Meson Protocol

Fix Review

October 3, 2022

Prepared for:
Phil Li and Edrick Yuhui Guan
Meson

Prepared by: **Tjaden Hess**

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.
228 Park Ave S #80688
New York, NY 10003
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.

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.

Table of Contents

Executive Summary

Engagement Overview

Meson engaged Trail of Bits to review the security of its Meson protocol. From August 22 to September 9, 2022, a team of three consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

Meson contracted Trail of Bits to review the fixes implemented for issues identified in the original report. From September 21 to September 23, 2022, one consultant conducted a review of the client-provided source code, with three person-days of effort.

Summary of Findings

The original audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the original findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	2
Low	1
Informational	7
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	1
Cryptography	3
Data Validation	3
Denial of Service	2
Testing	1
Undefined Behavior	4

Overview of Fix Review Results

Meson has sufficiently addressed most of the issues described in the original audit report.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.o'brien@trailofbits.com

The following engineer was associated with this project:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Project Timeline

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

Date	Event
August 17, 2022	Pre-project kickoff call
August 29, 2022	Status update meeting #1
September 6, 2022	Status update meeting #2
September 9, 2022	Delivery of report draft; report readout meeting
September 27, 2022	Delivery of fix review draft
October 3, 2022	Delivery of public fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material

Project Targets

The engagement involved a review of the fixes implemented in the following target.

Meson Protocol

Repository	https://github.com/MesonFi/meson-contracts-solidity
Version	`334a67fd31cd09028214cbd274fb27aa637e4c70`
Type	Solidity
Platform	Ethereum

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Hash collisions in untyped signatures	Resolved
2	Typed signatures implement insecure nonstandard encodings	Unresolved
3	Missing validation in the _addSupportToken function	Resolved
4	Insufficient event generation	Resolved
5	Use of an uninitialized state variable in functions	Resolved
6	Risk of upgrade issues due to missing __gap variable	Resolved
7	Lack of a zero-value check on the initialize function	Partially Resolved
8	Solidity compiler optimizations can be problematic	Resolved
9	Service fees cannot be withdrawn	Resolved
10	Lack of contract existence check on transfer / transferFrom calls	Resolved
11	USDT transfers to third-party contracts will fail	Resolved
12	SDK function _randomHex returns low-quality randomness	Partially Resolved

13	encodedSwap values are used as primary swap identifier	Unresolved
14	Unnecessary _releasing mutex increases gas costs	Resolved
15	Misleading result returned by view function getPostedSwap	Resolved

Detailed Findings

1. Hash collisions in untyped signatures

Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-MES-1
Target: `contracts/utils/MesonHelpers.sol`	

Description

To post or execute a swap, a user must provide an ECDSA signature on a message containing the encoded swap information. The Meson protocol supports both typed (EIP-712) and legacy untyped (EIP-191) messages. The format of a message is determined by a bit in the encoded swap information itself. The Meson protocol defines two message types, a "request message" containing only an encoded swap and a "release message" containing the hash of an encoded swap concatenated with the recipient's address.

Figure 1.1 shows the relevant signature-verification code.

```
  function _checkRequestSignature(
...
   if (nonTyped) {
     bytes32 digest = keccak256(abi.encodePacked(
       bytes28(0x19457468657265756d205369676e6564204d6573736167653a0a3332), // HEX of "\x19Ethereum Signed Message:\n32"
       encodedSwap
     ));
     require(signer == ecrecover(digest, v, r, s), "Invalid signature");
     return;
   }
...
  function _checkReleaseSignature(
...
   if (nonTyped) {
     digest = keccak256(abi.encodePacked(
       bytes28(0x19457468657265756d205369676e6564204d6573736167653a0a3332), // HEX of "\x19Ethereum Signed Message:\n32"
       keccak256(abi.encodePacked(encodedSwap, recipient))
     ));
...
```

Figure 1.1: `contracts/utils/MesonHelpers.sol`

Note that the form of both the request and release messages in the figure is `"\x19Ethereum Signed Message:\n32" + msg`, where `msg` is a 32-byte string. If an attacker could find a message that would be interpreted as valid in both contexts, the attacker could use the signature on that message to both request and release funds, facilitating a number of potential attacks.

Specifically, the attacker would need to identify `swap1`, `swap2`, and `recipient` values such that `swap1 = keccak256(swap2, recipient)`.

