SECURITY REVIEW REPORT FOR 1INCH

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

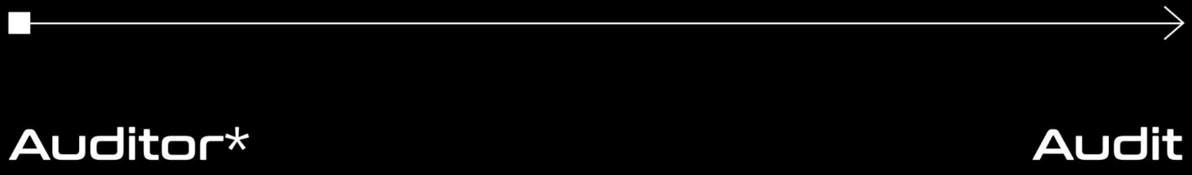
Lead Smart Contract Auditor | Hexens

Audit Starting Date	Audit Completion Date
16.03.2023	03.04.2023

METHODOLOGY

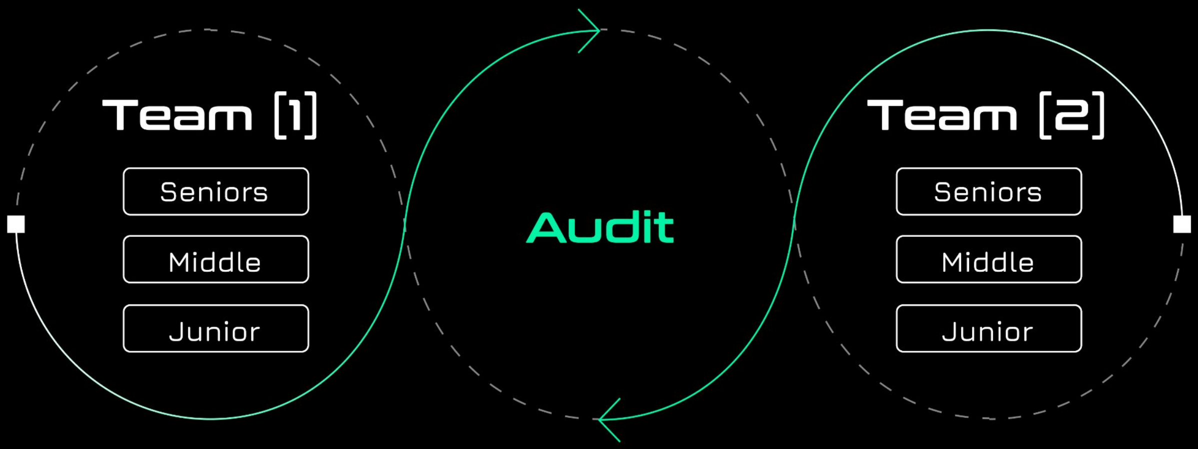
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.



HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

The newly developed projects by 1inch include ERC20Pods, limit order settlements and a new delegation system. The latter two are both build upon the ERC20Pods contracts.

Our security assessment was a full review of the projects and its smart contracts. We have thoroughly reviewed each contract individually, as well as the system as a whole.

During our audit, we have identified 1 critical vulnerability in the ERC20Pods contracts and as a result both St1inch and the delegation system were affected. The vulnerability would allow an attacker to gain an arbitrary amount of voting power.

We have also identified various minor vulnerabilities and optimisations.

Finally, all of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

The newly developed projects by 1inch include ERC20Pods, limit order settlements and a new delegation system. The latter two are both build upon the ERC20Pods contracts.

Our security assessment was a full review of the projects and its smart contracts. We have thoroughly reviewed each contract individually, as well as the system as a whole.

During our audit, we have identified 1 critical vulnerability in the ERC20Pods contracts and as a result both St1inch and the delegation system were affected. The vulnerability would allow an attacker to gain an arbitrary amount of voting power.

We have also identified various minor vulnerabilities and optimisations.

