ZeroEx - Permit2Payment

Security Assessment (Summary Report)

June 5, 2024

Prepared for:
Eric Wong, Duncan Townsend
ZeroEx

Prepared by: **Elvis Skoždopolj and Kurt Willis**

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:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Project Timeline

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

Date	Event
May 29, 2024	Pre-project kickoff call
June 10, 2024	Delivery of report draft
June 10, 2024	Report readout meeting

Project Targets

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

Permit2Payment

Repository	https://github.com/0xProject/0x-settler
Version	`f6e15f6`
Type	Solidity
Platform	Ethereum

Executive Summary

Engagement Overview

ZeroEx engaged Trail of Bits to review the security of the `Permit2Payment` smart contracts. These contracts are meant to be inherited by the main contracts, `Settler` and `SettlerMetaTx`, that enable executing optimized and configurable batched actions. The `Permit2Payment` contracts implement internal functions and libraries which manage how transient storage values are set, checked, and cleared, how calls to `Permit2` and the `AllowanceHolder` contracts are made, and how callbacks are validated and executed. The main goal of the contracts is to securely handle token allowances and permits via calls and callbacks performed through the `0xSettler` protocol.

A team of 1 consultant conducted the review from May 30 to June 5, for a total of 1 engineer-week of effort. With full access to source code and documentation, we performed static and dynamic testing of the target, using automated and manual processes.

Observations and Impact

The review was focused on checking if the `Permit2Payment` contracts can correctly and securely handle calls related to token transfers and callbacks performed through the `0xSettler` protocol or any valid target of the protocol, with a focus on the correct handling of `Permit2` calls. We reviewed how transient storage slots are set, cleared, and validated in order to ensure that “old” values cannot be misused, that the slots cannot be arbitrarily modified, and that reentrancies are properly mitigated. Additionally, we checked that the assembly is correct, that masking cannot truncate important values such as the internal function pointers, that values are correctly packed inside the operator slot, and that arbitrary callbacks cannot be performed.

Interactions with other contracts in the codebase were reviewed to gain context on the usage of `Permit2Payment`, however, they were not the main focus of the review. Findings listed in the previous `0xSettler` security review report were omitted.

During the review we have not identified any findings that currently have a direct impact on the security of the codebase, although one informational-severity issue was identified (TOB-0XP2-1) that should be considered prior to deploying the contracts on new chains. The codebase uses advanced low-level optimization techniques and new Solidity opcode features, minimizing unnecessary checks and optimizing gas costs. While this is currently implemented securely, it does make the system more difficult to review and maintain. Additionally, it poses a risk that minor and difficult to find mistakes, and as of yet undiscovered compiler bugs, could impact the security of the protocol.

Recommendations

Due to the heavy use of assembly and new Solidity opcodes we recommend that the ZeroEx team simplify the assembly heavy code, documented all unwritten assumptions (e.g., the internal function pointer size assumption), and regularly check for compiler bugs that could impact the features being used in the codebase.

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 target contracts do not use any arithmetic operations.	**Not Applicable**
Auditing	The target contracts do not emit any events, this is left to the inheriting contracts.	**Not Applicable**
Authentication / Access Controls	The target contracts implement access control and reentrancy protection based on the value of transient storage slots. The `Permit2` interactions on behalf of users other than the caller are performed via `SettlerMetaTx` and are protected by calculating the witness parameter in the contract. Arbitrary callbacks are prevented by setting the function selector, internal function pointer, and the expected caller in a transient storage slot, with the callback data being tightly controlled.	**Satisfactory**
Complexity Management	The callback mechanism makes use of transient storage slots, packed values, and internal function pointers to determine which callback function will be triggered and provide access control. While this provides additional flexibility it also creates additional overhead when reviewing the code and could be error prone. While the project makes extensive use of inline assembly and heavy optimizations, the `Permit2Payment` contracts have a relatively simple design compared to the rest of the system.	**Moderate**
Decentralization	The target contract and the protocol are permissionless and no privileged roles or actions exist. The contracts are not upgradeable and have no configuration parameters.	**Strong**

