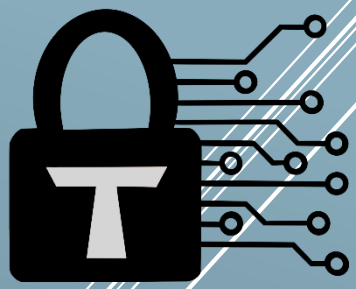
# Trust Security



Smart Contract Audit

Tea Protocol

26/02/2025

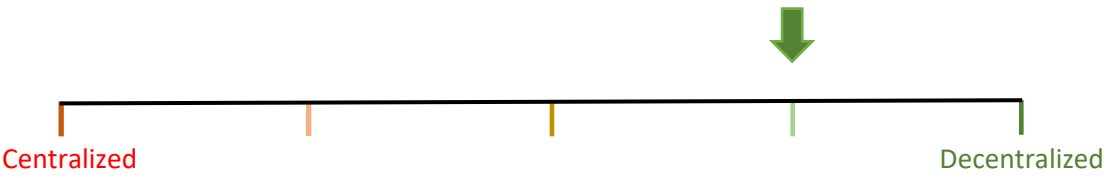
# Executive summary


| Category | Rollups |
|---|---|
| Auditor | Trust<br>MiloTruck |
| Time period | 27/01/2025 - 31/01/2025 |

Findings

| Severity | Total | Fixed | Acknowledged |
|---|---|---|---|
| High | 1 | 1 | 0 |
| Medium | 8 | 5 | 3 |
| Low | 7 | 6 | 1 |

Centralization score




Signature

Digitally signed by Trust
Date: 2025.02.26 22:41:05 +02'00'

# Document properties

## Versioning

| Version | Date | Description |
|---|---|---|
| 0.1 | 05/02/2025 | Client report |
| 0.2 | 20/02/2025 | Fix Review |
| 0.3 | 26/02/2025 | Fix Review #2 |

## Contact

**Trust**

trust@trust-security.xyz

# Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

## Scope

All changes in **tea-geth** and **optimism** against Optimism's Holocene release:

- Diff of tea-geth
- Diff of optimism

In **gpg-wallet**:

- src/GPGWalletDeployer.sol
- src/GPGWalletImpl.sol
- src/Airdropper.sol
- script/Deploy.s.sol
- script/Airdrop.s.sol

## Repository details

- **Repository URLs:**
  - https://github.com/teaxyz/tea-geth
  - https://github.com/teaxyz/optimism
  - https://github.com/teaxyz/gpg-wallet
- **Commit hashes:**
  - tea-geth: 01cce205c75756e93ec368fadc6facc1705993ae
  - optimism: fc1850596d96dd95d9b12b82f249ae2d6ab86ce8
  - gpg-wallet: 3e4d3a9558c20d3d00fe6816389d99b31c8a5b1a
- **Fix Review commit hashes:**
  - tea-geth: ec2722f02e0029c1510d70e7ee60977772ca042a
  - optimism: 4bb59058e6a144062be38fe13b538aa712c9187a
  - gpg-wallet: 49fa4be4102848797040055163712 5a80b27a161
- **Fix Review #2 commit hashes:**
  - tea-geth: 7c67edd4f04949692dda07a8aa015b0a5cec1ad4
  - optimism: a1784a0bd73feb807ddb664444172b211cbc9607
  - gpg-wallet: 16201b75e50ec71939c6bc54892fee549ace3f45

## About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

## About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge

MiloTruck is a blockchain security researcher who specializes in smart contract security. Since March 2022, he has competed in over 25 auditing contests on Code4rena and won several of them against the best auditors in the field. He has also found multiple critical bugs in live protocols on Immunefi and is an active judge on Code4rena.

## Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

## Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

# Qualitative analysis

| Metric | Rating | Comments |
|---|---|---|
| Code complexity | **Excellent** | The code is modularized well to reduce complexity. |
| Documentation | **Moderate** | Project currently has minimal documentation. |
| Best practices | **Excellent** | Project consistently adheres to industry standards. |
| Centralization risks | **Good** | There are minimal permissioned roles in the protocol, which are held by multisigs. |

# Findings

## High severity findings

### TRST-H-1: *teaPerETH()* price is susceptible to slippage

- **Category:** Integration flaws
- **Source:** TeaWAPOracle.sol
- **Status:** Fixed

**Description**

In **TeaWAPOracle**, **MIN_WETH_BALANCE** is declared as 1e18, which is the minimum amount of WETH reserves the Velodrome pool must hold for it to be used to determine the price of TEA.