Finally, all of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/1inch/1inch-contract/commit/b301c15f8c58d51b371cfd10baf445461d09d1b7
https://github.com/1inch/limit-order-protocol/commit/59a1b4a90fd97d7beb5cb393e914a2e447e711dc
https://github.com/1inch/solidity-utils/commit/42615efa5dca3ed43be565097ccb82a9c3e87273

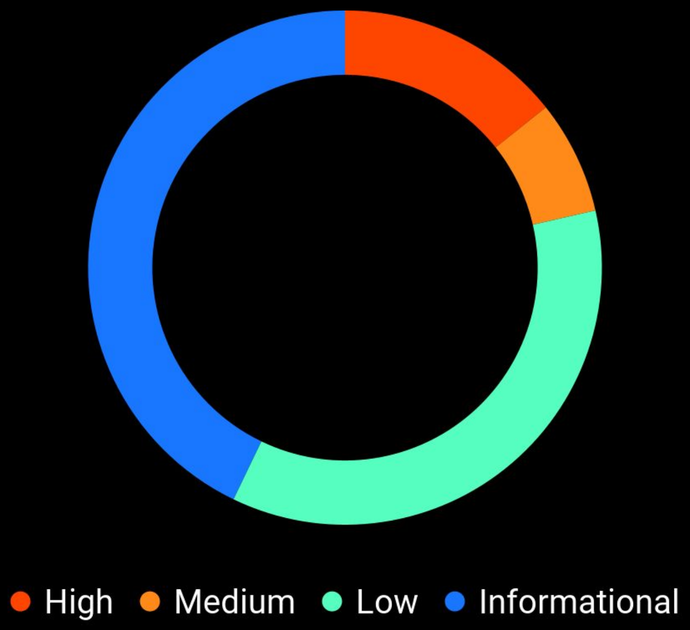
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

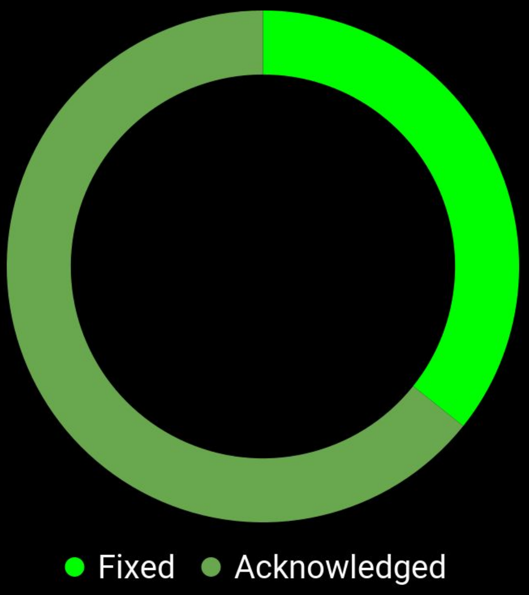
SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	2
MEDIUM	1
LOW	5
INFORMATIONAL	6

TOTAL: 14

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

ONI-12. UNOSWAP ROUTER SWAPS CAN BE BLOCKED FOR USDT ON CURVE

SEVERITY: High

PATH: routers/UnoswapRouter.sol:_curfe (L338-502)

REMEDIATION: clear the approval of the from token for the pool after the swap

STATUS: fixed

DESCRIPTION:

The function `_curfe` is called from `_unoswap` to facilitate a swap through a Curve pool.

Basically the function will call the right `coins` function on the Curve pool with the provided indices to get the `from` and `to` tokens. Then the function calls `ERC20.approve` on the `from` token with the pool and the specified amount.

However, this mechanism is not compatible with USDT, which reverts on an `ERC20.approve` if there already exists a non-zero allowance from the `sender` (the router) to the `spender` (the pool).

Therefore if such an allowance already exists before the swap, then any subsequent swap using USDT through the Curve swap function will revert on **USDT.approve** and fail.