Documentation	The project documentation includes diagrams that detail various user flows and expected contract interactions. The inline documentation is comprehensive, however, the documentation of some assembly blocks and inherent system assumptions could be improved.	**Satisfactory**
Low-Level Manipulation	The `Permit2Payment` contracts use low-level manipulation for setting, fetching, and clearing transient storage slots, as well as packing different values into a single slot. The system is designed with the goal of minimizing gas consumption and some checks are skipped due to being checked elsewhere in the code, or due to being considered unnecessary. The use of new Solidity features and low-level manipulation increases the chance that a small mistake made during future development could have significant security implications.	**Moderate**
Testing and Verification	The testing suite makes heavy use of mocked calls which lowers the effectiveness of the tests and could potentially prevent issues from being discovered. Mutation testing indicates that gaps exist in the coverage of the testing suite (for more information check appendix C).	**Moderate**
Transaction Ordering	We did not uncover any front-running, back-running, or sandwich-related issues.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Internal function pointer masking could become insufficient in the future	Data Validation	**Informational**

1. Internal function pointer masking could become insufficient in the future

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-0XP2-1
Target: `src/core/Permit2Payment.sol`	

Description

When packing the internal function pointer value into the `operator` transient storage slot, the `callback` variable's higher bits are cleaned by using a bitwise `AND` operation, preserving the least significant two bytes of data.

```
function setOperatorAndCallback(
    address operator,
    uint32 selector,
    function (bytes calldata) internal returns (bytes memory) callback
) internal {
    // …
    assembly ("memory-safe") {
        tstore(
            _OPERATOR_SLOT,
            or(
                shl(0xe0, selector),
                or(shl(0xa0, and(0xffff, callback)),
and(0xffffffffffffffffffffffffffffffffffffffff, operator))
            )
        )
    }
```

Figure 1.1: Packing of the callback pointer into the operator transient storage slot (`0x-settler/src/core/Permit2Payment.sol#57–64`)

Internal function pointers are encoded as jump locations in the contract code. EIP-170 (Spurious Dragon) introduces a maximum contract code size limit of 24KB (`0x6000` in hexadecimal representation), this means that 2 bytes are enough to store any internal function pointer on Ethereum for now. However, it might be the case that this limit is changed in the future. Other networks might also opt for a higher contract size limit. If the code is deployed to a network with a higher contract size limit this code might not perform as expected.

Recommendations

Short term, consider using 4 bytes of data for the internal function pointer.

Long term, document the assumptions around the byte masking for internal function pointers and ensure they are checked for each deployment to a new chain.

A. 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.

B. Code Quality Issues

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

- The `setOperatorAndCallback` and `setWitness` function of the `TransientStorage` library do not check that the inputs are not zero. While these parameters in their current use cannot be zero due to their derivation, the assumptions should be clearly documented.

```
function setOperatorAndCallback(
    address operator,
    uint32 selector,
    function (bytes calldata) internal returns (bytes memory) callback
) internal {
    // ...

    assembly ("memory-safe") {
        tstore(
            _OPERATOR_SLOT,
            or(
                shl(0xe0, selector),
                or(shl(0xa0, and(0xffff, callback)),
and(0xffffffffffffffffffffffffffffffffffffffff, operator))
            )
        )
    }
}
```

Figure B.1: The `setOperatorAndCallback` function does not perform a zero check (`0x-settler/src/core/Permit2Payment.sol#35–66`)

