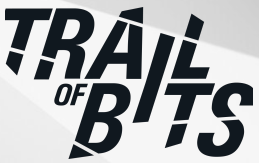

Meson Protocol

Security Assessment

October 3, 2022

Prepared for:
Phil Li and Edrick Yuhui Guan
Meson

Prepared by: **Alexander Remie, Tjaden Hess, and Damilola Edwards**

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.

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 timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are 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

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-MES-1**
  A hash collision between a “request” message and a “release” message could enable an attacker to steal other users’ funds.

- **TOB-MES-6**
  The Meson protocol contracts lack a `__gap` variable. Thus, during an upgrade, new variables cannot be added to any contract but the root contract (`UpgradableMeson`).

- **TOB-MES-10**
  The `MesonHelpers` contract executes a low-level call when transferring ERC20 tokens but does not check whether there is a contract at the target address. As a result, a swap can succeed even if its initiator has not deposited any tokens.

- **TOB-MES-11**
  The Meson protocol uses the nonstandard `increaseAllowance` method during the release phase of a swap. However, certain ERC20 tokens (e.g., USDT) do not support this method, which means that those tokens cannot be released during a swap.

Summary of Recommendations

The Meson protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Meson address the findings detailed in this report and take the following additional steps prior to deployment:

- Consider creating an incident response plan to supplement the "Security Precautions" section of the documentation. See appendix E for recommendations on creating an incident response plan.
- Consider using an upgradeability pattern that does not involve use of the `delegatecall` proxy pattern; see appendix D for related recommendations.
- Implement automated testing of forked versions of all supported ERC20 token contracts. Ensure that the tests are run against the contract versions that are actually present on the supported networks; do not assume that token contract implementations on testnets will match the mainnet versions.
- Write unit tests for the `addAuthorizedAddr` and `removeAuthorizedAddr` functions. They currently have no unit tests.
- Integrate the Echidna fuzz test provided in appendix F into the continuous integration pipeline of the Meson protocol repository. This will help prevent future updates from causing any of the bit-shifting operations to return incorrect results.
- Consider performing a separate audit of the off-chain relayer, liquidity provider (LP) client, and JavaScript software development kit (SDK) to uncover any bugs that could impact the overall security of the system.

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 engineers were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@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
October 3, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Meson protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there any flaws in the deployment mechanism?
- Are there any flaws in the contracts that could prevent upgrades?
- Is support for the EVM-based chains implemented correctly?
- Do all important actions trigger events?
- Are all function inputs validated?
- Could invalid data be included in a contract's storage?
- Are any of the functions susceptible to front-running?
- Is the system vulnerable to reentrancy attacks?
- Could there be hash collisions between the different hashing schemas?
- Are all of the bit-shifting operations implemented correctly?
- Could the use of low-level calls cause any problems?
- Are there any timing issues in the system?
- Is cross-chain bridging implemented correctly?
- Are any of the chains susceptible to congestion?
- Are there any ways to create a denial of service in the system?
- Could a user's funds become stuck in the system?
- Can LPs steal users' funds?

Project Targets

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

Meson Protocol

Repository	https://github.com/MesonFi/meson-contracts-solidity
Versions	`d89ccc23d3c28d12d7110578d08903864b75b434`, `e26107628136fe2be3675a28a3fe12cae618fa64`
Type	Solidity
Platform	Ethereum

Project Coverage

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

`UpgradableMeson`. `UpgradableMeson`, the main contract in the system, inherits from all of the other contracts and uses the OpenZeppelin Universal Upgradeable Proxy Standard pattern. We reviewed the contract's initialization and its upgradeability mechanism.

`Meson`. This contract is simply a non-upgradeable version of the `UpgradableMeson` contract.

`MesonSwap`. This contract contains the functionality for initiating a swap request on the source chain. We checked the contract for mishandled or overlooked edge cases, front-running and reentrancy risks, and ways in which an LP or user could steal another user's funds or lose access to his or her own funds.

`MesonPools`. This contract manages pools of LP tokens and finalizes swaps on the destination chain. We reviewed the contract for mishandled or overlooked edge cases, front-running and reentrancy risks, and ways in which an LP or user could steal another user's funds or lose access to his or her own funds.

`MesonHelpers`. This contract contains helper functions for transferring ERC20 tokens, functions that verify initiator signatures on swaps, and all of the functions required to encode / decode swap data into / from `uint256` values. We manually reviewed the transfer functions, checked the signature-validation functions for collision risks, and used Echidna to perform manual and dynamic testing of all of the bit-shifting operations.

`MesonStates`. This contract is inherited by the `MesonPools` and `MesonSwap` contracts and contains the storage variables that keep track of pool information. We manually reviewed the contract to identify any cases in which it could return an incorrect pool token balance.

`MesonTokens`. This contract stores all of the supported ERC20 tokens. We performed a manual review of the contract, looking for ways to register the same supported token multiple times and for any missing input validation.