Thanks to the customisability of the Unoswap Router it becomes possible for an attacker to create such a 'hanging' allowance (that will still exist after the swap). For example:

1. The attacker calls **unoswap** with USDT as token, 1 wei as value, 0 wei as minimum return amount and a Curve pool with USDT as DEX (e.g. DAI/USDC/USDT pool).
2. The DEX address would have the following encoded:
    a. **coins(uint256)** as coins selector
    b. 2 as **from** token (USDT)
    c. 1 as **to** token (USDC)
    d. **HAVE_ARG_DESTINATION** as true
    e. **coins(uint256)** as swap selector
3. The router would now call **ERC20.approve** on USDT for the pool for 1 wei.
4. The call for the swap will actually use the **coins(uint256)** selector (the selector is fully controllable by the caller) and as a result nothing is swapped (and the allowance is not used) but the call does succeed.
5. Then the **HAVE_ARG_DESINATION** flag allows for skipping the block on line 467 so that the router doesn't try to send the fake amount back to the attacker and revert.

Afterwards the call succeeds and the router has an existing allowance of 1 wei for the Curve pool. The attacker can then repeat this for every Curve pool with USDT.

Any subsequent swaps from other users would fail for USDT on any Curve pool.

```
if or(iszero(useEth), and(useEth, eq(toToken, _WETH))) {
  let fromSelectorOffset := and(shr(_CURVE_FROM_COINS_SELECTOR_OFFSET, dex),
_CURVE_FROM_COINS_SELECTOR_MASK)
  let fromTokenIndex := and(shr(_CURVE_FROM_COINS_ARG_OFFSET, dex), _CURVE_FROM_COINS_ARG_MASK)
  let fromToken := curveCoins(pool, fromSelectorOffset, fromTokenIndex)
  if eq(fromToken, 0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee) {
     fromToken := _WETH
  }

  // fromToken.approve(pool, amount)
  let ptr := mload(0x40)
  mstore(ptr, 0x095ea7b300000000000000000000000000000000000000000000000000000000) //
IERC20.approve.selector
  mstore(add(ptr, 0x04), pool)
  mstore(add(ptr, 0x24), amount)
  safeERC20(fromToken, 0, ptr, 0x44)
}
```

ONI-6. MAKER CAN STEAL TAKER'S ASSETS IF THRESHOLD IS ZERO

SEVERITY: High

PATH: limit-order-protocol/OrderMixin.sol:_fillOrderTo (L240-426)

REMEDIATION: either make threshold mandatory (e.g. revert if it's equal to zero) or to use a default threshold if it's equal to zero (e.g. 1.05 times the specified amount at the initial rate as defined by the maker)

STATUS: acknowledged

DESCRIPTION:

When filling an order, the taker provides **TakerTraits** which includes a threshold. This threshold is used to calculate an acceptable ratio between the maker amount and the taker amount.

This is especially important because the maker or taker amount can be calculated on the spot through **calculateMakingAmount** or **calculateTakingAmount**, which can make a static call to a contract specified by the maker.

If the threshold is zero (which is the default), then this check is skipped. This allows the maker to change the maker or taker amount, such that the taker pays the full amount, while the maker only transfers an insignificant amount. This is effectively stealing.