In *teaPerETH()*, the price of 1e18 WETH is queried from the pool using *quote()*. *teaPerETH()* is meant to return the instantaneous price of TEA (i.e. how many units of TEA are worth 1 unit of ETH).

However, *quote()* calculates the amount of TEA using the xy=k formula on its reserves. This means the amount returned by *teaPerETH()* is affected by the liquidity of the pool and susceptible to slippage. More specifically, when the pool has the minimum of 1e18 WETH reserves, the amount of TEA returned will be far smaller than the actual price of TEA.

For example:

- Assume the pool has (1.1 TEA, 1.1 WETH) of reserves.
- Logically, 1 TEA is worth 1 WETH.
- However, *quote()* will return 0.52 TEA for 1 WETH based on the xy=k formula.

As a result, the price of TEA returned by *teaPerETH()* and stored in **GasPriceOracle** will be far smaller than its actual price, causing the L1 DA fee to be calculated incorrectly.

**Recommendation**

Consider reducing the amount of WETH passed to *quote()* to 1e9.

**Team Response**

Fixed.

**Mitigation Review**

During the refactor, an overflow issue was introduced when expanding the 1e9 quote to 1e18.

```
    oldPrice = price;
    unchecked { price = price * 1e9; }
    if (price < oldPrice) return (false, fallbackPrice);
```

It is actually possible for the new **price** to be larger than **oldPrice**, passing the check, although there was an overflow. Multiplication behaves differently from addition in that respect, and needs to be checked with **“product / x == y”-**style validation.

**Team response**

Fixed.

**Mitigation Review**

Verified, the overflow check has been modified as recommended.

## Medium severity findings

### TRST-M-1: *teaPerETH()* price is inaccurate when Veldorome's pool is paused

- **Category:** Integration flaws
- **Source:** TeaWAPOracle.sol
- **Status:** Fixed

**Description**

In *teaPerETH()*, the price of TEA is *queried from a Velodrome pool using quote()*.

However, *teaPerETH()* does not check if the pool is paused. When the pool is paused, its reserves will be outdated since it's not possible for users to swap through the pool. Therefore, the price returned from *quote()* will not reflect the actual value of TEA.

As a result, **GasPriceOracle** will store an incorrect price whenever the pool is paused, causing the L1 DA fee to be calculated incorrectly.

**Recommendation**

In *teaPerETH()*, check if the pool is paused and use the fallback price if so.

**Team Response**

Fixed.

**Mitigation Review**

While code has been introduced to validate the pool is not paused, it uses the wrong function selector. The signature should be *isPaused()*.

**Team response**

Fixed.

**Mitigation Review**

Verified, the signature has been changed to *isPaused()*.

### TRST-M-2: *updateGasTokenPriceRatio()* uses an outdated price if TEA is worth exactly the fallback price

- **Category:** Logical flaws
- **Source:** GasPriceOracle.sol
- **Status:** Fixed

**Description**

In GasPriceOracle, *updateGasTokenPriceRatio()* checks if *teaPerETH()* is not equal to the fallback price to determine if the fallback price should be used:

```
// The oracle calculates the current price of 1e18 ETH in TEA (18 decimals).
uint160 currentPrice = teaPerETH();

// If the call didn't return the fallback price, it succeeded.
if (currentPrice != getFallbackPrice()) {
    _setLatestPrice(currentPrice);
} else {
    // If the call returned the fallback price, it failed.
    emit OracleReturnedFallbackPrice();

    // If the last result is from within the past 1 hour, keep it.
    // Otherwise, replace it with currentPrice (fallback)
    (uint96 lastUpdate, uint160 lastPrice) = getLatestPrice();
    if (currentPrice != lastPrice) {
        if (block.timestamp >= lastUpdate + MAX_ORACLE_DOWNTIME) {
            _setLatestPrice(currentPrice);
        }
    }
}
```

However, this check could wrongly fail if the price of TEA, which is queried from the Velodrome pool, happens to be exactly the fallback price. Should this occur, the price stored in **GasPriceOracle** will not be updated and the node ends up using an outdated price for the next hour.

