https://github.com/AmbireTech/signature-validator/commit/085b2027f12842a4240c3520eb5314a5544ff8d5

BVSS

AO:A/AC:M/AX:M/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (3.4)

Recommendation

As a general best-security practice, it is highly advisable to consistently review all third-party code being utilized and meticulously assess the accompanying security audits.
For this particular case, we suggest examining the enhancements outlined in this report, along with Ambire's updates in their GitHub repository, to ensure alignment with the latest version.

Remediation Plan

SOLVED: The **Odos team** updated the implementation of the `UniversalSigValidator` contract in line with the reference code.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/7f115b8b24b8472c2f92e2c5fea94cf33d69afb7

7.3 SINGLE-STEP OWNERSHIP TRANSFER PROCESS

// LOW

Description

It was identified that the `OdosLimitOrderRouter` contract inherited from OpenZeppelin's `Ownable` library. Ownership of the contracts that are inherited from the `Ownable` module can be lost, as the ownership is transferred in a single-step process. The address that the ownership is changed to should be verified to be active or willing to act as the `owner`. `Ownable2Step` is safer than `Ownable` for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function testTransferOwnership() public {
  ROUTER.transferOwnership(address(1));
}
```

To run it, just execute the following `forge` command:

```
forge test --mt testTransferOwnership -vvvv
```

Observe that the test passes and the new owner is now the invalid `address(1)`.

```
[PASS] testTransferOwnership() (gas: 10355)
Traces:
  [10355] OdosLimitOrdersTests::testTransferOwnership()
    ├─ [7240]
OdosLimitOrderRouter::transferOwnership(0x0000000000000000000000000000000000000001)
    │   ├─ emit OwnershipTransferred(previousOwner: OdosOwner:
[0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], newOwner:
0x0000000000000000000000000000000000000001)
    │   └─ ← [Stop]
    └─ ← [Stop]
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:P/S:U (2.5)

Recommendation

To mitigate the risks associated with single-step ownership transitions and enhance contract security, it is recommended to adopt a two-step ownership transition mechanism, such as OpenZeppelin's `Ownable2Step`. This approach introduces an additional step in the ownership transfer process, requiring the new owner to accept ownership before the transition is finalized. The process typically involves the current owner calling a function to nominate a new owner, and the nominee then calling another function to accept ownership.
Implementing `Ownable2Step` provides several benefits:
1. **Reduces Risk of Accidental Loss of Ownership**: By requiring explicit acceptance of ownership, the risk of accidentally transferring ownership to an incorrect or zero address is significantly reduced.
2. **Enhanced Security**: It adds another layer of security by ensuring that the new owner is prepared and willing to take over the responsibilities associated with contract ownership.
3. **Flexibility in Ownership Transitions**: Allows for a smoother transition of ownership, as the nominee has the opportunity to prepare for the acceptance of their new role.
By adopting `Ownable2Step`, contract administrators can ensure a more secure and controlled process for transferring ownership, safeguarding against the risks associated with accidental or unauthorized ownership changes.

Remediation Plan

SOLVED: The **Odos team** solved the issue by implementing the recommendations described above.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/0607f136d1d28b8211b0dc19d1d050302a8e8b70

7.4 PRE-SIGNATURES CAN BE USED TO SCAM EXTERNAL USERS

// LOW

Description

During the security review, it was observed that users of the `OdosLimitOrderRouter` can pre-sign an order by using the following external function:

```
/// @notice Sets a pre-signature for the specified order hash
/// @param orderHash EIP712 encoded order hash of single or multi input limit order
/// @param preSigned True to set the order as enabled for filling with pre-sign, false
to unset it
function setPreSignature(
  bytes32 orderHash,
  bool preSigned
)
external
{
  preSignedOrders[msg.sender][orderHash] = preSigned;
  emit OrderPreSigned(orderHash, msg.sender, preSigned);
}
```

A malicious user could pre-sign an order using one smart contract wallet (SCW) of their own, and sell the account afterward for the amount of assets it holds. Then, once the transaction has taken place, the pre-signed Odos limit order can still be filled without the approval of the current SCW owner.
This security finding has been rated as low risk because part of the risk is the responsibility of the user purchasing the SCW. Additionally, as indicated in the "Centraliztion Risks" finding, Odos can decide whether to fill an order or not, so it could still perform some off-chain validations to avoid executing transactions in such circumstances.