Simulating the transaction for the taker also does not offer full protection (e.g. through **eth_call**). The maker has ways to bypass this, such as having the contract that calculates the amounts return a normal amount if it's a simulation and an almost zero amount if it is a real transaction.

```
if (takerTraits.isMakingAmount()) {
   makingAmount = Math.min(amount, remainingMakingAmount);
   takingAmount = order.calculateTakingAmount(extension, makingAmount, remainingMakingAmount, orderHash);

   uint256 threshold = takerTraits.threshold();
   if (threshold > 0) {
      // Check rate: takingAmount / makingAmount <= threshold / amount
      if (amount == makingAmount) {  // Gas optimization, no SafeMath.mul()
         if (takingAmount > threshold) revert TakingAmountTooHigh();
      } else {
         if (takingAmount * amount > threshold * makingAmount) revert TakingAmountTooHigh();
      }
   }
}
else {
   takingAmount = amount;
   makingAmount = order.calculateMakingAmount(extension, takingAmount, remainingMakingAmount, orderHash);
   if (makingAmount > remainingMakingAmount) {
      // Try to decrease taking amount because computed making amount exceeds remaining amount
      makingAmount = remainingMakingAmount;
      takingAmount = order.calculateTakingAmount(extension, makingAmount, remainingMakingAmount, orderHash);
      if (takingAmount > amount) revert TakingAmountExceeded();
   }

   uint256 threshold = takerTraits.threshold();
   if (threshold > 0) {
      // Check rate: makingAmount / takingAmount >= threshold / amount
      if (amount == takingAmount) { // Gas optimization, no SafeMath.mul()
         if (makingAmount < threshold) revert MakingAmountTooLow();
      } else {
         if (makingAmount * amount < threshold * takingAmount) revert MakingAmountTooLow();
      }
   }
}
```

ONI-11. TAKER CAN BE GAS GRIEFED BY MAKER

SEVERITY: Medium

PATH: see description

REMEDIATION: add some gas limits for external calls to any address controlled by the maker

STATUS: acknowledged

DESCRIPTION:

There are multiple locations in the **OrderMixin.sol** where an external call is made to a contract controlled by the maker when the taker wants to fulfil an order:

1. In **_applyMakerPermit** on line 441 a permit call is made to an arbitrary address.
2. In **_fillOrderTo** on line 266 the predicate is checked which can make an arbitrary static call.
3. In **_fillOrderTo** on line 273, line 287 and line 291 a static call is made to an arbitrary address.
4. In **_fillOrderTo** on line 323 and line 420 an interaction call is made to an arbitrary address.

Each of these calls do not enforce a gas limit and the maker can therefore spend all of the taker's gas in one of these calls, effectively gas griefing the taker.

Transaction simulation (e.g. through **eth_estimateGas** and **eth_call**) does not protect from this type of attack. A contract is able to know whether execution is a simulation or a real transaction and act accordingly to hide malicious behaviour with regards to gas griefing.

```
if (order.makerTraits.needPreInteractionCall()) {
   bytes calldata data = extension.preInteractionTargetAndData();
   address listener = order.maker.get();
   if (data.length > 0) {
       listener = address(bytes20(data));
       data = data[20:];
   }
   IPreInteraction(listener).preInteraction(
       order, orderHash, msg.sender, makingAmount, takingAmount, remainingMakingAmount, data
   );
}
[...]
if (order.makerTraits.needPreInteractionCall()) {
   bytes calldata data = extension.preInteractionTargetAndData();
   address listener = order.maker.get();
   if (data.length > 0) {
       listener = address(bytes20(data));
       data = data[20:];
   }
   IPreInteraction(listener).preInteraction(
       order, orderHash, msg.sender, makingAmount, takingAmount, remainingMakingAmount, data
   );
}
```

ONI-2. INCORRECT TYPE HASH FOR LIMIT ORDER

SEVERITY: Low

PATH: limit-order-protocol/OrderLib.sol:_LIMIT_ORDER_TYPEHASH (L24-34)

REMEDIATION: Change the type address to uint256 for the _LIMIT_ORDER_TYPEHASH

STATUS: acknowledged

DESCRIPTION:

We found that the limit order type hash in in the OrderLib library is incorrectly generated.

