Oct.23

SECURITY REVIEW REPORT FOR 1INCH

CONFIDENTIAL

CONFIDENTIAL

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.

CONFIDENTIAL

LIMITATIONS ON DISCLOSURE AND USAGE OF THIS REPORT

This report has been developed by the company Hexens (the Service Provider) based on the Smart Contract Audit of 1inch (the Client). The document contains vulnerability information and remediation advice.

The information, presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of 1inch.

If you are not the intended recipient of this document, remember that any disclosure, copying or dissemination of it is forbidden.

CONFIDENTIAL

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
23.10.2023

Audit Completion Date
06.11.2023

CONFIDENTIAL

METHODOLOGY

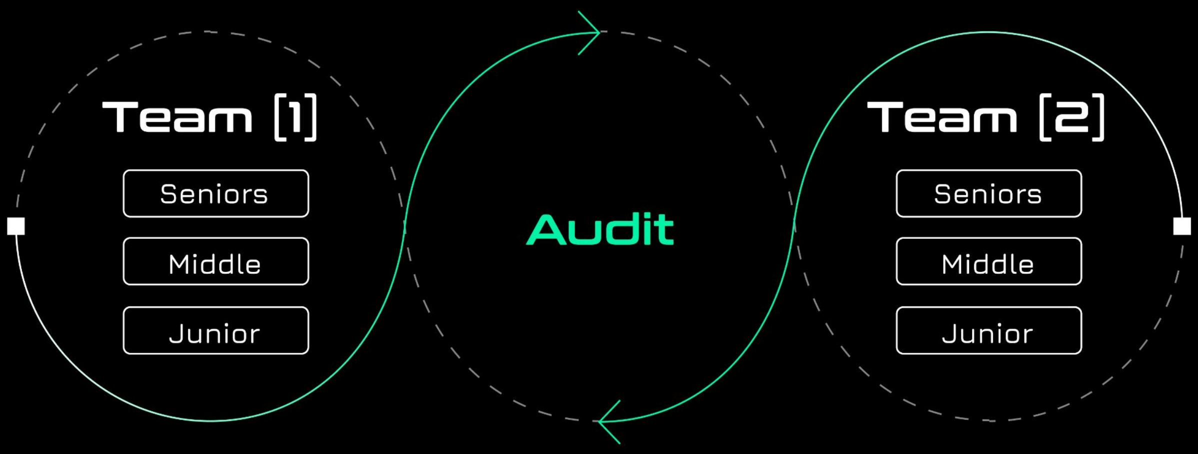
COMMON AUDIT PROCESS

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.

Auditor*
Audit

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.

SCOPE

The analyzed resources are located on:

https://github.com/1inch/1inch-contract/commit/230fe7e53ab10744eb1afa38dac5ddc888ca7b99

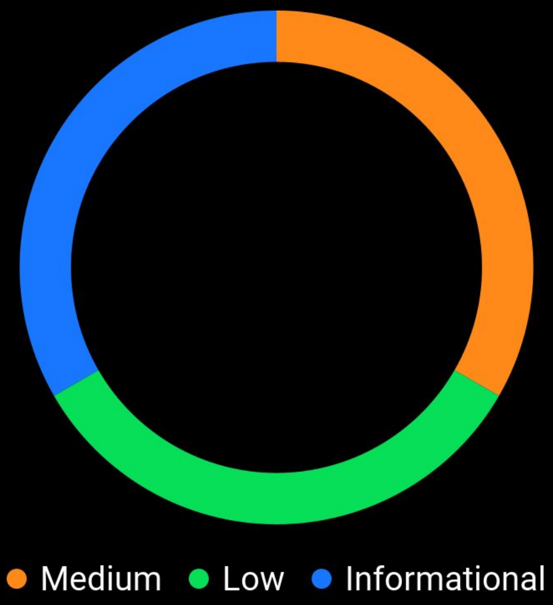
https://github.com/1inch/limit-order-protocol/commit/da3e187f46885dfd234d9f630bd0afdf5c2ccfc0

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	2
INFORMATIONAL	2

TOTAL: 6

SEVERITY



STATUS

WEAKNESSES

This section contains the list of discovered weaknesses.