Proof of Concept

The following Foundry test can be used to illustrate a potential attack scenario where:
1. The attacker owns a smart contract wallet (SCW) and uses it to pre-sign a valid order
2. Sells the SCW to a victim (transferring the ownership)
3. Filling the order afterwards

```
function test_PreSign_changeOwner() public {
    address accountAddress = address(SCW);

    // mint & approve input token
    MockERC20(defaultOrder.input.tokenAddress).faucet(accountAddress,
```

```
defaultOrder.input.tokenAmount);
    vm.prank(accountAddress);
    MockERC20(defaultOrder.input.tokenAddress).approve(address(ROUTER2),
defaultOrder.input.tokenAmount);

    // encode account address to 20 bytes
    bytes memory encodedSignature = abi.encodePacked(accountAddress);

    // orderOwner still not retrievable because the order has not been pre-signed yet
    vm.expectRevert();
    address orderOwner = ROUTER2.exposed_getOrderOwnerOrRevert(
        orderHash, encodedSignature,
SignatureValidator.SignatureValidationMethod.PreSign
    );

    // pre-sign the order using the SCW
    vm.prank(accountAddress);
    ROUTER2.setPreSignature(orderHash, true);

    // pre-signature working correctly now to retrieve the orderOwner (SCW address)
    orderOwner = ROUTER2.exposed_getOrderOwnerOrRevert(
        orderHash, encodedSignature,
SignatureValidator.SignatureValidationMethod.PreSign
    );

    assertEq(accountAddress, orderOwner);

    // Sell & transfer the ownership of the SCW
    SCW.transferOwnership(VICTIM_ADDRESS);

    // The preSign still works
    orderOwner = ROUTER2.exposed_getOrderOwnerOrRevert(
        orderHash, encodedSignature,
SignatureValidator.SignatureValidationMethod.PreSign
    );

    assertEq(accountAddress, orderOwner);

    SignatureValidator.Signature memory sig =
SignatureValidator.Signature(encodedSignature,
SignatureValidator.SignatureValidationMethod.PreSign);
    OdosLimitOrderRouter.LimitOrderContext memory context =
getDefaultContext(defaultOrder.output.tokenAmount);
```

```
    // Whitelist a filler to fill the order
    ROUTER2.addAllowedFiller(address(this));

    // Allowed filler fills the pre-signed order
    ROUTER2.fillLimitOrder(defaultOrder, sig, context);
}
```

BVSS

AO:S/AC:L/AX:M/C:N/I:C/A:H/D:H/Y:H/R:N/S:U (2.1)

Recommendation

Implement a system to avoid scammers from executing pre-signed limit orders of smart contract accounts they do not own anymore. Odos could check whether the owner of the SCW has changed since the pre-signature was registered.

Remediation Plan

RISK ACCEPTED: The **Odos team** accepted the risk of this finding.

References

SignatureValidator.sol#L91-99

7.5 USE OF `ECRECOVER` IS SUSCEPTIBLE TO SIGNATURE MALLEABILITY

// INFORMATIONAL

Description

The built-in EVM precompile `ecrecover` is susceptible to signature malleability, which could lead to replay attacks. While this is not immediately exploitable, this may become a vulnerability if used elsewhere.

The `ecrecover` function is used in the `isValidSigImpl()` function to recover the address from the signature.

BVSS

AO:A/AC:L/AX:M/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (1.7)

Recommendation

Consider using OpenZeppelin's ECDSA library instead of the built-in function.