`UCTUpgradeable`. The protocol uses this upgradeable ERC20 token contract to provide payouts during promotional events. We manually reviewed the contract's functionality and used `slither-check-erc` to check its conformance to the ERC20 standard.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- The Meson protocol JavaScript SDK
- Off-chain components located in other repositories, such as the Meson relayer, the LP client, and the web front end

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tool in the automated testing phase of this project:

Tool	Description	Policy
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix D

Test Results

The results of this focused testing are detailed below.

`MesonHelpers.sol`. This contract contains the functions that perform bit shifts to encode / decode values into / from `uint256` values.

Property	Tool	Result
The `_amountFrom` function always returns the encoded `amount` value.	Echidna	**Passed**
The `_saltFrom` function always returns the encoded `salt` value.	Echidna	**Passed**
The `_feeForLp` function always returns the encoded `fee` value.	Echidna	**Passed**
The `_expireTsFrom` function always returns the encoded `expireTs` value.	Echidna	**Passed**

The `_outChainFrom` function always returns the encoded `outChain` value.	Echidna	**Passed**
The `_outTokenIndexFrom` function always returns the encoded `outToken` value.	Echidna	**Passed**
The `_inChainFrom` function always returns the encoded `inChain` value.	Echidna	**Passed**
The `_inTokenIndexFrom` function always returns the encoded `inToken` value.	Echidna	**Passed**
The `_untilFromLocked` function always returns the encoded `until` value.	Echidna	**Passed**
The `_poolIndexFromLocked` function always returns the encoded `poolIndex` value.	Echidna	**Passed**
The `_tokenIndexFrom` function always returns the encoded `tokenIndex` value.	Echidna	**Passed**
The `_poolIndexFrom` function always returns the encoded `poolIndex` value.	Echidna	**Passed**
The `_initiatorFromPosted` function always returns the encoded `initiator` value.	Echidna	**Passed**
The `_poolIndexfromPosted` function always returns the encoded `poolIndex` value.	Echidna	**Passed**
The `_poolTokenIndexForOutToken` function always returns the encoded `outToken` value.	Echidna	**Passed**

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 system uses Solidity v0.8.0 arithmetic, which automatically prevents integer overflows, and contains little arithmetic beyond a handful of addition and subtraction operations.	**Satisfactory**
Auditing	We identified multiple critical operations that do not trigger events (TOB-MES-4). The system also lacks a blockchain-monitoring mechanism for tracking important system events.	**Moderate**
Authentication / Access Controls	We did not identify any access control issues, and the Meson protocol documentation clearly describes the different actors in the system.	**Satisfactory**
Complexity Management	Each of the system's functions performs a single task and is well documented in NatSpec and inline comments.	**Satisfactory**
Cryptography and Key Management	We identified a couple of cases in which message signatures could collide (TOB-MES-1, TOB-MES-2) and found that the values used to generate salts are insufficiently random (TOB-MES-12).	**Moderate**
Decentralization	The smart contracts are decentralized by design; the sole privileged role is the premium manager, which lacks the ability to steal or lock user funds. Moreover, the Meson team indicated that the `UpgradableMeson` contract is controlled by a multisignature wallet. However, since the	**Satisfactory**

	`UpgradableMeson` contract can be upgraded by the Meson team (or by an attacker with access to the proxy admin account), the implementation of the other contracts could completely change at any time. Lastly, the off-chain relayer, used to relay swap requests, is currently centralized and controlled by Meson; the team should consider decentralizing the component in the future.	
Documentation	The official Meson protocol documentation is extensive and correctly describes the implementation. The implementation contains NatSpec and inline comments.	**Satisfactory**
Front-Running Resistance	The `MesonSwap` contract is vulnerable to front-running, which could prevent users from successfully submitting swaps (TOB-MES-13). However, because the service supports only 1-to-1 stablecoin swaps, front-running may not be profitable.	**Moderate**
Low-Level Manipulation	Three code paths use assembly, which is error-prone. Additionally, when transferring ERC20 tokens, the `MesonHelpers` contract executes low-level calls without performing contract existence checks (TOB-MES-10).	**Moderate**
Testing and Verification	The tests provided by the Meson team are appropriate, achieve sufficient coverage, and appear to cover various edge cases. However, the tests are run in a local environment, with a mock ERC20 token contract standing in for the supported stablecoin contracts. As a result, the test environment behavior may differ significantly from the on-chain behavior (TOB-MES-11). Additionally, there is no unit testing of the `addAuthorizedAddr` or `removeAuthorizedAddr` function. Lastly, the Meson team should consider writing additional Echidna properties for the swapping logic.	**Moderate**

Summary of Findings

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

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

13	encodedSwap values are used as primary swap identifier	Denial of Service	**Medium**
14	Unnecessary _releasing mutex increases gas costs	Denial of Service	**Informational**
15	Misleading result returned by view function getPostedSwap	Undefined Behavior	**Low**