INOCT-2. INSECURE RESCUE MECHANISM FOR STUCK TOKENS

SEVERITY: Medium

PATH: AggregationRouterV6.sol, UnoswapRouter.sol

REMEDIATION: rescue mechanism for stuck tokens should be secure and accessible to the owner only. The _unoswap function should revert for unsupported protocols

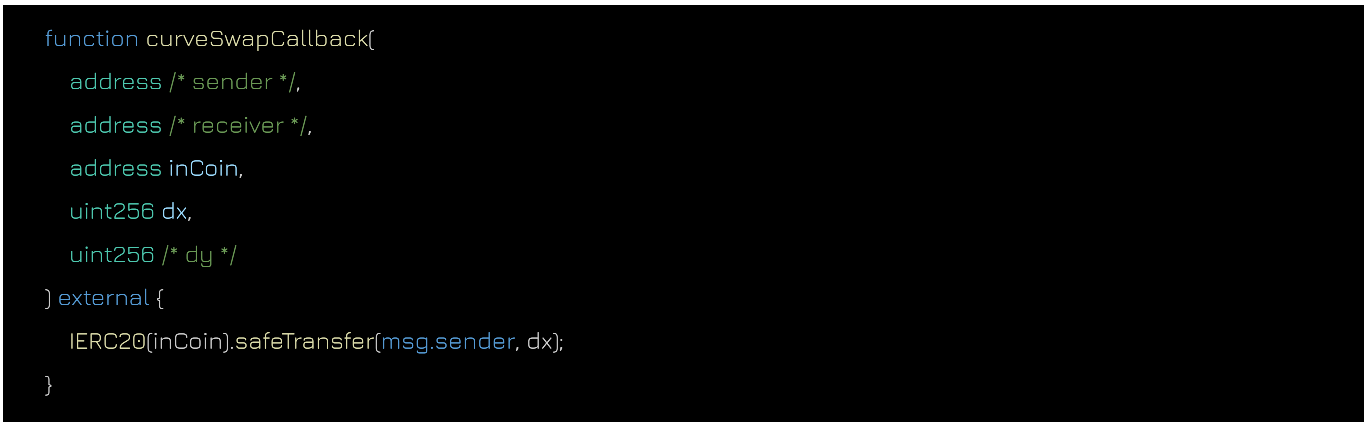
STATUS:

DESCRIPTION:

AggregationRouterV6 has the permissioned **rescueFunds()** function callable by the contact's owner. It should allow to secure users' funds when they are accidentally transferred to the router, or there is an issue during a swap. Case in point for mentioned swap issues could be awry **dex2** or **dex3** parameters for **unoswap2()** or **unoswap3()**.