As a result, L1 DA fees will be calculated incorrectly for the next hour.

**Recommendation**

A better implementation would be for *teaPerETH()* to also return a boolean indicating if the returned price is from the oracle or the fallback price, and *updateGasTokenPriceRatio()* uses that to determine if the price stored should be updated immediately.

**Team Response**

Fixed.

**Mitigation Review**

The issue has been fixed as suggested.

### TRST-M-3: Optimism's CGT-functionality is incompatible with governance tokens

- **Category:** Integration flaws
- **Source:** OptimismPortal2.sol
- **Status:** Acknowledged

**Description**

The protocol forks Optimism's Holocene release and uses TEA as its custom gas token (CGT).

In the event the CGT is also used as a governance token, the balance of **OptimismPortal** can be used to manipulate the results of votes. This is due to two facts:

1. When users deposit the CGT to L2, the native asset on L1 is held by **OptimismPortal**.
2. Anyone can perform an arbitrary call from the portal by sending a withdrawal transaction on L2.

As a result, any user on L2 can cast a governance vote on behalf of the portal on L1.

However, TEA token on L1 inherits **ERC20Votes**, which suggests it is planned to be used for governance in the future.

**Recommendation**

Ensure that TEA token on L1 is not used for governance.

**Team Response**

Accepted and acknowledged.

### TRST-M-4: *teaPerETH()* price will be outdated when the sequencer is down

- **Category:** Integration flaws
- **Source:** TeaWAPOracle.sol
- **Status:** Acknowledged

**Description**

In *teaPerETH()*, the price of TEA is *queried from a Velodrome pool using quote()*. This price is later stored in **GasPriceOracle** and used for L1 DA fee calculations.

However, when the L2 sequencer is down, especially for longer periods of time, using the TWAP price from *quote()* is unsafe because it can be manipulated or simply wrong for much longer periods of time. For example, users can forcefully include deposit transactions from L1 to swap through the pool and manipulate reserves.

As a result, the price of TEA will be inaccurate when the sequencer comes back up, until the pool is arbitraged to match the actual TEA price.

**Recommendation**

Consider checking if the sequencer was down in the last X seconds and using the fallback price if so.

**Team Response**

Acknowledged.

### TRST-M-5: GPG wallet does not enforce transaction ordering for signed transactions

- **Category:** Logical flaws
- **Source:** GPGWalletImpl.sol

- **Status:** Fixed

**Description**

In **GPGWalletImpl**, there is no transaction ordering mechanism. This means signed transactions can be executed in any order by paymasters (or any other user).

If a wallet has multiple signed transactions at one time, an attacker could re-order a signer's transactions to affect the final state of the wallet.

For example, *withdrawAll()* sends the wallet's entire balance to the **to** address. Consider the following scenario:

- There are two signatures, one to call *addSigner()* and another to call *withdrawAll()*.
- Signature to call *addSigner()* includes a paymaster fee of 10 TEA.
- If *addSigner()* is executed before *withdrawAll()*, the **to** address will get 10 TEA less.
- If *withdrawAll()* is executed before *addSigner()*, the **to** address receives the 10 TEA and *addSigner()* cannot be executed as the wallet has no funds remaining to pay the paymaster fee.

Therefore, the signer has no control over the result of his transactions. This is especially problematic as the wallet allows signed transactions to execute arbitrary calls.

**Recommendation**

Implement a transaction ordering mechanism for signed transactions, such as a nonce.

**Team Response**

Fixed.

**Mitigation Review**

The issue has been addressed by replacing the salt field in the signature with an ever-increasing nonce. It is recommended to remove the **usedDigests** mapping as **nonce** ensures signatures can't be reused.

**Team Response**

Fixed.

**Mitigation Review #2**

Verified, the **usedDigests** mapping has been removed.

## TRST-M-6: EIP-712 type hashes do not adhere to the specification

- **Category:** Implementation error
- **Source:** GPGWalletImpl.sol
- **Status:** Fixed

**Description**

In **GPGWalletImpl**, all EIP-712 type hashes contain spaces between their parameters. For example, the type hash for **AddSigner** is as shown:

```
bytes32 typehash = keccak256("AddSigner(address signer, uint256 paymasterFee, uint256 deadline, bytes32 salt)");
```

However, according to the EIP-712 specification, type hashes should not have spaces between their parameters.