Detailed Findings

1. Hash collisions in untyped signatures

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.

```
 213    function _checkRequestSignature(
...
 237     if (nonTyped) {
 238       bytes32 digest = keccak256(abi.encodePacked(
 239         bytes28(0x19457468657265756d205369676e6564204d6573736167653a0a3332), //
HEX of "\x19Ethereum Signed Message:\n32"
 240         encodedSwap
 241       ));
 242       require(signer == ecrecover(digest, v, r, s), "Invalid signature");
 243       return;
 244     }
...
 266    function _checkReleaseSignature(
...
 293     if (nonTyped) {
 294       digest = keccak256(abi.encodePacked(
 295         bytes28(0x19457468657265756d205369676e6564204d6573736167653a0a3332), //
HEX of "\x19Ethereum Signed Message:\n32"
 296         keccak256(abi.encodePacked(encodedSwap, recipient))
 297       ));
...
```

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.

Exploit Scenario

Alice computes `swap1`, `swap2`, and `recipient` values such that `swap1 = keccak256(swap2, recipient)` and `swap1` and `swap2` refer to valid swaps from chain A to chain B.

Alice then convinces Bob to post `swap1` and `swap2` on chain A, locks `swap2` on chain B, and uses the request signature on `swap1` as the release signature on `swap2`. This enables her to release the funds to her wallet (the `recipient` address) and to collect the funds on both sides of the swap.

Recommendations

Short term, prefix untyped messages with the message type (request or release), and include all message fields in the top-level digest, as shown in figure 1.2:

```
bytes32 digest = keccak256(abi.encodePacked(
        bytes28("\x19Ethereum Signed Message:\n71"),
        bytes7("REQUEST"),
        encodedSwap,
        recipient
      ));
```

Figure 1.2: The recommended modifications to the top-level digest

Long term, document the formats of all messages that could possibly be validated by the `MesonHelpers` contract, and ensure that the message type and all parameters can be unambiguously extracted from the top-level digest for verification. This will help ensure that the message encoding is injective. Additionally, consider implementing legacy signatures by using EIP-712-compliant messages as the input to `eth_sign`.

2. Typed signatures implement insecure nonstandard encodings	
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:

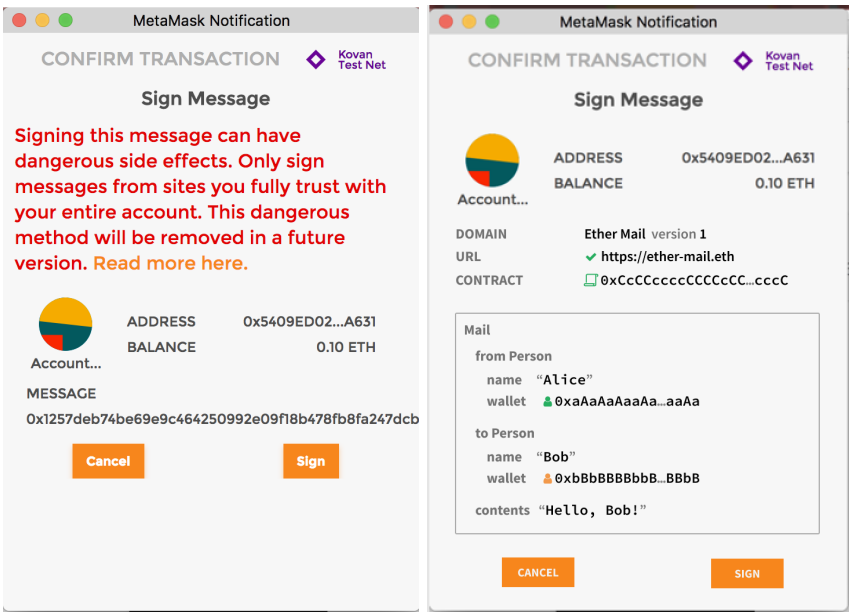

Figure 2.1: A reproduction of images from the EIP-712 standard

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)")`.

```
246    bytes32 typehash = REQUEST_TYPE_HASH;
247    bytes32 digest;
248    assembly {
249     mstore(0, encodedSwap)
250     mstore(32, keccak256(0, 32))
251     mstore(0, typehash)
252     digest := keccak256(0, 64)
253    }
```

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.

Exploit Scenario

The Meson team adds a new message type, the `typehash` of which collides with EIP-191-defined message types (e.g., begins with "`\x19\x45`") or with RLP-encoded data (e.g., Ethereum transactions). As a result, signatures can be replayed in unintended contexts.

Recommendations

Short term, include an EIP-191 prefix in the message encoding to distinguish the encoding from that used with other message types. For example, beginning all messages with "`\x19\xff`" would differentiate messages that do not adhere to EIP-712 from other EIP-191 messages.

Long term, ensure that all messages are formatted, hashed, and signed in compliance with EIP-712.

3. Missing validation in the _addSupportToken function

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.

```
47    function _addSupportToken(address token, uint8 index) internal {
48     require(index != 0, "Cannot use 0 as token index");
49     _indexOfToken[token] = index;
50     _tokenList[index] = token;
51    }
```

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

Recommendations

Short term, have `_addSupportToken` check that the token is not already registered in the mapping (i.e., that its index is greater than zero).