The attacker could do that by choosing a valid `swap2` value and then iterating through `recipient` values until finding one for which `keccak256(swap2, recipient)` would be interpreted as a valid message. With the current restrictions on the swap amount, chain, and token fields, we estimate that this would take between 2^{60} and 2^{70} tries.

Fix Analysis

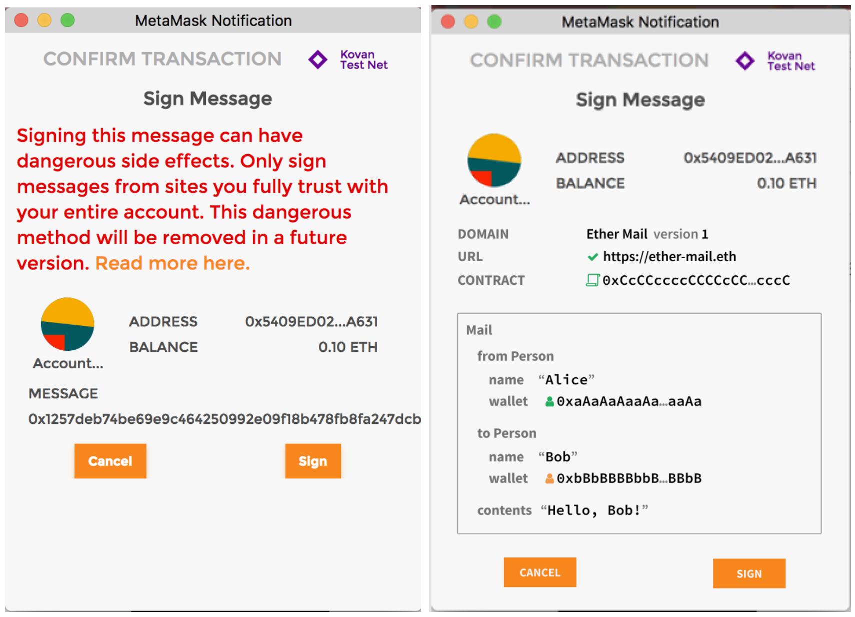
This issue has been resolved. Untyped release messages are now prefixed by the string `"\x19Ethereum Signed Message:\n52"`, while request messages are prefixed by `"\x19Ethereum Signed Message:\n32"`. However, if Meson ever introduces new message types with a length of 32 or 53 bytes, their encodings may collide with the encodings of the existing message types.

2. Typed signatures implement insecure nonstandard encodings	
Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-MES-2
Target: `contracts/utils/MesonHelpers.sol`	

Description

EIP-712 specifies standard encodings for the hashing and signing of typed structured data. The goal of typed structured signing standards is twofold: ensuring a unique injective encoding for structured data in order to prevent collisions (like that detailed in TOB-MES-1) and allowing wallets to display complex structured messages unambiguously in human-readable form.

The images in figure 2.1 demonstrate the difference between a complex untyped unstructured message (left) and its EIP-712 equivalent (right), both in MetaMask:



Signing this message can have dangerous side effects. Only sign messages from sites you fully trust with your entire account. This dangerous method will be removed in a future version. Read more here.

Account...
ADDRESS 0x5409ED02...A631
BALANCE 0.10 ETH
MESSAGE
0x1257deb74be69e9c464250992e09f18b478fb8fa247dcb

Account...
ADDRESS 0x5409ED02...A631
BALANCE 0.10 ETH
DOMAIN Ether Mail version 1
URL https://ether-mail.eth
CONTRACT 0xCcCCccccCCCCcCC...cccC

Mail
from Person
name "Alice"
wallet 0xaAaAaAaaAa...aaAa
to Person
name "Bob"
wallet 0xbBbBBBBbbB...BBbB
contents "Hello, Bob!"

Figure 2.1: A reproduction of images from the EIP-712 standard

Meson currently uses a form of typed message encoding that does not conform to EIP-712. Specifically, the encoding is not EIP-191 compliant and thus could theoretically collide with the encoding of personal messages (Ethereum signed messages) or Recursive Length Prefix (RLP)-encoded transactions.

The digest format for swap requests is included in figure 2.2, in which `REQUEST_TYPE_HASH` corresponds to `keccak256("bytes32 Sign to request a swap on Meson (Testnet)")`.

```
bytes32 typehash = REQUEST_TYPE_HASH;
bytes32 digest;
assembly {
 mstore(0, encodedSwap)
 mstore(32, keccak256(0, 32))
 mstore(0, typehash)
 digest := keccak256(0, 64)
}
```

Figure 2.2: `contracts/utils/MesonHelpers.sol#246–253`

While the message types currently used in the protocol do not appear to have any dangerous interactions with each other, message types added to future versions of the protocol could theoretically introduce such issues.