- The constants `GENERIC`, `ASSERT_FAIL`, `CORRUPT_STORAGE_ARRAY`, `POP_EMPTY_ARRAY`, `ZERO_FUNCTION_POINTER` from the `Panic` library are never used.

```
library Panic {

    // ...

    uint8 internal constant GENERIC = 0x00;
    uint8 internal constant ASSERT_FAIL = 0x01;
    uint8 internal constant ARITHMETIC_OVERFLOW = 0x11;
    uint8 internal constant DIVISION_BY_ZERO = 0x12;
    uint8 internal constant ENUM_CAST = 0x21;
    uint8 internal constant CORRUPT_STORAGE_ARRAY = 0x22;
    uint8 internal constant POP_EMPTY_ARRAY = 0x31;
    uint8 internal constant ARRAY_OUT_OF_BOUNDS = 0x32;
```

```
    uint8 internal constant OUT_OF_MEMORY = 0x41;
    uint8 internal constant ZERO_FUNCTION_POINTER = 0x51;
}
```

Figure B.2: The Panic library defines unused error constants (0x-settler/src/utils/Panic.sol#4–24)

- The `getAndClearWitness` function in the `TransientStorage` library uses decimal notation for zero, while the `clearPayer` and `getAndClearOperatorAndCallback` functions use hexadecimal notation.

```
function getAndClearWitness() internal returns (bytes32 witness) {
    assembly ("memory-safe") {
        witness := tload(_WITNESS_SLOT)
        tstore(_WITNESS_SLOT, 0)
    }
}
```

Figure B.3: The `getAndClearWitness` function (0x-settler/src/core/Permit2Payment.sol#115–120)

```
function clearPayer(address expectedOldPayer) internal {
    address oldPayer;
    assembly ("memory-safe") {
        oldPayer := tload(_PAYER_SLOT)
    }
    if (oldPayer != expectedOldPayer) {
        revert PayerSpent();
    }
    assembly ("memory-safe") {
        tstore(_PAYER_SLOT, 0x00)
    }
}
```

Figure B.4: The `clearPayer` function (0x-settler/src/core/Permit2Payment.sol#144–155)

C. Mutation Testing

This appendix outlines how we conducted mutation testing and highlights some of the most actionable results.

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in the test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We used an experimental new mutation tool, `slither-mutate`, to conduct our mutation testing campaign. This tool is custom-made for Solidity and features higher performance and fewer false positives than existing tools such as `universalmutator`.

```
python3 -m pip install slither-analyzer
```

Figure C.1: The command that installs `slither` using `pip`

The mutation campaign was run against the smart contracts using the following commands:

```
slither-mutate . --test-cmd='forge test' --test-dir='test'
--ignore-dirs='script,lib,test,node_modules,deployer,multicall,vendor' --timeout 120
```

Figure C.2: A bash script that runs a mutation testing campaign against each Solidity file in the `src` directory (unit tests)

```
slither-mutate . --test-cmd='FOUNDRY_PROFILE=integration forge test'
--test-dir='test'
--ignore-dirs='script,lib,test,node_modules,deployer,multicall,vendor' --timeout 130
```

Figure C.3: A bash script that runs a mutation testing campaign against each Solidity file in the `src` directory (integration tests)

Consider the following notes about the above commands:

- On a consumer-grade laptop, the overall runtime of the mutation testing campaign is approximately eleven hours.
- The `--test-cmd` flags specify the command to run to assess mutant validity. A `--fail-fast` or `--bail` flag will automatically be added to test commands to improve runtime.