Remediation Plan

SOLVED: The **Odos team** solved the issue by implementing the recommendations described above.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/46a12802c5bca8aaf6e77127db3c57e63fc7a39a

References

UniversalSigValidator.sol#L77

7.6 OWNER CAN RENOUNCE OWNERSHIP

// INFORMATIONAL

Description

It was identified that the `OdosLimitOrderRouter` contract inherited from OpenZeppelin's `Ownable` library. In the `Ownable` contracts, the `renounceOwnership()` function is used to renounce the `Owner` permission. Renouncing ownership before transferring would result in the contract having no owner, eliminating the ability to call privileged functions.

```
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby disabling any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function testRenounceOwnership() public {
  ROUTER.renounceOwnership();
}
```

To run it, just execute the following `forge` command:

```
forge test --mt testRenounceOwnership -vvvv
```

Observe that the test passes and the ownership has been renounced (`address(0)`).

```
[PASS] testRenounceOwnership() (gas: 5291)
Traces:
  [8073] OdosLimitOrdersTests::testRenounceOwnership()
    ├─ [7052] OdosLimitOrderRouter::renounceOwnership()
    │   ├─ emit OwnershipTransferred(previousOwner: OdosOwner:
[0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], newOwner:
0x0000000000000000000000000000000000000000)
```

```
    |   └─ ← [Stop]
    └─ ← [Stop]
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:P/S:U (1.3)

Recommendation

It is recommended that the Owner cannot call `renounceOwnership()` without first transferring ownership to another address. In addition, if a multi-signature wallet is used, the call to the `renounceOwnership()` function should be confirmed for two or more users.

Remediation Plan

SOLVED: The **Odos team** solved the issue by implementing the recommendations described above.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/f826a02656eb71044c79ebd2f2da88b44089fdd6

7.7 USE OF CUSTOM ERRORS INSTEAD OF REVERT STRINGS MAY HELP REDUCE GAS USAGE

// INFORMATIONAL

Description

In Solidity smart contract development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts.
The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (1.0)

Recommendation

It is recommended to replace hard-coded revert strings in `require` statements for custom errors, which can be done following the logic below.
1. Standard require statement (to be replaced):

```
require(condition, "Condition not met");
```

2. Declare the error definition to state

```
error ConditionNotMet();
```

3. As currently is not possible to use custom errors in combination with `require` statements, the standard syntax is:

```
if (!condition) revert ConditionNotMet();
```

More information about this topic in Official Solidity Documentation.

Remediation Plan

SOLVED: The **Odos team** solved the issue by implementing the recommendations described above.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/20bfc1fa2367f3ab8ffbe0fd366c76afe632db31

References

UniversalSigValidator.sol#L69

7.8 NOT ALL EVM-COMPATIBLE CHAINS SUPPORT SOLIDITY 0.8.20+

// INFORMATIONAL

Description

It was identified that the contracts in scope are using or accepting a Solidity version greater or equal than [0.8.20](https://github.com/ethereum/solidity/releases/tag/v0.8.20), with its default Shanghai EVM version. The Shanghai fork introduced the `PUSH0` opcode that is still not supported on all EVM chains.
Furthermore, by reviewing the Foundry configuration (`foundry.toml` file), it was observed that the contracts would be deployed using the Solidity compiler version `0.8.24`:

```
solc-version = "0.8.24"
```

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (0.5)

Recommendation

Given that the contracts in scope are meant to be deployed to multiple chains (some of them not yet supporting the new opcode `PUSH0`), it is recommended to consider fixing the pragma Solidity version to a version that does not use `PUSH0`.

Remediation Plan

SOLVED: The **Odos team** solved the issue by fixing the Solidity version to `0.8.19` on all contracts.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/6de3fe087fd7f21bf11ffbf21c7f777e7cfa9166

7.9 LACK OF REENTRANCY PROTECTION IN THE FILL FUNCTIONS

// INFORMATIONAL

Description

All the functions to fill limit orders in the provided contracts are functionalities that involve the transfer of ERC20 tokens to external addresses. However, none of these functions implement any reentrancy protection mechanism. While the Odos team indicated that the fulfillment of limit orders will be executed by them, so they could prevent certain behaviors, it is always recommended to have a reentrancy protection built-in.

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (0.5)

Recommendation

Even with the limited risk, consider implementing re-entrancy protection on all `external` and `public` functions with external interactions.
To protect against cross-function reentrancy attacks, it may be necessary to use a mutex. By using this lock, an attacker can no longer exploit the withdraw function with a recursive call. OpenZeppelin has its own mutex implementation called `ReentrancyGuard` which provides a modifier to any function called `nonReentrant` that guards the function with a mutex against Reentrancy attacks.

Remediation Plan

ACKNOWLEDGED: The **Odos team** acknowledged this finding.

7.10 UNSAFE ERC20 OPERATION

// INFORMATIONAL

Description

The `approve()` function defined by the ERC20 token standard can be vulnerable to race-condition attacks if not protected carefully. The main problem is that the function overwrites the current allowance without considering whether the spender has already utilized it, potentially leading to unintended token transfers beyond the token owner's intention. For more information, see: https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
As shown below, the `OdosLimitOrderRouter` contract is making use of the insecure `approve()` function, and it is not checking the returned value:

```
// Approve spending
IERC20(tokensIn[i]).approve(inputReceivers[i], amountsIn[i]);
```

BVSS

AO:S/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (0.4)

Recommendation

To circumvent ERC20's approve functions race-condition vulnerability, use OpenZeppelin's `SafeERC20` library's `safe{Increase|Decrease}Allowance()` functions.

Remediation Plan

SOLVED: The **Odos team** solved the issue by removing the use of the `approve()` function.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/4e18d0bf1ee961d6ac2fc51325c3ab571f1ac3ca

References

OdosLimitOrderRouter.sol#L539

7.11 MULTIPLE UNLOCKED SOLIDITY VERSIONS IN USE

// INFORMATIONAL

Description

The contracts in scope use multiple pragma versions with unlocked versions. Contracts should be deployed with the same compiler version and flags with which they were thoroughly tested. Locking the pragma helps to ensure that contracts are not accidentally deployed using another pragma, for example, either an outdated pragma version that could introduce bugs that negatively affect the contract system or a recently released pragma version that has not been extensively tested.

Reference: https://github.com/ethereum/solidity/releases

In the Solidity GitHub repository, there is a json file where are all the bugs found in the different compiler versions.

Reference: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json"

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider locking the pragma version. When possible, do not use floating pragmas. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan

SOLVED: The **Odos team** solved the issue by fixing the Solidity version to `0.8.19` on all contracts.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/6de3fe087fd7f21bf11ffbf21c7f777e7cfa9166

7.12 CACHE ARRAY LENGTH OUTSIDE OF LOOP

// INFORMATIONAL

Description

In Solidity, when working with arrays, it is common to use a loop to iterate over the elements. However, if the array length is not cached outside the loop, the compiler will read the length of the array during each iteration. This can result in additional gas consumption and reduced efficiency.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

After thorough investigation, it was noted that none of the instances were using arrays from `storage` but from `calldata`. Therefore, this finding has been marked as 'Not Applicable' because the gas savings no longer apply.

Remediation Plan

NOT APPLICABLE: The **Odos team** stated that the finding is not applicable.

References

OdosLimitOrderRouter.sol#L414
OdosLimitOrderRouter.sol#L458
OdosLimitOrderRouter.sol#L531
OdosLimitOrderRouter.sol#L584
OdosLimitOrderRouter.sol#L659
OdosLimitOrderRouter.sol#L663
OdosLimitOrderRouter.sol#L861
OdosLimitOrderRouter.sol#L862
OdosLimitOrderRouter.sol#L916
OdosLimitOrderRouter.sol#L924
OdosLimitOrderRouter.sol#L948
OdosLimitOrderRouter.sol#L959
OdosLimitOrderRouter.sol#L969

7.13 LACK OF ZERO ADDRESS CHECK

// INFORMATIONAL

Description

The constructor of the `OdosLimitOrderRouter` contract lacks protection against using the `address(0)` value for the parameter named `_odosRouterV2`. Consequently, if called with a value of 0, the `ODOS_ROUTER_V2` state variable could remain uninitialized, rendering most functionalities unusable.