It uses **address** for the **maker**, **makerAsset** and **takerAsset** fields. However, these fields are actually of type **Address**, which is **uint256** as defined in **AddressLib.sol**.

```
bytes32 constant internal _LIMIT_ORDER_TYPEHASH = keccak256(
  "Order("
    "uint256 salt,"
    "address maker,"
    "address makerAsset,"
    "address takerAsset,"
    "uint256 makingAmount,"
    "uint256 takingAmount,"
    "uint256 makerTraits"
  ")"
);
```

ONI-3. ORDERLIB AMOUNT GETTER INCORRECT LENGTH CHECK

SEVERITY: Low

PATH: limit-order-protocol/OrderLib.sol:_callGetter (L79-90)

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

The OrderLib allows for dynamically getting the taker or maker amount from a maker-defined address. The data is encoded in the extension.

However, the function only checks whether the length is at least 20 bytes for the target of the call and as a result, the selector could potentially be missing. In that case, the first 4 bytes of the **requestedAmount** would become the selector and the last 4 bytes would be shifted into the next parameter.

```
function _callGetter(
  bytes calldata getter,
  uint256 requestedAmount,
  uint256 remainingMakingAmount,
  bytes32 orderHash
) private view returns(uint256) {
  if (getter.length < 20) revert WrongGetter();

  (bool success, bytes memory result) = address(bytes20(getter)).staticcall(abi.encodePacked(getter[20:],
requestedAmount, remainingMakingAmount, orderHash));
  if (!success || result.length != 32) revert GetAmountCallFailed();
  return abi.decode(result, (uint256));
}
```

change line 85 in OrderLib.sol to:

```
if (getter.length < 24) revert WrongGetter();
```

ONI-8. EXTENSION CHECK FOR PREDICATE OPTIMISATION

SEVERITY: Low

PATH: limit-order-protocol/OrderMixin.sol:_fillOrderTo (L240-426)

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In **_fillOrderTo** on line 263, the order's maker traits are checked for an extension and only if it exist, will it perform the predicate check.

This check can be optimised by directly checking **extension**, because it was already validated whether there should be an extension in **fillOrderToExt** and **fillContractOrderExt** with **order.validateExtension(extension)**. In other words, if the extension is not empty, then **makerTraits.hasExtension()** is **true**.

The check can be optimised to:

```
if (extension.length != 0) {
  bytes calldata predicate = extension.predicate();
  if (predicate.length > 0) {
     if (!checkPredicate(predicate)) revert PredicateIsNotTrue();
  }
}
```

The check is already redundant, because **extension.predicate()** would return an empty predicate if **extension** is also empty. But it is more gas efficient to first check extension if not having a predicate is the more frequent case.

```
if (order.makerTraits.hasExtension()) {
  bytes calldata predicate = extension.predicate();
  if (predicate.length > 0) {
     if (!checkPredicate(predicate)) revert PredicateIsNotTrue();
  }
}
```

ONI-7. MAKER/TAKER ZERO AMOUNT CHECK OPTIMISATION

SEVERITY: Low

PATH: limit-order-protocol/OrderMixin.sol:_fillOrderTo (L240-426)

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In **_fillOrderTo** on line 306, both the maker and taker amount are checked against zero.

We found that this check could be optimised to save gas on each call:

```
if (makingAmount & takingAmount == 0) revert SwapWithZeroAmount();
```

```
if (makingAmount == 0 || takingAmount == 0) revert SwapWithZeroAmount();
```

ONI-5. WETH VARIABLE IS DEFINED AND SET TWICE

SEVERITY: Low

PATH: limit-order-protocol/OrderMixin.sol (L40)

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

The **OrderMixin** contract defines a private immutable variable for WETH. However, it also inherits from **OnlyWethReceiver** which also defines a private immutable variable for WETH, which gets set through the constructor.

The same applies to **limit-order-protocol/helpers/WethUnwrapper.sol** and **limit-order-protocol/helpers/ETHOrders.sol**.