An abbreviated, illustrative example of a mutation test output file is shown in figure D.3.

```
INFO:Slither-Mutate:Mutating contract UniswapV3Fork
INFO:Slither-Mutate:[RR] Line 121: 'Panic.panic(Panic.ARITHMETIC_OVERFLOW)' ==> 'revert()' -->
UNCAUGHT
INFO:Slither-Mutate:[RR] Line 134: '(token0, token1) = (token1, token0)' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 144: '_updateSwapCallbackData(swapCallbackData, token1, payer)' ==>
'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 158: 'tempPrice = MIN_PRICE_SQRT_RATIO + 1' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 160: 'tempPrice = MAX_PRICE_SQRT_RATIO - 1' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 162: '(amount0, amount1) = abi.decode(
                    _setOperatorAndCall(
                        address(pool),
                        abi.encodeCall(
                            pool.swap,
                            (
                                // Intermediate tokens go to this contract.
                                address(this),
                                zeroForOne,
                                int256(sellAmount),
                                tempPrice,
                                swapCallbackData
                            )
                        ),
                        uint32(callbackSelector),
                        _uniV3ForkCallback
                    ),
                    (int256, int256)
                )' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 189: 'tempPrice = MAX_PRICE_SQRT_RATIO - 1' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 216: '_buyAmount = -(amount0)' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 219: 'Panic.panic(Panic.ARITHMETIC_OVERFLOW)' ==> 'revert()' -->
UNCAUGHT
INFO:Slither-Mutate:[RR] Line 228: 'payer = address(this)' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 229: 'sellAmount = buyAmount' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 231: 'encodedPath = _shiftHopFromPathInPlace(encodedPath)' ==>
'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[CR] Line 154: 'freeMemPtr := mload(0x40)' ==> '//freeMemPtr := mload(0x40)' -->
UNCAUGHT
INFO:Slither-Mutate:[CR] Line 182: 'mstore(0x40, freeMemPtr)' ==> '//mstore(0x40, freeMemPtr)' -->
UNCAUGHT
INFO:Slither-Mutate:[CR] Line 233: 'mstore(swapCallbackData, SWAP_CALLBACK_PREFIX_DATA_SIZE)' ==>
'//mstore(swapCallbackData, SWAP_CALLBACK_PREFIX_DATA_SIZE)' --> UNCAUGHT
INFO:Slither-Mutate:[FHR] Line 112: 'function _uniV3ForkSwap(
        address recipient,
        bytes memory encodedPath,
        uint256 sellAmount,
        uint256 minBuyAmount,
        address payer,
        bytes memory swapCallbackData
    ) internal returns (uint256 buyAmount) ' ==> 'function _uniV3ForkSwap(
        address recipient,
        bytes memory encodedPath,
        uint256 sellAmount,
        uint256 minBuyAmount,
        address payer,
        bytes memory swapCallbackData
    ) private returns (uint256 buyAmount) ' --> UNCAUGHT
INFO:Slither-Mutate:[FHR] Line 242: 'function _isPathMultiHop(bytes memory encodedPath) private pure
returns (bool) ' ==> 'function _isPathMultiHop(bytes memory encodedPath) private view returns (bool)
' --> UNCAUGHT
...
```

Figure C.4: Abbreviated output from the mutation testing campaign assessing test coverage of the in-scope contracts

Unit tests

The following table displays the portion of each type of mutant for which all unit tests passed. The presence of valid mutants indicates that there are gaps in test coverage because the test suite did not catch the introduced change.

Contracts supporting tests, contracts that produced zero analyzed mutants (e.g., interfaces), and contracts for which mutation testing failed were omitted from mutation testing analysis.

We recommend running the commands in figures C.2 and C.3 to get a full list of uncaught mutants.

Target (Unit tests)	Uncaught Reverts	Uncaught Comments	Uncaught Tweaks
`SettlerMetaTx`	100% (13/13)	100% (13/13)	None analyzed
`Settler`	0% (0/13)	0% (0/16)	81.6% (31/38)
`CurveTricrypto`	100% (6/6)	100% (26/26)	None analyzed
`MakerPSM`	0% (0/5)	20% (1/5)	14.3% (3/21)
`UniswapV3Fork`	50% (12/24)	23% (3/13)	46% (37/80)
`Permit2Payment`	14.3% (1/7)	0% (0/7)	33.33% (1/3)
`RfqOrderSettlement`	10% (1/10)	0% (0/16)	None analyzed

The following is a summary of the mutations that were not caught by the unit testing suite:

- `SettlerMetaTx`

- Replacing lines 29, 37, 48, 112, 121, 130, 132, 134, 143, 155, 162, or 163 with ‘`revert()`’.
- Commenting out lines 33, 64, 65, 69, 70, 72, 88, 89, 90, 91, 151, or 158.