Fix Analysis

This issue has not been resolved. Although the issue is not currently exploitable, we recommend that Meson exercise caution when adding new message types to prevent unexpected collisions between those message types and message types used by other protocols.

3. Missing validation in the _addSupportToken function

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MES-3
Target: `contracts/utils/MesonTokens.sol`	

Description

Insufficient input validation in the `_addSupportToken` function makes it possible to register the same token as supported multiple times. This does not cause a problem, because if there are duplicate entries for a token in the token list, the last one added will be the one that is used. However, it does mean that multiple indexes could point to the same token, while the token would point to only one of those indexes.

```
  function _addSupportToken(address token, uint8 index) internal {
   require(index != 0, "Cannot use 0 as token index");
   _indexOfToken[token] = index;
   _tokenList[index] = token;
  }
```

Figure 3.1: `contracts/utils/MesonTokens.sol`

Fix Analysis

This issue has been resolved. The `_addSupportToken` function now validates that the token has not previously been registered, that the associated list index has not previously been used, and that the token's address is not zero. The Meson team has also added tests to validate this behavior.

4. Insufficient event generation	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-MES-4
Target: `contracts/Pools/MesonPools.sol`	

Description

Several critical operations in the `MesonPools` contract do not emit events. As a result, it will be difficult to review the correct behavior of the contract once it has been deployed.

The following operations should trigger events:

- `MesonPools.depositAndRegister`
- `MesonPools.deposit`
- `MesonPools.withdraw`
- `MesonPools.addAuthorizedAddr`
- `MesonPools.removeAuthorizedAddr`
- `MesonPools.unlock`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior and may therefore overlook attacks or malfunctioning contracts.

Fix Analysis

This issue has been resolved. All of the functions listed in this finding now emit events, enabling Meson and protocol users to easily track all contract operations.

5. Use of an uninitialized state variable in functions

Status: **Resolved**	
Severity: **Medium**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-MES-5
Target: `contracts/Token/UCTUpgradeable.sol`	

Description

The `_mesonContract` address is not set in the `UCTUpgradeable` contract's `initialize` function during the contract's initialization. As a result, the value of `_mesonContract` defaults to the zero address.

The `UCTUpgradeable.allowance` and `UCTUpgradeable.transferFrom` functions perform checks that rely on the value of the `_mesonContract` state variable, which may lead to unexpected behavior.

```
18    address private _mesonContract;
19
20    function initialize(address minter) public initializer {
21      __ERC20_init("USD Coupon Token (https://meson.fi)", "UCT");
22      _owner = _msgSender();
23      _minter = minter;
24      // _mesonContract = ;
25    }
```

Figure 5.1: `contracts/Token/UCTUpgradeable.sol:18–25`

```
54    function allowance(address owner, address spender) public view override
returns (uint256) {
55      if (spender == _mesonContract) {
```

Figure 5.2: `contracts/Token/UCTUpgradeable.sol:54–55`

```
65    if (msgSender == _mesonContract && ERC20Upgradeable.allowance(sender,
msgSender) < amount) {
```

Figure 5.3: `contracts/Token/UCTUpgradeable.sol:65`

Fix Analysis

This issue has been resolved. The `_mesonContract` address is now populated by the `initialize` function.

6. Risk of upgrade issues due to missing __gap variable	
Status: **Resolved**	
Severity: **High**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-MES-6
Target: `contracts/**/*.sol`	

Description

None of the Meson protocol contracts include a `__gap` variable. Without this variable, it is not possible to add any new variables to the inherited contracts without causing storage slot issues. Specifically, if variables are added to an inherited contract, the storage slots of all subsequent variables in the contract will shift by the number of variables added. Such a shift would likely break the contract.

All upgradeable OpenZeppelin contracts contain a `__gap` variable, as shown in figure 6.1.

```
/**
 * @dev This empty reserved space is put in place to allow future versions to add new
 * variables without shifting down storage in the inheritance chain.
 * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
 */
uint256[49] private __gap;
```

Figure 6.1: `openzeppelin-contracts-upgradeable/OwnerUpgradeable.sol`

Fix Analysis

This issue has been resolved. All stateful contracts inherited by `UpgradableMeson` now contain gap slots. Thus, new state variables can be added in future upgrades.