By changing the **_WETH** variable in **OnlyWethReceiver** to **internal**, removing the private immutable variable **_WETH** from **OrderMixin**, **WethUnwrapper** and **ETHOrders**, and changing **address(_WETH)** to **_WETH** and **_WETH.** to **IWETH(_WETH).** almost 100k gas can be saved for deployment.

```
IWETH private immutable _WETH;


constructor(IWETH weth) OnlyWethReceiver(address(weth)) {
  _WETH = weth;
}
```

ONI-1. INCORRECT DOCUMENTATION

SEVERITY: Informational

PATH: see description

REMEDIATION: fix the incorrect comments

STATUS: acknowledged

DESCRIPTION:

We have identified incorrect documentation in the code:

1. **solidity-utils/contracts/libraries/SafeERC20.sol** the documentation string functions **safeTransferFromUniversal**, **safeTransferFrom** and **safeTransfer** on line 24, 39 and 94 respectively it states that the function allows for calling to non-smart-contracts addresses. However this is false, as the code size of the target is checked to be greater than 0 upon success of the call and if so, it returns false.
2. **limit-order-protocol/contracts/LimitOrderProtocol.sol** on line 9 the version should be 4 instead of 3, so it is in accordance with the parameter values in the EIP712 constructor.

```
/// @dev Ensures method do not revert or return boolean `true`, admits call to non-smart-contract.


* @title ##1inch Limit Order Protocol v3
```

ONI-4. MISSING ADDRESS ZERO CHECKS

SEVERITY: Informational

PATH: see description

REMEDIATION: check each mentioned parameter for address zero

STATUS: acknowledged

DESCRIPTION:

We have identified the following locations where checks for the zero address are missing:

1. **solidity-utils/OnlyWethReceiver.sol**: There is no check for the parameter **weth** in the constructor.

```
constructor(address weth) {
  _WETH = address(weth);
}
```

ONI-9. ORDER FULFILMENT EVENT FIELDS COULD BE INDEXED

SEVERITY: Informational

PATH: limit-order-protocol/interfaces/IOrderMixin.sol:OrderFilled (L46-49)

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

The event **OrderFilled** is emitted when a part of or the full order is fulfilled. It takes the hash and the making amount that was fulfilled in the order.

Because orders can be filled partially and so multiple times per hash, we would recommend to add **indexed** to the **orderHash** field, to allow searching and aggregating of amounts for each order.

```
event OrderFilled(
  bytes32 orderHash,
  uint256 makingAmount
);
```

ONI-10. ORDER SIGNATURE CHECKS ALLOWS FOR ADDRESS ZERO

SEVERITY: Informational

PATH: limit-order-protocol/OrderMixin.sol:fillOrderToExt (L155-178)

REMEDIATION: check that maker's address is not address(0)

STATUS: fixed

DESCRIPTION:

When fulfilling an order, the signature is checked against the provided order's hash and the signer should be equal to the order's maker. The library that is used is **solidity-utils/libraries/ECDSA.sol** and the function called is r**ecover(bytes32, bytes32, bytes32)**, which uses the built-in **ecrecover**.

This function does not check whether the return value of **ecrecover** is **address(0)**, which happens when the signature is invalid. As a result, it is possible to create and 'sign' orders for the zero address.

The orders can still be executed with a custom maker asset, so it is possible to set order bits or remaining amounts for the zero address.