```
constructor(address initialOwner, address _odosRouterV2)
  EIP712("OdosLimitOrderRouter", "1")
  Ownable(initialOwner) {
    ODOS_ROUTER_V2 = _odosRouterV2;
}
```

Additionally, the `transferRouterFunds()` function contained a `dest` address parameter that is never checked against `address(0)` to avoid burning router funds mistakenly.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to make sure all addresses are properly initialized (avoiding the zero address) as this would break critical functionalities and, given that `ODOS_ROUTER_V2` is an `immutable` variable, the situation would not be reversible.

Remediation Plan

SOLVED: The **Odos team** solved the issue by implementing the recommendations described above.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/f30748fc6ca92c255a136f0be71390c271de9e5f

References

OdosLimitOrderRouter.sol#L301-L305
OdosLimitOrderRouter.sol#L575-L597

7.14 UNUSED CONSTANT

// INFORMATIONAL

Description

The `SignatureValidator` contract declares the following constant, but it is never used.

```
// bytes4(keccak256("isValidSignature(bytes32,bytes)")
bytes4 internal constant EIP1271_MAGICVALUE = 0x1626ba7e;
```

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Unused variables should be cleaned up from the code if they have no purpose. Clearing these variables can reduce gas usage during the deployment of contracts, and also increases the readability of the code.

Remediation Plan

SOLVED: The **Odos team** solved the issue by removing the unused constant.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/d2a9f92ef9ea2ce6a255e8e13053a77a3c6eb3a0

References

SignatureValidator.sol#L14-15

7.15 TYPO IN COMMENTS

// INFORMATIONAL

Description

The following comment was found in the `UniversalSigValidator` contract:

```
// The order here is striclty defined in https://eips.ethereum.org/EIPS/eip-6492
```

The word `striclty` above should be spelled as `strictly` instead.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

To maintain clarity and trustworthiness, it is essential to rectify any typographical errors present within the contracts. Correcting such errors minimizes the likelihood of confusion and reinforces confidence in the accuracy and integrity of the documentation.

Remediation Plan

SOLVED: The **Odos team** solved the issue by implementing the recommendations described above.

Remediation Hash

https://github.com/odos-xyz/odos-limit-orders/commit/170b0001e6b2ba148dd5791156bdbd6fd7cd4db8

References

UniversalSigValidator.sol#L30

STATIC ANALYSIS REPORT

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Output

```
INFO:Detectors:
OdosLimitOrderRouter.fillLimitOrder(OdosLimitOrderRouter.LimitOrder,SignatureValidator.Signature,OdosLimitOrderRouter.LimitOrderContext) (contracts/OdosLimitOrderRouter.sol#240-262) uses arbitrary
 from in transferFrom: IERC20(order.input.tokenAddress).safeTransferFrom(orderOwner,context.inputReceiver,context.currentAmount) (contracts/OdosLimitOrderRouter.sol#258)
OdosLimitOrderRouter.fillMultiLimitOrder(OdosLimitOrderRouter.MultiLimitOrder,SignatureValidator.Signature,OdosLimitOrderRouter.MultiLimitOrderContext) (contracts/OdosLimitOrderRouter.sol#308-337)
 uses arbitrary from in transferFrom: IERC20(order.inputs[i].tokenAddress).safeTransferFrom(orderOwner,context.inputReceivers[i],context.currentAmounts[i]) (contracts/OdosLimitOrderRouter.sol#327-
329)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
```

```
INFO:Detectors:
OdosLimitOrderRouter.swapRouterFunds(OdosLimitOrderRouter.TokenInfo[],address[],OdosLimitOrderRouter.TokenInfo,address,bytes,address) (contracts/OdosLimitOrderRouter.sol#416-466) ignores return
 value by IERC20(tokensIn[i]).approve(inputReceivers[i],amountsIn[i]) (contracts/OdosLimitOrderRouter.sol#435)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.