Long term, implement validation of function inputs whenever possible.

4. Insufficient event generation	
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.

Exploit Scenario
An attacker discovers a vulnerability in the `MesonPools` contract and is able to modify its execution. Because the attacker's actions do not trigger any events, the behavior goes unnoticed until it has caused damage such as financial losses.

Recommendations
Short term, add events for all operations to strengthen the monitoring and alert systems of the protocol. Events aid in contract monitoring and the detection of suspicious behavior.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

5. Use of an uninitialized state variable in functions

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`

Recommendations

Short term, set the `_mesonContract` address in the `initialize` function.

Long term, carefully review all state variables in the contracts and ensure that they are explicitly set upon the creation of the contracts or during their construction / initialization.

6. Risk of upgrade issues due to missing `__gap` variable

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.

```
 89     /**
 90     * @dev This empty reserved space is put in place to allow future versions to
add new
 91     * variables without shifting down storage in the inheritance chain.
 92     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
 93     */
 94     uint256[49] private __gap;
```

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

Exploit Scenario

Alice, a developer of the Meson protocol, adds a new variable to the `MesonStates` contract as part of an upgrade. As a result of the addition, the storage slot of each subsequent variable changes, and the contract stops working.

Recommendations

Short term, add a `__gap` variable (specifically `uint256 __gap[100]`) to all stateful Meson protocol contracts from which `UpgradableMeson` inherits.

Long term, consider redesigning the system such that the contracts can be upgraded through a mechanism other than the proxy pattern.

7. Lack of a zero-value check on the initialize function

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.

```
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 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.

Exploit Scenario

When deploying a new version of the `UCTUpgradeable` contract, Alice mistakenly passes in the zero address for the `minter` argument. The misconfiguration causes the contract to exhibit unexpected behavior.

Recommendations

Short term, add a zero-value check for the `minter` argument to ensure that users cannot accidentally set an incorrect value, misconfiguring the system.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's continuous integration pipeline, pre-commit hooks, or build scripts.

8. Solidity compiler optimizations can be problematic	
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.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Meson protocol contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

9. Service fees cannot be withdrawn

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.

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

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

```
70      function withdraw(uint256 amount, uint48 poolTokenIndex) external {
71          require(amount > 0, "Amount must be positive");
72
73          uint40 poolIndex = _poolIndexFrom(poolTokenIndex);
74          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.

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

```
18
19     for (uint8 i = 0; i < supportedTokens.length; i++) {
20       _addSupportToken(supportedTokens[i], i + 1);
21     }
22    }
```

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

Recommendations

Short term, either allow fees stored at `poolIndex` 0 to be withdrawn, or add a comment to the implementation explaining the reason that they cannot be withdrawn.

Long term, consider burning any fees that will not ever be withdrawn.

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

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.

```
 53    function _unsafeDepositToken(
 54     address token,
 55     address sender,
 56     uint256 amount,
 57     bool isUCT
 58    ) internal {
 59     require(token != address(0), "Token not supported");
 60     require(amount > 0, "Amount must be greater than zero");
 61     (bool success, bytes memory data) = token.call(abi.encodeWithSelector(
 62       bytes4(0x23b872dd), //
bytes4(keccak256(bytes("transferFrom(address,address,uint256)")))
 63       sender,
 64       address(this),
 65       amount
 66       // isUCT ? amount : amount * 1e12 // need to switch to this line if
deploying to BNB Chain or Conflux
 67     ));
 68     require(success && (data.length == 0 || abi.decode(data, (bool))),
"transferFrom failed");
 69    }
```

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`

Exploit Scenario

A token that is supported by the Meson protocol, token X, is self-destructed. However, the Meson protocol does not immediately detect the change and still allows users to post swaps with token X. Because the `MesonHelpers` contract lacks a contract existence check, these swaps succeed.

Recommendations

Short term, implement a contract existence check before each low-level call, and / or check the balance of the sender before and after each call to verify that the expected `amount` has actually been transferred.

Long term, carefully review the Solidity documentation, especially the "Warnings" section.

11. USDT transfers to third-party contracts will fail

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.

Exploit Scenario

Bob, a Meson protocol user, creates a swap from USDC to USDT on the Ethereum mainnet, using the third-party decentralized application (dApp) wallet feature. Bob then attempts to release the funds. However, because USDT does not implement the `increaseAllowance` function, the call fails, and Bob's funds are trapped.

Recommendations

Short term, use standard ERC20 methods to perform allowance increases, as shown in figure 11.2.

```
    uint256 adjustedAmount = amount + IERC20Minimal(token).allowance(contractAddr);
    IERC20Minimal(token).approve(contractAddr, adjustedAmount);
    ITransferWithBeneficiary(contractAddr).transferWithBeneficiary(token,
adjustedAmount, beneficiary, data);
```

Figure 11.2: An example implementation of an allowance increase

An implementation such as this one is safe because the retrieval and incrementation of an allowance happen atomically, preventing race conditions.