- `Settler`
  - Replacing line 90 with `false` or negating the condition (e.g., `!condition`).
  - Replacing line 144 with `true` or `false`.
  - Replacing the operator == with `<`, `>`, `<=`, `>=`, or `!=` at lines 69, 90, 112, or 114.
- `CurveTricrypto`
  - Replacing lines 88, 109, 121, 122, 128, or 166 with ‘`revert()`’.
  - Commenting out lines 71, 72, 73, 79, 80, 84, 85, 114, 116, 117, 118, 119, 134, 135, 136, 137, 138, 139, 140, 146, 147, 148, 152, 153, 155, 156.
- `MakerPSM`
  - Commenting out line 41.
  - Updating the variable on line 36 to `immutable` or to the `int128` type.
  - Updating the variable on line 38 with `immutable`.
- `UniswapV3Fork`
  - Replacing lines 121, 134, 144, 158, 160, 162, 189, 216, 219, 228, 229, or 231 with ‘`revert()`’.
  - Commenting out lines 154, 182, or 233.
  - Replacing lines 120, 141, 151, 157, 186, 213, 218, 223 with `true`, `false`, or with a negated condition.
  - Updating the > operator on line 120 to >= or ==.
  - Updating the operator < on line 133 with <= or `!=`.
  - Updating the operator < on line 218 with <= or `==`.
  - Updating the operator `<` on line 236 with `!=`.
  - Updating uint256 to uint128 on lines 41, 44, 46, 48, 49, 50, 58, or 152.

- Updating line 120 'sellAmount > uint256(type(int256).max)' to 'sellAmount > uint128(type(int128).max)'

- `Permit2Payment`
    - Replacing line 255 with 'revert()'.
    - Replacing line 242 with `false`.
- `RfqOrderSettlement`
    - Replacing line 145 with 'revert()'.

Integration tests

The following table displays the portion of each type of mutant for which all integration tests passed.

Target (Integration tests)	Uncaught Reverts	Uncaught Comments	Uncaught Tweaks
`SettlerMetaTx`	15.4% (2/13)	39.4% (7/23)	None analyzed
`SettlerAbstract`	0% (0/1)	100% (0/1)	0% (0/2)
`SettlerBase`	16.6% (1/6)	80% (8/10)	0% (0/2)
`Context`	33.3% (1/3)	50% (1/2)	33.3% (1/3)
`Settler`	0% (0/13)	56.2% (9/16)	18.4% (7/38)
`MakerPSM`	80% (4/5)	100% (1/1)	90.4% (19/21)
`CurveTricrypto`	0% (0/6)	41.9% (13/31)	None analyzed

The following is a summary of the mutations that were not caught by the integration testing suite:

- `SettlerMetaTx`
  - Replacing lines 37 or 132 with '`revert()`'.
  - Commenting out lines 33, 143, 151, 158, 162, or 163.
- `SettlerAbstract`
  - Commenting out line 14.
- `SettlerBase`
  - Commenting out lines 58, 59, 80, or 86.
- `Context`
  - Replacing line 18 with '`revert()`'.
  - Commenting out line 22.
- `Settler`
  - Commenting out lines 18, 31, 33, 65, 89, 99, 115, 120, or 162.
  - Updating the operator == on line 24 to <= or `>=`.
  - Updating the operator == on line 90 to `<=`.
  - Updating the operator == on line 114 to <= or `>=`.
- `MakerPSM`
  - Replacing lines 47, 48, 59, or 60 with 'revert()'.
  - Replacing the operator * with +, /, -, or % on line 46 or line 53.
  - Replacing the operator + with /, -, *, or % on line 55.
  - Removing variable assignment on lines 46, 53, 55, or 57.
- `CurveTricrypto`
  - Commenting out lines 71, 109, 114, 117, 119, 122, 128, 134, 135, 137, 139, 140, or 156.

Recommendations

We recommend that the ZeroEx team review the existing tests and add additional verification that would catch the aforementioned types of mutations. Then, use a script similar to the ones provided in figure C.2 and figure C.3 to rerun a mutation testing campaign to ensure that the added tests provide adequate coverage.