**Recommendation**

Remove the spaces between parameters for the **AddSigner**, **WithdrawAll** and **Execute** type hashes.

**Team Response**

Fixed.

**Mitigation Review**

Issue has been correctly addressed.

### TRST-M-7: GPG precompile gas cost does not scale with input size

- **Category:** DOS attacks
- **Source:** protocol_params.go, contracts.go
- **Status:** Acknowledged

**Description**

The **gpgVerify** precompile has a fixed gas cost of 7900:

```
GpgVerifyGas  uint64 = 7900 // GPG signature verification gas price
```

```
// RequiredGas returns the gas required to execute the pre-compiled contract
func (c *gpgVerify) RequiredGas(input []byte) uint64 {
    return params.GpgVerifyGas
}
```

However, in *decodegpgVerifyInput()*, the precompile decodes input into two arbitrary-sized variables, **pubKey** and **signature**, which performs an expensive copy operation. Subsequently, both variables are fed into the PGP library, whose computation costs scale with the size of inputs:

```
// Create public keyring
pubKeyRing, err := pgpcrypto.NewKeyRing(pubKeyObj)
if err != nil {
    return nil, errInvalidPublicKey
}

// Create signature object
signatureObj := pgpcrypto.NewPGPSignature(signature)
```

As a result, the fixed gas cost of **gpgVerify** does not scale with its computation cost and creates a risk of the node being DOSed.

**Recommendation**

Consider calculating the gas cost of **gpgVerify** based on its input size, similar to the identity precompile. Additionally, implement a maximum array size for both **pubKey** and **signature**.

**Team Response**

Fixed, the gas cost of **gpgVerify** is now calculated as **23_500 + ((length - 3264) * 16)**.

**Mitigation Review**

From the benchmark docs, it's not clear if all signing algorithms were checked. As long as the list is not comprehensive, it is possible that attacker will find a scheme which is more expensive than RSA to cause insufficient gas spend.

**Team Response**

Acknowledged, the team will run a more complete benchmarking suite and verify internally before mainnet deployment.

### TRST-M-8: L1Fee in receipts is wrongly set for deposit transactions

- **Category:** Implementation error
- **Source:** state_processor.go
- **Status:** Fixed

**Description**

In *MakeReceipt()*, **receipt.L1Fee**, which represents the L1 DA fee, is set when *tx.IsDepositTx()* is true:

```
if tx.IsDepositTx() && config.IsOptimismRegolith(evm.Context.Time) {
    // ...

    if l1Cost := evm.Context.L1CostFunc(...); l1Cost != nil {
        receipt.L1Fee = l1Cost
    }
}
```

However, this is incorrect as deposit transactions do not have an L1 DA fee. Instead, **receipt.L1Fee** should be set for regular L2 transactions as they have an L1 DA fee.

As a result, transaction receipts will be incorrect for all node operators.

**Recommendation**

Set **receipt.L1Fee** when *tx.IsDepositTx()* is false instead.

**Mitigation Review**

Issue has been fixed as suggested.

## Low severity findings

### TRST-L-1: *airdropToAddresses()* transfers native assets using *.transfer()*

- **Category:** Implementation error
- **Source:** Airdropper.sol
- **Status:** Fixed

**Description**

In **Airdropper**, *airdropToAddresses()* transfers native assets using *.transfer()*:

```
for (uint256 i = 0; i < len; i++) {
    payable(addrs[i]).transfer(amounts[i]);
    emit AirdropToAddress(addrs[i], amounts[i]);
}
```

However, it is best practice to transfer native assets using *.call()* instead. Since *.transfer()* has a fixed gas cost of 2300, it might not be possible to transfer native assets to contracts with logic in their receive function.

**Recommendation**

Transfer native assets using *.call()* instead.

**Team Response**

Fixed.

**Mitigation Review**

Issue has been fixed as suggested.