Long term, implement tests for all features, especially those used to deposit and withdraw funds, create swaps, and release funds during swaps. Additionally, ensure that these tests use exact copies of the on-chain token contracts and cover all combinations of supported chains and tokens. For example, test against a fork of the Ethereum mainnet rather than against a fresh network with a synthetic mock token.

12. SDK function _randomHex returns low-quality randomness

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.

```
 95    private _makeFullSalt(salt?: string): string {
 96     if (salt) {
 97       if (!isHexString(salt) || salt.length > 22) {
 98         throw new Error('The given salt is invalid')
 99       }
100       return `${salt}${this._randomHex(22 - salt.length)}`
101     }
102
103     return `0x0000${this._randomHex(16)}`
104    }
105
106    private _randomHex(strLength: number) {
107     if (strLength === 0) {
108       return ''
109     }
110     const max = 2 ** Math.min((strLength * 4), 32)
111     const rnd = BigNumber.from(Math.floor(Math.random() * max))
112     return hexZeroPad(rnd.toHexString(), strLength / 2).replace('0x', '')
113    }
```

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.

Exploit Scenario

The SDK is updated such that it uses the `_randomHex` function to generate cryptographic secrets. An attacker can then easily brute-force the keys generated by that function.

Recommendations

Short term, add a comment documenting the fact that `_randomHex` is not suitable for use in security-critical applications.

Long term, modify `_randomHex` to use a cryptographically secure pseudorandom number generator that outputs random data of the desired length (e.g., `ethers.utils.randomBytes`).

13. encodedSwap values are used as the primary swap identifier

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.

Exploit Scenario

A malicious actor wishes to censor a particular user. To do that, he submits a transaction with the `encodedSwap` value of the user's swap (having seen it in the mempool) ahead of the user's transaction. This prevents the user from posting the swap.

Recommendations

Short term, monitor the live `MesonSwap` contract for any transaction failures caused by `encodedSwap` duplication.

Long term, consider using `swapId` as the primary identifier of swaps in the `MesonSwap` contract, as is done in `MesonPools`.

14. Unnecessary _releasing mutex increases gas costs

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.

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

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

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

Exploit Scenario

Users taking advantage of third-party DApp integrations pay higher gas costs when executing swaps because of the unnecessary write operation.

Recommendations

Short term, remove the `_releasing` flag and guard logic.

Long term, ensure that calls to untrusted contracts occur only after all state modifications are complete. Consider moving the `_release` call so that it occurs after the `SwapReleased` event has been emitted; this will ensure consistency in the order of events.

15. Misleading result returned by view function getPostedSwap

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.

```
141     if (_expireTsFrom(encodedSwap) < block.timestamp + MIN_BOND_TIME_PERIOD) {
142       // The swap cannot be posted again and therefore safe to remove it.
143       // LPs who execute in this mode can save ~5000 gas.
144       _postedSwaps[encodedSwap] = 0;
145     } else {
146       // The same swap information can be posted again, so set `_postedSwaps`
value to 1 to prevent that.
147       _postedSwaps[encodedSwap] = 1;
148     }
```

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

```
161     /// @notice Read information for a posted swap
162     function getPostedSwap(uint256 encodedSwap) external view
163       returns (address initiator, address poolOwner, bool executed)
164     {
165       uint200 postedSwap = _postedSwaps[encodedSwap];
166       initiator = _initiatorFromPosted(postedSwap);
167       executed = postedSwap == 1;
168       if (initiator == address(0)) {
169         poolOwner = address(0);
170       } else {
171         poolOwner = ownerOfPool[_poolIndexFromPosted(postedSwap)];
172       }
173     }
```

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.

Recommendations

Short term, redesign the way that the `getPostedSwap` function identifies executed swaps so that it returns a correct result in all cases, rename its return value, or document the edge case.

Long term, when designing view functions that will be used by other applications, ensure that the names of their return values accurately reflect what the values are meant to convey, and document any edge cases or discrepancies in a NatSpec comment. This will help prevent those applications from misinterpreting a contract state.

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
Front-Running Resistance	The system's resistance to front-running attacks
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

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 Findings

This appendix lists findings that are not associated with specific vulnerabilities.