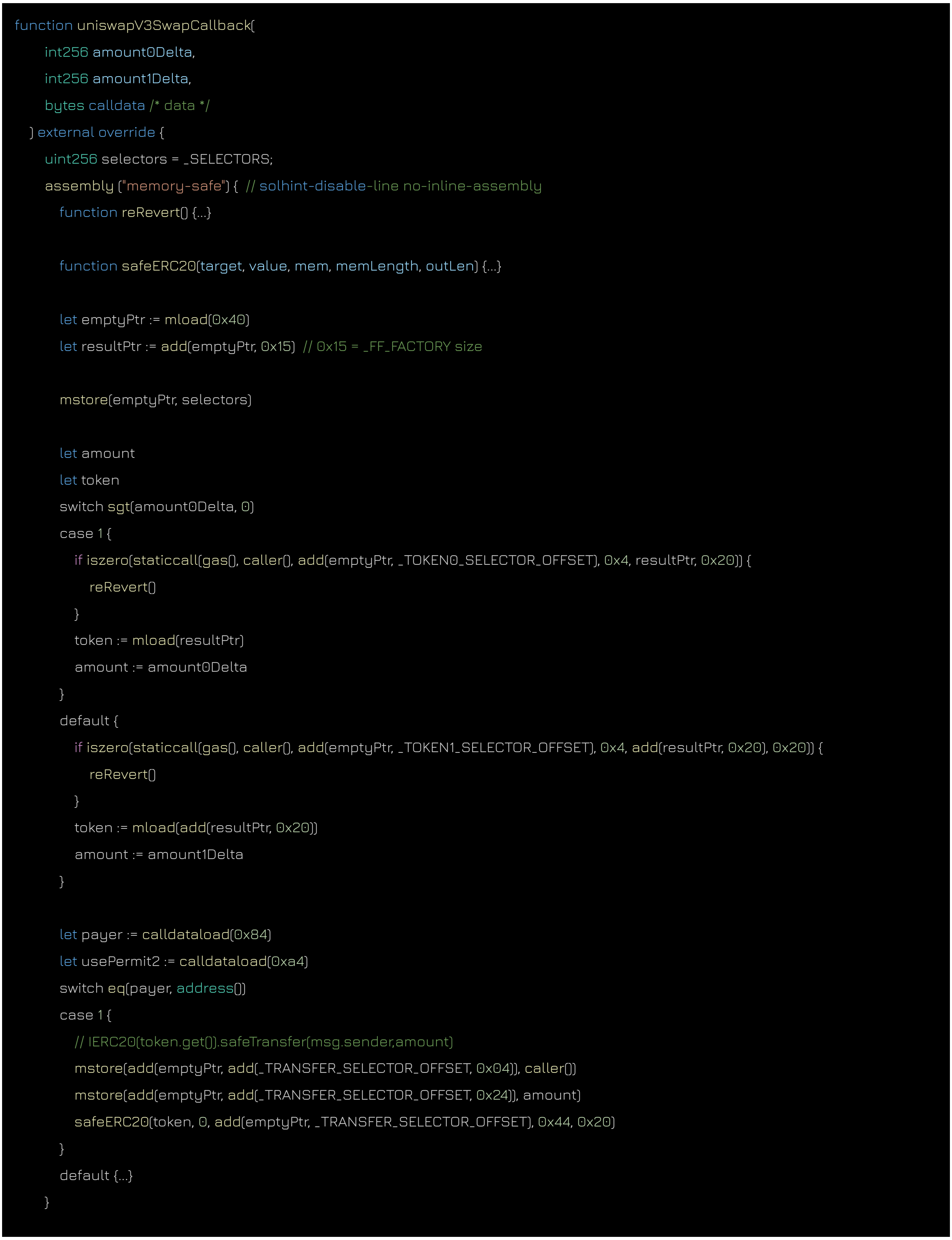
```
 /**
 * @notice Retrieves funds accidently sent directly to the contract address
 * @param token ERC20 token to retrieve
 * @param amount amount to retrieve
 */
function rescueFunds(IERC20 token, uint256 amount) external onlyOwner {
    token.uniTransfer(payable(msg.sender), amount);
}
```

Yet the permissionless **curveSwapCallback()** and **uniswapV3SwapCallback()** functions of **UnoswapRouter** exist, offering the same functionality. Anyone could call aforementioned functions to get stuck user's funds.



```
function curveSwapCallback(
    address /* sender */,
    address /* receiver */,
    address inCoin,
    uint256 dx,
    uint256 /* dy */
) external {
    IERC20(inCoin).safeTransfer(msg.sender, dx);
}
```