7. Lack of a zero-value check on the initialize function	
Status: **Partially Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MES-7
Target: `contracts/Token/UCTUpgradeable.sol`	

Description

The `UCTUpgradeable` contract's `initialize` function fails to validate the address of the incoming `minter` argument. This means that the caller can accidentally set the `minter` variable to the zero address.

```
function initialize(address minter) public initializer {
  __ERC20_init("USD Coupon Token (https://meson.fi)", "UCT");
  _owner = _msgSender();
  _minter = minter;
  // _mesonContract = ;
}
```

Figure 7.1: `contracts/Token/UCTUpgradeable.sol:20–25`

If the `minter` address is set to the zero address, the admin must immediately redeploy the contract and set the address to the correct value; a failure to do so could result in unexpected behavior.

Fix Analysis

This issue has been partially resolved. The `_mesonContract` address, added as a parameter in the resolution of TOB-MES-5, is now checked against the zero value. However, the `initialize` function does not validate that the `minter` address is non-zero.

8. Solidity compiler optimizations can be problematic

Status: **Resolved**	
Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MES-8
Target: Meson protocol	

Description
The Meson protocol has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Fix Analysis
This issue has been resolved. The Solidity optimizer in the commit evaluated in this fix review has been disabled. However, the Meson team indicated that this change causes the deployment gas amount to exceed the block gas limit. We recommend that Meson closely follow Solidity compiler releases and CHANGELOGs in order to quickly resolve any compiler optimization bugs.

9. Service fees cannot be withdrawn	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MES-9
Target: `contracts/Pools/MesonPools.sol`	

Description

If the service fee charged for a swap is waived, the fee collected for the swap is stored at index zero of the `_balanceOfPoolToken` mapping. However, because the fee withdrawal function does not allow withdrawals from index zero of the mapping, the fee can never be withdrawn. Although this limitation may be purposeful, the code appears to indicate that it is a mistake.

```
    if (!feeWaived) { // If the swap should pay service fee (charged by Meson protocol)
      uint256 serviceFee = _serviceFee(encodedSwap);
      // Subtract service fee from the release amount
      releaseAmount -= serviceFee;
      // The collected service fee will be stored in `_balanceOfPoolToken` with `poolIndex = 0`
      _balanceOfPoolToken[_poolTokenIndexForOutToken(encodedSwap, 0)] += serviceFee;
```

Figure 9.1: `contracts/Pools/MesonPools.sol:198–203`

```
  function withdraw(uint256 amount, uint48 poolTokenIndex) external {
    require(amount > 0, "Amount must be positive");

    uint40 poolIndex = _poolIndexFrom(poolTokenIndex);
    require(poolIndex != 0, "Cannot use 0 as pool index");
```

Figure 9.2: `contracts/Pools/MesonPools.sol:70–74`

Moreover, even if the function allowed the withdrawal of tokens stored at `poolIndex` 0, a withdrawal would still not be possible. This is because the owner of `poolIndex` 0 is not set during initialization, and it is not possible to register a pool with index 0.

```
  function initialize(address[] memory supportedTokens) public {
    require(!_initialized, "Contract instance has already been initialized");
    _initialized = true;
    _owner = _msgSender();
    _premiumManager = _msgSender();

    for (uint8 i = 0; i < supportedTokens.length; i++) {
      _addSupportToken(supportedTokens[i], i + 1);
    }
  }
```

Figure 9.3: `contracts/UpgradableMeson.sol:13–22`

Fix Analysis

This issue has been resolved. The source code now includes comments explaining that the service fee will not be withdrawable until the contract is updated.

10. Lack of contract existence check on transfer / transferFrom calls

Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MES-10
Target: `contracts/utils/MesonHelpers.sol`	

Description

The `MesonHelpers` contract uses the low-level `call` function to execute the `transfer` / `transferFrom` function of an ERC20 token. However, it does not first perform a contract existence check. Thus, if there is no contract at the `token` address, the low-level call will still return `success`. This means that if a supported token is subsequently self-destructed (which is unlikely to happen), it will be possible for a posted swap involving that token to succeed without actually depositing any tokens.

```
  function _unsafeDepositToken(
    address token,
    address sender,
    uint256 amount,
    bool isUCT
  ) internal {
    require(token != address(0), "Token not supported");
    require(amount > 0, "Amount must be greater than zero");
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(
      bytes4(0x23b872dd), //
bytes4(keccak256(bytes("transferFrom(address,address,uint256)")))
      sender,
      address(this),
      amount
      // isUCT ? amount : amount * 1e12 // need to switch to this line if
deploying to BNB Chain or Conflux
    ));
    require(success && (data.length == 0 || abi.decode(data, (bool))),
"transferFrom failed");
  }
```