- **Hard-coded hexadecimal values.** Several constants in the Meson codebase are represented as hard-coded hexadecimal values. Using these values rather than ASCII-encoded values and compile time–computed hashes reduces readability and increases the chance of typos.

```
   ? bytes25(0x1954524f4e205369676e6564204d6573736167653a0a33330a)  // HEX of
"\x19TRON Signed Message:\n33\n"
   : bytes25(0x1954524f4e205369676e6564204d6573736167653a0a33320a), // HEX of
"\x19TRON Signed Message:\n32\n"
```

Figure C.1: contracts/utils/MesonHelpers.sol#229–230

```
    ? bytes25("\x19TRON Signed Message:\n33\n")
    : bytes25("\x19TRON Signed Message:\n32\n"),
```

Figure C.2: The code in figure C.1, with ASCII literals instead of hard-coded hexadecimals

```
mstore(0, 0xf6ea10de668a877958d46ed7d53eaf47124fda9bee9423390a28c203556a2e55) //
mainnet
```

Figure C.3: contracts/utils/MesonHelpers.sol#304

```
bytes32 constant MAINNET_TYPEHASH = keccak256(abi.encodePacked("bytes32 Sign to
release a swap on Meson", "bytes21 Recipient (tron address in hex format)"));
...
mstore(0, MAINNET_TYPEHASH)
```

Figure C.4: The code in figure C.3, with a compile time–computed hash instead of a constant

- **Unused constant value.** The codebase contains an unused constant representing the ABI selector for the ERC20 `transfer` function. A duplicate hexadecimal constant is used instead.

```
bytes4 private constant ERC20_TRANSFER_SELECTOR =
bytes4(keccak256(bytes("transfer(address,uint256)")));
```

Figure C.5: contracts/utils/MesonHelpers.sol#10

```
bytes4(0xa9059cbb), // bytes4(keccak256(bytes("transfer(address,uint256)")))
```

Figure C.6: contracts/utils/MesonHelpers.sol#37

- **Comments that confuse the LP fee with the service fee.** The two comments shown in figures C.7 and C.8 refer to the "service fee," while the code actually deals

with the LP fee. The service fee is a separate fee that is deducted from the amount of a swap elsewhere.

```
// Only (amount - service fee) is locked from the LP pool. The service fee will be
charged on release
_balanceOfPoolToken[poolTokenIndex] -= (_amountFrom(encodedSwap) -
_feeForLp(encodedSwap));
```

Figure C.7: contracts/Pools/MesonPools.sol#134–135

```
// (amount - service fee) will be returned because only that amount was locked
_balanceOfPoolToken[poolTokenIndex] += (_amountFrom(encodedSwap) -
_feeForLp(encodedSwap));
```

Figure C.8: contracts/Pools/MesonPools.sol#154–155

- **Use of hard-coded values.** In the code in figure C.9, the value `1e11` represents the maximum swap amount, which is 100,000, with 8 decimals. In the code in figure C.10, the value `1000` is used to represent 0.1%. Using a named constant in both of these instances would increase the code's readability and be considered best practice.

```
require(amount <= 1e11, "For security reason, amount cannot be greater than 100k");
```

Figure C.9: contracts/Swap/MesonSwap.sol#54

```
function _serviceFee(uint256 encodedSwap) internal pure returns (uint256) {
  return _amountFrom(encodedSwap) / 1000; // Default to `serviceFee` = 0.1% *
`amount`
}
```

Figure C.10: contracts/utils/MesonHelpers.sol#85–87

D. Upgradeability Recommendations

This appendix provides recommendations on upgrading the contracts through migrations and designing the contracts to be upgraded without using the potentially risky `delegatecall` proxy pattern.

Upgradeability with Migration

In essence, the contract migration process consists of two steps: recovering data from the old contract and writing the data to the new contract.

Consider the following questions before implementing this approach:

- **Which contracts contain a state that will need to be migrated when a new version of the contract must be deployed?** When a new version of the Meson contract is deployed, the mappings in `MesonStates` will need to be migrated to preserve pool balance and ownership information. In addition, the Meson contract's ERC20 token balances will need to be preserved or migrated with each update.
- **Which features / contracts will need to be paused during the migration of a contract?** When a contract is being migrated, the system will need to be paused so that data can be cleanly extracted and then written back into a new storage contract.
- **How will the different types of data be extracted from the contracts?** Which types of data can be retrieved through public getters, which need to be recovered through events, which need to be recovered from dynamic arrays or mappings, and which need to be recovered from private variables? Each of these warrants a different way of extracting the data.
- **How will the team ensure that all data is read out of each storage contract?** Document the steps involved in recovering all of the data from each contract, and write tests to ensure that all data is successfully extracted.
- **How will each type of data be written into the new contract?** Create a process for writing data into the new contract and write related tests.
- **How much will a migration cost?** Because each transaction carries a gas cost and there is a block gas limit, the migration of a contract may be a costly procedure that spans multiple transactions / blocks. Calculating the cost of migrating the contracts (with their varying amounts of storage data) up front will enable the team to choose the most efficient plan when the need to migrate a contract arises. These calculations will also provide an estimate of the number of transactions / blocks that a migration will require.

- **In what state will the new contract be deployed?** Consider setting the contract's initial state to paused so that data can be written into it; the contract can then be unpaused.

- **Which references will need to be updated when the contracts are migrated to a new version?** When checking the functioning of the system after data is written into a new contract, also check whether all required references have been updated.

- **How will the migration of each contract affect external contracts that interact with the Meson protocol?** Communicating with external systems that interact with the Meson contracts could improve the user experience.