The **uniswapV3SwapCallback()** function behaves the same way as **curveSwapCallback()** when the **payer** parameter from calldata equals the address of the **UnoswapRouter** contract.



```
function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata /* data */
  ) external override {
    uint256 selectors = _SELECTORS;
    assembly ("memory-safe") {  // solhint-disable-line no-inline-assembly
      function reRevert() {...}

      function safeERC20(target, value, mem, memLength, outLen) {...}

      let emptyPtr := mload(0x40)
      let resultPtr := add(emptyPtr, 0x15)  // 0x15 = _FF_FACTORY size

      mstore(emptyPtr, selectors)

      let amount
      let token
      switch sgt(amount0Delta, 0)
      case 1 {
        if iszero(staticcall(gas(), caller(), add(emptyPtr, _TOKEN0_SELECTOR_OFFSET), 0x4, resultPtr, 0x20)) {
          reRevert()
        }
        token := mload(resultPtr)
        amount := amount0Delta
      }
      default {
        if iszero(staticcall(gas(), caller(), add(emptyPtr, _TOKEN1_SELECTOR_OFFSET), 0x4, add(resultPtr, 0x20), 0x20)) {
          reRevert()
        }
        token := mload(add(resultPtr, 0x20))
        amount := amount1Delta
      }

      let payer := calldataload(0x84)
      let usePermit2 := calldataload(0xa4)
      switch eq(payer, address())
      case 1 {
        // IERC20(token.get()).safeTransfer(msg.sender,amount)
        mstore(add(emptyPtr, add(_TRANSFER_SELECTOR_OFFSET, 0x04)), caller())
        mstore(add(emptyPtr, add(_TRANSFER_SELECTOR_OFFSET, 0x24)), amount)
        safeERC20(token, 0, add(emptyPtr, _TRANSFER_SELECTOR_OFFSET), 0x44, 0x20)
      }
      default {...}
    }
```

A possible attack scenario:

1. The user calls **unoswap2()** for Uniswap v3 protocol with correct **dex** but amiss **dex2** parameter.
2. The first swap uses address of the **UnoswapRouter** contract as the destination for Uniswap v3 protocol.
3. Since **dex2** is incorrect **_unoswap()** might execute without reverts if the protocol byte in **dex2** doesn't match any if statements.
4. Eventually, funds remain on the router contract balance.
5. A threat actor could get tokens by calling **curveSwapCallback** or **uniswapV3SwapCallback** directly.

INOCT-4. MAKING AN UNLIMITED APPROVAL FOR UNOSWAPROUTER TO ANY ADDRESS

SEVERITY: Medium

PATH: UnoswapRouter.sol

REMEDIATION: to mitigate the issue, asmApprove() should use the same token address as safeTransferFromUniversal() inside _unoswap()

STATUS:

DESCRIPTION:

In the current workflow for swapping on Curve, the user's funds are first transferred to the **UnoswapRouter**, and further **UnoswapRouter** approves token amount to the Curve pool inside **_curfe()**.

```
function _unoswap(
   address spender,
   address recipient,
   Address token,
   uint256 amount,
   uint256 minReturn,
   Address dex
) private returns(uint256 returnAmount) {
   ProtocolLib.Protocol protocol = dex.protocol();
   ...
   } else if (protocol == ProtocolLib.Protocol.Curve) {
       if (spender == msg.sender && msg.value == 0) {
           IERC20(token.get()).safeTransferFromUniversal(msg.sender, address(this), amount, dex.usePermit2());
       }
      returnAmount = _curfe(recipient, amount, minReturn, dex);
   }
}
```

Considering **pool**, **fromToken**, and **amount** are fully controlled by the user for **asmApprove()**:

1. **pool** comes from the **dex** input parameter.
2. **amount** is an input parameter.
3. **fromToken** is retrieved by calling **pool.coins()**.

Additionally, there is a mismatch between the **token** input parameter for **_unoswap()** and **fromToken** for **asmApprove()**, allowing to present a fake token for **_unoswap()** and a real one for **asmApprove()**.

Therefore, it's feasible for a threat actor to make unlimited approval from **UnoswapRouter** to a malicious contract for any token by calling **unoswap()** with selected input parameters. Eventually, any tokens on the router balance could be stolen. This might happen when someone transfers tokens to the router accidentally, or there is an issue during a multi-step swap (**unoswap2** or **unoswap3**).

```
function _curfe(
    address recipient,
    uint256 amount,
    uint256 minReturn,
    Address dex
) internal returns(uint256 ret) {
  ...
  if or(iszero(useEth), and(useEth, eq(toToken, _WETH))) {
      ...
      if iszero(hasCallback) {
          asmApprove(fromToken, pool, amount, mload(0x40))
      }
  }
  ...
}
```