This could potentially cause problems on off-chain or front-end applications.

```
function fillOrderToExt(
  IOrderMixin.Order calldata order,
  bytes32 r,
  bytes32 vs,
  uint256 amount,
  TakerTraits takerTraits,
  address target,
  bytes calldata interaction,
  bytes calldata extension
) public payable returns(uint256 makingAmount, uint256 takingAmount, bytes32 orderHash) {
  order.validateExtension(extension);
  orderHash = order.hash(_domainSeparatorV4());

  // Check signature and apply order permit only on the first fill
  uint256 remainingMakingAmount = _checkRemainingMakingAmount(order, orderHash);
  if (remainingMakingAmount == order.makingAmount) {
     if (order.maker.get() != ECDSA.recover(orderHash, r, vs)) revert BadSignature();
     if (!takerTraits.skipMakerPermit()) {
        _applyMakerPermit(order, orderHash, extension);
     }
  }

  (makingAmount, takingAmount) = _fillOrderTo(order, orderHash, extension,
remainingMakingAmount, amount, takerTraits, target, _wrap(interaction));
}
```

```
function recover(
  bytes32 hash,
  bytes32 r,
  bytes32 vs
) internal view returns (address signer) {
  assembly ("memory-safe") { // solhint-disable-line no-inline-assembly
      let s := and(vs, _COMPACT_S_MASK)
      if lt(s, _S_BOUNDARY) {
          let ptr := mload(0x40)

          mstore(ptr, hash)
          mstore(add(ptr, 0x20), add(27, shr(_COMPACT_V_SHIFT, vs)))
          mstore(add(ptr, 0x40), r)
          mstore(add(ptr, 0x60), s)
          mstore(0, 0)
          pop(staticcall(gas(), 0x1, ptr, 0x80, 0, 0x20))
          signer := mload(0)
      }
  }
}
```

ONI-13. MULTIPLE UNDERFLOWS IN UNOSWAP ROUTER

SEVERITY: Informational

PATH: routers/UnoswapRouter.sol:_curfe (L338-502)

REMEDIATION: considering reverting if the returned swap amount is 0 (before subtracting the 1, or maximum uint256 after subtracting)

STATUS: acknowledged

DESCRIPTION:

We found that there are underflows possible in the function **_curfe** that executes a swap on a Curve pool. More specifically:

1. On line 457 the returned swap amount is calculated from the contract's ETH balance minus 1. If the swap returned 0, then this will underflow to maximum **uint256**.
2. On line 460 and then line 363 the returned swap amount is calculated using **ERC20.balanceOf** minus 1. Again, it the swap returned 0, then this will also underflow to maximum **uint256**.

If due to some configuration or other factors a swap would return 0, then the router would note a swap amount of maximum **uint256** and the **minReturn** check would pass. Instead, the router would revert trying to send maximum **uint256** back to the user.

This will lead to incorrect error messages and a worse user experience.

```
function tokenBalanceOf(tokenAddress, accountAddress) -> tokenBalance {
  mstore(0,
0x70a0823100000000000000000000000000000000000000000000000000000000)
  mstore(4, accountAddress)
  if iszero(callReturnSize(staticcall(gas(), tokenAddress, 0, 0x24, 0, 0x20))) {
    revert(0, 0)
  }
  tokenBalance := sub(mload(0), 1) // Keep 1 wei
}
[...]
switch and(useEth, eq(toToken, _WETH))
case true {
  ret := sub(balance(address()), 1) // Keep 1 wei
}
default {
  ret := tokenBalanceOf(toToken, address())
}
```

ONI-14. MAKERTRAITS ISALLOWEDSENDER COULD BE SIMPLIFIED

SEVERITY: Informational

PATH:
limit-order-protocol/libraries/MakerTraitsLib.sol:isAllowedSender (L31-34)

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function **isAllowedSender** is used to check the taker's address against the allowed sender mask in the Maker Traits.

We found that this function contains a number of redundant conversions and masks and could be simplified to:

```
function isAllowedSender(MakerTraits makerTraits, address sender) internal pure returns (bool) {
   uint160 allowedSender = uint160(MakerTraits.unwrap(makerTraits) & _ALLOWED_SENDER_MASK);
   return allowedSender == 0 || allowedSender == uint160(sender) & _ALLOWED_SENDER_MASK;
}
```

```
function isAllowedSender(MakerTraits makerTraits, address sender) internal pure returns (bool) {
   address allowedSender = address(uint160(MakerTraits.unwrap(makerTraits) &
_ALLOWED_SENDER_MASK));
   return allowedSender == address(0) || (uint160(allowedSender) & _ALLOWED_SENDER_MASK) ==
uint160(sender) & _ALLOWED_SENDER_MASK;
}
```