References

- How contract migration works

Upgradeability with Data Separation

It is possible to design a contract such that it can be upgraded without using the `delegatecall` proxy pattern. The core idea is to develop separate logic and storage contracts. Additionally, if an entry point contract is implemented, an upgrade to the logic or storage contract will not change the address of the contract; thus, external contracts that interact with the system will not need to change the address that they point to.

Consider the following questions before implementing this approach:

- **How will the access controls be designed?** Who will be allowed to upgrade the contracts? Will the pause functionality of the system be implemented in the entry point contract or the logic contract, and who will be able to pause / unpause the contract?

- **Which transaction-related variables will need to be adjusted to implement this approach?** For example, in a contract that does not use the `delegatecall` proxy pattern, the original `msg.sender` is not propagated; if the target function requires the value, `msg.sender` will need to be passed to that function explicitly in a function argument. Similarly, the ether sent in the call will need to be passed explicitly to the next contract's function call, as it will not be automatically propagated like it would be through a `delegatecall`.

- **Will there be one storage contract for the entire system, or one storage contract per contract?** Using one contract would simplify the system's storage. However, migrating the data of a single contract would require migrating *all* of the contracts' data from the old storage contract to the new storage contract.

- **How will the team separate the logic to minimize the amount of logic that needs to be updated?** If the business logic is separated from the arithmetic,

upgrading either aspect of the system will require updating only the relevant contract. Functionality that is shared by multiple contracts can be put in a single contract referenced by those other contracts.

- **How will contract upgrades be tested?** Writing tests for upgrades of the logic and / or the storage of each contract will help the team uncover unforeseen problems.
- **How will data be stored in the storage contract(s)?** The method of storage could be low level (e.g., `storeUint256()`) or more high level, like, for example, `storeVestingSchedule()`.
- **Is the migration process designed to efficiently read out all data from the storage contract(s) and write it back?** Designing the storage contract layout in a way that facilitates the process of reading out and writing back all of the data could make the migration process much easier and cheaper.

References

- Designing the Gemini dollar, a regulated, upgradeable, transparent stablecoin

E. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Meson will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

F. Echidna Integration

During the audit, we integrated Echidna into the codebase to implement various invariant checks. This practice allowed us to identify system properties and implement a fuzz test that automatically generates random inputs to call smart contract functions.

The code in figure F.1 can be added to the `contracts/` folder in a file named `Echidna.sol`. After `yarn build` has been used to compile the fuzz test, it can be executed via the following command:

```
$ echidna-test contracts/Echidna.sol –test-mode assertion
```

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.6;

import "./utils/MesonHelpers.sol";

contract Echidna is MesonHelpers {

  function encodeSwap(
    uint48 amount,
    uint80 salt,
    uint40 fee,
    uint40 expireTs,
    bytes2 outChain,
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) public pure returns (bytes memory) {
    return
      abi.encodePacked(
        amount,
        salt,
        fee,
        expireTs,
        outChain,
        outToken,
        inChain,
        inToken
      );
  }

  function postedSwap(address initiator, uint40 poolIndex) public pure returns(bytes
memory) {
    return
    abi.encodePacked(
      initiator,
      poolIndex
    );
  }
```

```
function testAmountFrom(
  uint48 amount,
  uint80 salt,
  uint40 fee,
  uint40 expireTs,
  bytes2 outChain,
  uint8 outToken,
  bytes2 inChain,
  uint8 inToken
) external {
  bytes memory encoded = encodeSwap(
    amount,
    salt,
    fee,
    expireTs,
    outChain,
    outToken,
    inChain,
    inToken
  );
  assert(amount == _amountFrom(uint256(bytes32(encoded))));
}

function testSaltFrom(
  uint48 amount,
  uint80 salt,
  uint40 fee,
  uint40 expireTs,
  bytes2 outChain,
  uint8 outToken,
  bytes2 inChain,
  uint8 inToken
) external {
  bytes memory encoded = encodeSwap(
    amount,
    salt,
    fee,
    expireTs,
    outChain,
    outToken,
    inChain,
    inToken
  );
  assert(salt == _saltFrom(uint256(bytes32(encoded))));
}

function testFee(
  uint48 amount,
  uint80 salt,
  uint40 fee,
  uint40 expireTs,
  bytes2 outChain,
```

```
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    assert(fee == _feeForLp(uint256(bytes32(encoded))));
  }

  function testExpireTsFrom(
    uint48 amount,
    uint80 salt,
    uint40 fee,
    uint40 expireTs,
    bytes2 outChain,
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    assert(expireTs == _expireTsFrom(uint256(bytes32(encoded))));
  }

  function testOutChainFrom(
    uint48 amount,
    uint80 salt,
    uint40 fee,
    uint40 expireTs,
    bytes2 outChain,
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
```