Figure 10.1: `contracts/util/MesonHelpers.sol:53–69`

The Solidity documentation includes the following warning:

```
The low-level functions call, delegatecall and staticcall return true as their first
```

```
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 10.2: A snippet of the Solidity documentation detailing unexpected behavior related to `call`

Fix Analysis

This issue has been resolved. The low-level call is now paired with OpenZeppelin's `Address.isContract` function, which ensures that the contract at the target address is populated as expected. This makes the deposit mechanism robust against self-destructs.

11. USDT transfers to third-party contracts will fail

Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-MES-11
Target: `contracts/utils/MesonHelpers.sol (PR #65)`	

Description

To allow a user to release funds to a smart contract, the Meson protocol increases the contract's allowance (via a call to `increaseAllowance`) and then calls the contract, as shown in figure 11.1.

```
 66    IERC20Minimal(token).increaseAllowance(contractAddr, adjustedAmount);
 67    ITransferWithBeneficiary(contractAddr).transferWithBeneficiary(token,
adjustedAmount, beneficiary, data);
```

Figure 11.1: `contracts/utils/MesonHelpers.sol#66–67`

The `increaseAllowance` method, which is part of OpenZeppelin's ERC20 library, was introduced to prevent race conditions when token allowances are changed via top-level calls. However, this method is not in the ERC20 specification, and not all tokens implement it. In particular, USDT does not implement the method on the Ethereum mainnet. Thus, any attempt to release USDT to a smart contract wallet during a swap will fail, trapping the user's funds.

Fix Analysis

This issue has been resolved. The protocol now uses the standard ERC20 `approve` function to increase allowances. The team also made subtle changes to the allowance behavior: instead of incrementing an allowance when executing a transfer, the Meson contract now sets the allowance to the most recent transfer amount. If a third-party contract has an outstanding allowance from a previous swap release, it will forfeit those tokens upon the next transfer. Meson has confirmed that this is the intended behavior.

12. SDK function _randomHex returns low-quality randomness

Status: **Partially Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-MES-12
Target: `sdk/src/Swap.ts`	

Description

The Meson protocol software development kit (SDK) uses the `_randomHex` function to generate random salts for new swaps. This function accepts a string length as input and produces a random hexadecimal string of that length. To do that, `_randomHex` uses the JavaScript `Math.random` function to generate a 32-bit integer and then encodes the integer as a zero-padded hexadecimal string. The result is eight random hexadecimal characters, padded with zeros to the desired length. However, the function is called with an argument of 16, so half of the characters in the salt it produces will be zero.

```
  private _makeFullSalt(salt?: string): string {
   if (salt) {
     if (!isHexString(salt) || salt.length > 22) {
       throw new Error('The given salt is invalid')
     }
     return `${salt}${this._randomHex(22 - salt.length)}`
   }

   return `0x0000${this._randomHex(16)}`
  }

  private _randomHex(strLength: number) {
   if (strLength === 0) {
     return ''
   }
   const max = 2 ** Math.min((strLength * 4), 32)
   const rnd = BigNumber.from(Math.floor(Math.random() * max))
   return hexZeroPad(rnd.toHexString(), strLength / 2).replace('0x', '')
  }
```

Figure 12.1: `packages/sdk/src/Swap.ts#95–113`

Furthermore, the `Math.random` function is not suitable for uses in which the output of the random number generator should be unpredictable. While the protocol's current use of the function does not pose a security risk, future implementers and library users may assume that the function produces the requested amount of high-quality entropy.

Fix Analysis

This issue has been partially resolved. While the `_randomHex` function now uses cryptographic randomness to generate random hexadecimal characters, the function continues to silently output leading zeros when more than eight characters are requested or when an odd number of characters is requested. To prevent future misuse of this function, we recommend having it return a uniformly random string with the exact number of characters requested.

13. encodedSwap values are used as primary swap identifier

Status: **Unresolved**	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-MES-13
Target: `contracts/Swap/MesonSwap.sol`	

Description

The primary identifier of swaps in the `MesonSwap` contract is the `encodedSwap` structure. This structure does not contain the address of a swap's initiator, which is recorded, along with the `poolIndex` of the bonded liquidity provider (LP), as the `postingValue`. If a malicious actor or maximal extractable value (MEV) bot were able to front-run a user's transaction and post an identical `encodedSwap`, the original initiator's transaction would fail, and the initiator's swap would not be posted.