INOCT-1. WETH CASE SHOULD BE HANDLED BY CLIPPEREXCHANGE.SWAP()

SEVERITY: Low

PATH: ClipperRouter.sol

REMEDIATION: see description

STATUS:

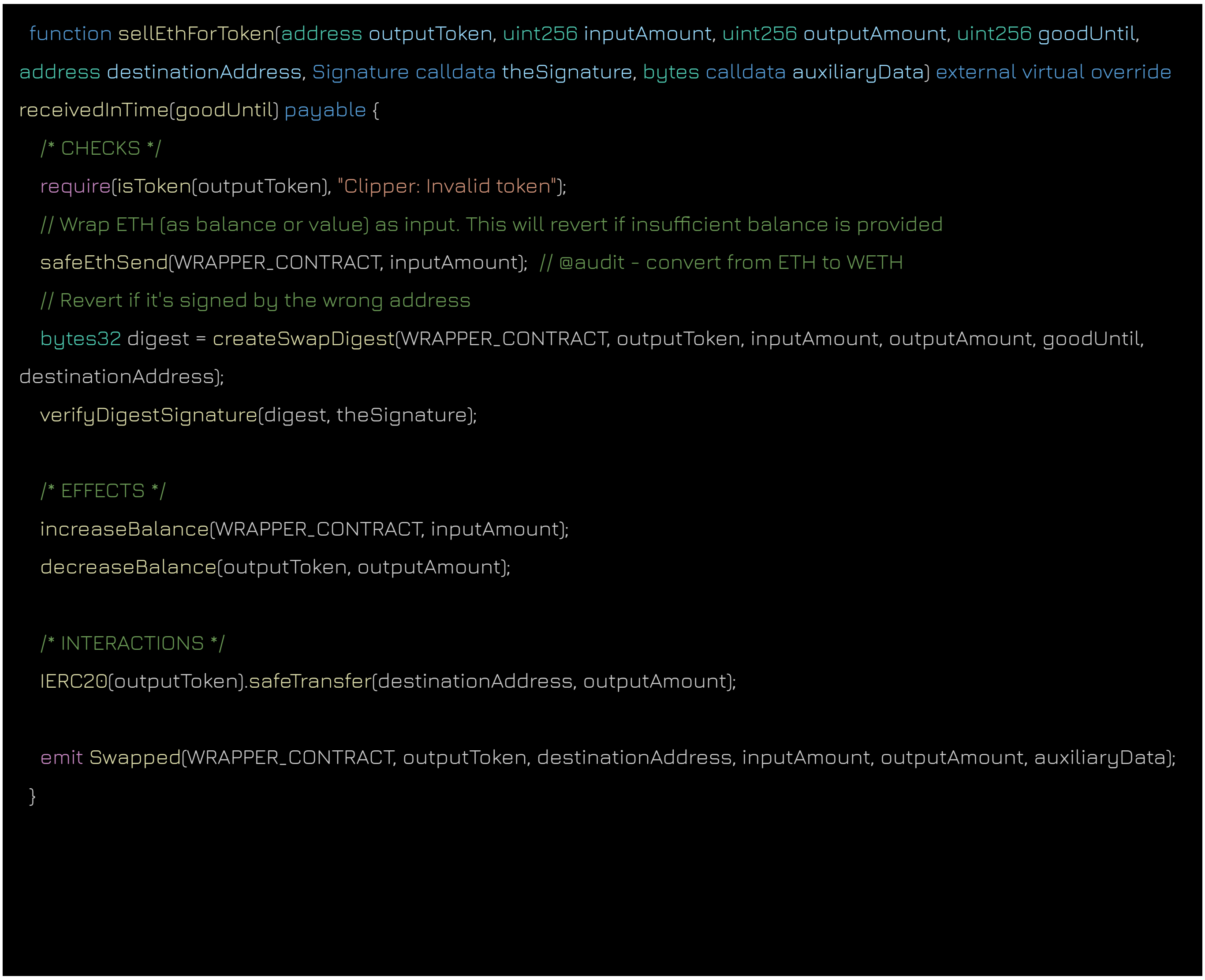
DESCRIPTION:

There is no need to check conditions srcToken_ == _WETH and dstToken == _WETH inside the if statement since WETH case is seamlessly handled by clipperExchange.swap(). Currently, when treating the WETH case specifically, it creates superfluous rounds of conversion between ETH and WETH.