```
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    assert(outChain == bytes2(_outChainFrom(uint256(bytes32(encoded)))));
  }

  function testOutTokenIndexFrom(
    uint48 amount,
    uint80 salt,
    uint40 fee,
    uint40 expireTs,
    bytes2 outChain,
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    assert(outToken == _outTokenIndexFrom(uint256(bytes32(encoded))));
  }

  function testInChainFrom(
    uint48 amount,
    uint80 salt,
    uint40 fee,
    uint40 expireTs,
    bytes2 outChain,
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
```

```
    assert(inChain == bytes2(_inChainFrom(uint256(bytes32(encoded)))));
  }

  function testInTokenIndexFrom(
    uint48 amount,
    uint80 salt,
    uint40 fee,
    uint40 expireTs,
    bytes2 outChain,
    uint8 outToken,
    bytes2 inChain,
    uint8 inToken
  ) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    assert(inToken == _inTokenIndexFrom(uint256(bytes32(encoded))));
  }

  function testDecodeUntilFromLockedSwaps(uint80 until, uint40 poolIndex) external {
    require(until != 0);
    require(poolIndex != 0);
    uint80 lockedSwap = _lockedSwapFrom(until, poolIndex);
    assert(uint40(until) == uint40(_untilFromLocked(lockedSwap)));
  }

  function testDecodePoolIndexFromLockedSwaps(uint256 until, uint40 poolIndex)
external {
    require(until != 0);
    require(poolIndex != 0);
    uint80 lockedSwap = _lockedSwapFrom(until, poolIndex);
    assert(poolIndex == _poolIndexFromLocked(lockedSwap));
  }

  function testDecodeTokenIndex(uint8 tokenIndex, uint40 poolIndex) external {
    require(tokenIndex != 0);
    require(poolIndex != 0);
    bytes6 poolTokenIndex = bytes6(_poolTokenIndexFrom(tokenIndex, poolIndex));
    assert(tokenIndex == _tokenIndexFrom(uint48(poolTokenIndex)));
  }

  function testDecodePoolIndex(uint8 tokenIndex, uint40 poolIndex) external {
    require(tokenIndex != 0);
    require(poolIndex != 0);
    bytes6 poolTokenIndex = bytes6(_poolTokenIndexFrom(tokenIndex, poolIndex));
    assert(poolIndex == _poolIndexFrom(uint48(poolTokenIndex)));
```

```
  }

  function testPostedSwapInitiator(address initiator, uint40 tokenIndex) external {
    require(initiator != address(0));
    require(tokenIndex != 0);
    uint256 _postedSwap = uint256(bytes32(postedSwap(initiator, tokenIndex))) >> 56;
    assert(_initiatorFromPosted(uint200(_postedSwap)) == initiator);
  }

  function testPostedSwapPoolIndex(address initiator, uint40 poolIndex) external {
    require(initiator != address(0));
    require(poolIndex != 0);
    uint256 _postedSwap = uint256(bytes32(postedSwap(initiator, poolIndex))) >> 56;
    assert(_poolIndexFromPosted(uint200(_postedSwap)) == poolIndex);
  }

function testPoolTokenIndexForOutToken_ExtractOutToken(uint48 amount,
  uint80 salt,
  uint40 fee,
  uint40 expireTs,
  bytes2 outChain,
  uint8 outToken,
  bytes2 inChain,
  uint8 inToken,
  uint40 poolIndex
) external {
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    require(poolIndex > 0);

    uint256 _encodedSwap = uint256(bytes32(encoded));
    uint48 poolTokenIndex = _poolTokenIndexForOutToken(_encodedSwap, poolIndex);
    assert(uint8(poolTokenIndex >> 40) == outToken);
  }

  function testPoolTokenIndexForOutToken_ExtractPoolIndex(uint48 amount,
  uint80 salt,
  uint40 fee,
  uint40 expireTs,
  bytes2 outChain,
  uint8 outToken,
  bytes2 inChain,
  uint8 inToken,
  uint40 poolIndex
) external {
```

```
    bytes memory encoded = encodeSwap(
      amount,
      salt,
      fee,
      expireTs,
      outChain,
      outToken,
      inChain,
      inToken
    );
    require(poolIndex > 0);

    uint256 _encodedSwap = uint256(bytes32(encoded));
    uint48 poolTokenIndex = _poolTokenIndexForOutToken(_encodedSwap, poolIndex);
    assert(uint40(poolTokenIndex) == poolIndex);
  }


}
```

Figure F.1: An Echidna test for the `MesonHelpers.sol` contract

G. Security Best Practices for the Use of a Multisignature Wallet

Consensus requirements for sensitive actions such as spending the funds in a wallet are meant to mitigate the risk of

- any one person's judgment overruling the others',
- any one person's mistake causing a failure, and
- the compromise of any one person's credentials causing a failure.

In a 2-of-3 multisignature Ethereum wallet, for example, the execution of a "spend" transaction requires the consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, it must fulfill the following requirements:

1. The private keys must be stored or held separately, and access to each one must be limited to a different individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) ought to refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- A whitelist of specific Ethereum addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction, or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a "duress code" (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willfully, without alerting the attacker.