```
48    function postSwap(uint256 encodedSwap, bytes32 r, bytes32 s, uint8 v, uint200
postingValue)
49     external forInitialChain(encodedSwap)
50    {
51     require(_postedSwaps[encodedSwap] == 0, "Swap already exists");
       ...
```

Figure 13.1: `contracts/Swap/MesonSwap.sol#48–52`

Because the Meson protocol supports only 1-to-1 stablecoin swaps, transaction front-running is unlikely to be profitable. However, a bad actor could dramatically affect a specific user's ability to transact within the system.

Fix Analysis

This issue has not been resolved. Meson acknowledged that user transactions can be blocked from execution by malicious actors. However, blocking a swap transaction would require an adversary to post a corresponding swap, and to thus burn gas and have his or her funds temporarily locked; these disincentives limit the impact of this issue.

14. Unnecessary _releasing mutex increases gas costs	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-MES-14
Target: `contracts/Pools/MesonPools.sol (PR #65)`	

Description

When executing a swap in the third-party dApp integration release mode, the Meson protocol makes a call to an untrusted user-specified smart contract. To prevent reentrancy attacks, a flag is set before and cleared after the untrusted contract call.

```
require(!_releasing, "Another release is running");
 ...
_releasing = true;
_transferToContract(_tokenList[tokenIndex], recipient, initiator, amount, tokenIndex == 255, _saltDataFrom(encodedSwap));
_releasing = false;
```

Figure 14.1: `contracts/Pools/MesonPools.sol#181–221`

This flag is not strictly necessary, as by the time the contract reaches the untrusted call, it has already cleared the `_lockSwaps` entry corresponding to the release, preventing duplicate releases via reentrancy.

```
uint80 lockedSwap = _lockedSwaps[swapId];
require(lockedSwap != 0, "Swap does not exist");
...
_checkReleaseSignature(encodedSwap, recipient, r, s, v, initiator);
_lockedSwaps[swapId] = 0;
...
_release(encodedSwap, tokenIndex, initiator, recipient, releaseAmount);
```

Figure 14.2: `contracts/Pools/MesonPools.sol#191–197`

Fix Analysis

This issue has been resolved. The redundant `_releasing` flag has been removed. The call to the external contract is the last step in the transaction, which prevents reentrancy attacks.

15. Misleading result returned by view function getPostedSwap

Status: **Resolved**	
Severity: Low	**Difficulty: Low**
Type: Undefined Behavior	Finding ID: TOB-MES-15
Target: `contracts/Swap/MesonSwap.sol`	

Description

The value returned by the `getPostedSwap` function to indicate whether a swap has been executed can be misleading. Once a swap has been executed, the value of the swap is reset to either 0 or 1. However, the `getPostedSwap` function returns a result indicating that a swap has been executed only if the swap's value is 1.

```
if (_expireTsFrom(encodedSwap) < block.timestamp + MIN_BOND_TIME_PERIOD) {
  // The swap cannot be posted again and therefore safe to remove it.
  // LPs who execute in this mode can save ~5000 gas.
  _postedSwaps[encodedSwap] = 0;
} else {
  // The same swap information can be posted again, so set `_postedSwaps` value to 1 to prevent that.
  _postedSwaps[encodedSwap] = 1;
}
```

Figure 15.1: `contracts/Swap/MesonSwap.sol:140–148`

```
/// @notice Read information for a posted swap
function getPostedSwap(uint256 encodedSwap) external view
  returns (address initiator, address poolOwner, bool executed)
{
  uint200 postedSwap = _postedSwaps[encodedSwap];
  initiator = _initiatorFromPosted(postedSwap);
  executed = postedSwap == 1;
  if (initiator == address(0)) {
    poolOwner = address(0);
  } else {
    poolOwner = ownerOfPool[_poolIndexFromPosted(postedSwap)];
  }
}
```

Figure 15.2: `contracts/Swap/MesonSwap.sol:162–173`

Front-end services (or any other service interacting with this function) may be misled by the return value, reacting as though a swap has not been executed when it actually has.

Fix Analysis

This issue has been resolved. The `getPostedSwap` function's return value has been renamed to `exist`, which more accurately reflects the meaning of the value.

A. 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.

B. 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.