In **srcToken_ == _WETH** case, **WETH** is converted to **ETH** in the begining of **clipperSwapTo()**.

```
function clipperSwapTo(
    IClipperExchange clipperExchange,
    address payable recipient,
    Address srcToken,
    IERC20 dstToken,
    uint256 inputAmount,
    uint256 outputAmount,
    uint256 goodUntil,
    bytes32 r,
    bytes32 vs
) public payable returns(uint256 returnAmount) {
    IERC20 srcToken_ = IERC20(srcToken.get());
    if (srcToken_ == _ETH) {
        if (msg.value != inputAmount) revert RouterErrors.InvalidMsgValue();
    } else if (srcToken_ == _WETH) {   // @audit from WETH to ETH
        if (msg.value != 0) revert RouterErrors.InvalidMsgValue();
        _WETH.safeTransferFrom(msg.sender, address(this), inputAmount);
        _WETH.safeWithdraw(inputAmount);
    } else {
        if (msg.value != 0) revert RouterErrors.InvalidMsgValue();
        srcToken_.safeTransferFromUniversal(msg.sender, address(clipperExchange), inputAmount,
srcToken.getFlag(_PERMIT2_FLAG));
    }
    ...
}
```

Yet **ClipperCaravelExchange.sellEthForToken()** (https://remix.ethereum.org/address/0xe7b0ce0526fbe3969035a145c9e9691d4d9d216c) converts **ETH** to **WETH** back. So, we have two conversions, but we can have 0 if **clipperExchange.swap()** is used.



```
 function sellEthForToken(address outputToken, uint256 inputAmount, uint256 outputAmount, uint256 goodUntil,
address destinationAddress, Signature calldata theSignature, bytes calldata auxiliaryData) external virtual override
receivedInTime(goodUntil) payable {
   /* CHECKS */
   require(isToken(outputToken), "Clipper: Invalid token");
   // Wrap ETH (as balance or value) as input. This will revert if insufficient balance is provided
   safeEthSend(WRAPPER_CONTRACT, inputAmount);  // @audit - convert from ETH to WETH
   // Revert if it's signed by the wrong address
   bytes32 digest = createSwapDigest(WRAPPER_CONTRACT, outputToken, inputAmount, outputAmount, goodUntil,
destinationAddress);
   verifyDigestSignature(digest, theSignature);

   /* EFFECTS */
   increaseBalance(WRAPPER_CONTRACT, inputAmount);
   decreaseBalance(outputToken, outputAmount);

   /* INTERACTIONS */
   IERC20(outputToken).safeTransfer(destinationAddress, outputAmount);

   emit Swapped(WRAPPER_CONTRACT, outputToken, destinationAddress, inputAmount, outputAmount, auxiliaryData);
 }
```

Moreover, case **dstToken == _WETH** requires **destinationAddress** pointing to the **ClipperRouter** contract instead of the user's address, which entails a user should request signed swap parameters from Clipper backend with different **destinationAddress** in such a case.

```
...
switch iszero(dstToken)
case 1 {
   mstore(add(ptr, 0x84), recipient)
}
default {
   mstore(add(ptr, 0x84), address())
}
...
```

INOCT-3. _CURFE() ALLOWS ANY FUNCTION SELECTOR FOR AN EXTERNAL CALL

SEVERITY: Low

PATH: UnoswapRouter.sol

REMEDIATION: see description

STATUS:

DESCRIPTION:

The **_curfe()** function allows to call any address on behalf of the **UnoswapRouter** with an arbitrary function selector both encoded into the dex parameter.

```
...
// Swap
let ptr := mload(0x40)
{  // stack too deep
   let swapSelector := shl(224, and(shr(_CURVE_SWAP_SELECTOR_OFFSET, dex), _CURVE_SWAP_SELECTOR_MASK))
   mstore(ptr, swapSelector)
}
```

The issue couldn't be exploited in a straightforward way by calling **transferFrom()** on the ERC20 token since the first two parameters are encoded into the **dex** input parameter and limited to **1 byte**.

```
...
mstore(add(ptr, 0x04), and(shr(_CURVE_FROM_TOKEN_OFFSET, dex), _CURVE_FROM_TOKEN_MASK))
mstore(add(ptr, 0x24), and(shr(_CURVE_TO_TOKEN_OFFSET, dex), _CURVE_TO_TOKEN_MASK))
...
```

Yet the third and forth parameters could have arbitrary values.

```
...
mstore(add(ptr, 0x44), amount)
mstore(add(ptr, 0x64), minReturn)
...
```

The recommendation is to employ a whitelist for the selector to mitigate future attacks, taking into account possible new integrations and router functionality.

INOCT-5. NATSPEC FOR TAKERTRAITSLIB LACKS DETAILS

SEVERITY: Informational

PATH: TakerTraitsLib.sol

REMEDIATION: see description

STATUS:

DESCRIPTION:

Natspec for **TakerTraitsLib** doesn't describe precisely all the flags and args packed into **TakerTraits**.

```
type TakerTraits is uint256;

/**
 * @title TakerTraitsLib
 * @notice This library to manage and check TakerTraits, which are used to encode the taker's preferences for an order in a single uint256.
 * @dev The TakerTraits are structured as follows:
 * High bits are used for flags
 * 255 bit `_MAKER_AMOUNT_FLAG`        - If set, the taking amount is calculated based on making amount, otherwise making amount is calculated based on taking amount.
 * 254 bit `_UNWRAP_WETH_FLAG`         - If set, the WETH will be unwrapped into ETH before sending to taker.
 * 253 bit `_SKIP_ORDER_PERMIT_FLAG`   - If set, the order skips maker's permit execution.
 * 252 bit `_USE_PERMIT2_FLAG`         - If set, the order uses the permit2 function for authorization.
 * The remaining bits are used to store the threshold amount (the maximum amount a taker agrees to give in exchange for a making amount).
 */
library TakerTraitsLib {
    uint256 private constant _MAKER_AMOUNT_FLAG = 1 << 255;
    uint256 private constant _UNWRAP_WETH_FLAG = 1 << 254;
    uint256 private constant _SKIP_ORDER_PERMIT_FLAG = 1 << 253;
    uint256 private constant _USE_PERMIT2_FLAG = 1 << 252;
    uint256 private constant _ARGS_HAS_TARGET = 1 << 251;

    uint256 private constant _ARGS_EXTENSION_LENGTH_OFFSET = 224;
    uint256 private constant _ARGS_EXTENSION_LENGTH_MASK = 0xffffff;
    uint256 private constant _ARGS_INTERACTION_LENGTH_OFFSET = 200;
    uint256 private constant _ARGS_INTERACTION_LENGTH_MASK = 0xffffff;

    uint256 private constant _AMOUNT_MASK = 0x000000000000000000ffffffffffffffffffffffffffffffffffffffffffffff;
    ...
}
```

INOCT-6. UNUSED FUNCTIONALITY FROM INHERITED ORDERMIXIN

SEVERITY: Informational

PATH: AggregationRouterV6.sol

REMEDIATION: see description

STATUS:

DESCRIPTION:

AggregationRouterV6 inherits **OrderMixin** functionality while only the **permitAndCall()** function is required for the router. We recommend decoupling the rest of **OrderMixin** functionality from **AggregationRouterV6** since it's unused and exposes an additional attack surface for the router.

```
contract AggregationRouterV6 is EIP712("1inch Aggregation Router", "6"), Ownable,
    ClipperRouter, GenericRouter, UnoswapRouter, OrderMixin {
    ...
}
```

hexens