### TRST-L-2: *keyId()* reads dirty bytes from free memory

- **Category:** Implementation error
- **Source:** GPGWalletImpl.sol
- **Status:** Fixed

**Description**

*keyId()* reads the wallet's key ID from code as shown:

```
bytes8 keyIdFromCode;
assembly {
    // Allocate memory for the bytes8
    let ptr := mload(0x40) // Get free memory pointer
    // Update the free memory pointer
    mstore(0x40, add(ptr, 0x20))
    // Copy the code to the pointer
    extcodecopy(address(), ptr, 0x2e, 0x08)
    // Load result into the bytes8 variable
    keyIdFromCode := mload(ptr)
}
return keyIdFromCode;
```

According to Solidity's documentation, "one should not expect the free memory to point to zeroed out memory".

However, the function copies 8 bytes into **ptr** to **ptr+0x8**, then loads 32 bytes at **ptr** into **keyIdFromCode**. Therefore, the lower 24 bytes of **keyIdFromCode** are from free memory and could be dirty. Fortunately, only the first 8 bytes are used as it is defined as **bytes8**. However, from the Solidity docs we learn that silently zeroing out dirty bytes during such conversion may change in future compiler versions, so it is highly recommended to explicitly zero out those values.

**Recommendation**

Copy 32 bytes from code using *extcodecopy*, instead of 8.

**Team Response**

Fixed.

**Mitigation review**

Issue has been addressed as suggested.

### TRST-L-3: Salt could contain dirty bytes when computing wallet addresses

- **Category:** Implementation error
- **Source:** GPGWalletImpl.sol
- **Status:** Fixed

**Description**

In *deploy()* and *predictAddress()*, the wallet's address is computed as shown:

```
// Copy create2 computation data to memory
mstore8(ptr, 0xff) // 0xFF
mstore(add(ptr, 0x35), keccak256(add(ptr, 0x55), bytecodeLength)) //
keccak256(bytecode)
mstore(add(ptr, 0x01), shl(96, address())) // deployer address
```

According to Solidity's documentation, "one should not expect the free memory to point to zeroed out memory".

The wallet's address is computed by hashing **ptr** to **ptr+0x55**. However, at this point in both functions, **ptr+0x21** to **ptr+0x35** (i.e. last 20 bytes of **salt**) is never written to. Therefore, the last 20 bytes of the salt is read from free memory and could be dirty bytes instead of 0x00.

**Recommendation**

Zero out the salt before computing the wallet address.

**Team Response**

Fixed.

**Mitigation review**

Issue has been addressed as suggested.

### TRST-L-4: receipt.L1GasUsed is uninitialized

- **Category:** Implementation error
- **Source:** receipt.go
- **Status:** Fixed

**Description**

In *DeriveFields()*, **L1GasUsed** is not initialized and left as **nil**:

```
// L1Fee will be set in MakeReceipt.
// L1GasUsed is deprecated as of Fjord, so will remain at 0.
// rs[i].L1Fee, rs[i].L1GasUsed = gasParams.costFunc(txs[i].RollupCostData())
```

As such, whenever any dereference operation on **L1GasUsed** occurs, the node will panic.

**Recommendation**

Set **L1GasUsed** to a pointer to 0 for safety.

**Team Response**

Fixed.

**Mitigation Review**

Fixed as suggested.

### TRST-L-5: Revert is encoded wrongly in *deploy()*

- **Category:** Implementation error

- **Source:** GPGWalletDeployer.sol
- **Status:** Fixed

**Description**

In *deploy()*, the following revert occurs if the wallet was not created successfully:

```
if iszero(eq(deployedAddress, walletAddress)) {
    mstore(0x89, 0x705f331c1) // `AccountCreationFailed(bytes8)`
    revert(0x89, 0xc) // keyId is already at 0x8D
}
```

However, there are multiple issues:

1. The 4-byte selector of "AccountCreationFailed(bytes8)" is **0x705f331c**, not **0x705f331c1**.
2. 0x89 is not the correct offset to write 4 bytes before **keyId** at 0x8D.
3. The free memory pointer is not added to the offsets.

**Recommendation**

Make the following change:

```
- mstore8(0x89, 0x705f331c1) // `AccountCreationFailed(bytes8)`
- revert(0x89, 0xc) // keyId is already at 0x8D
+ mstore(add(ptr, 0x6c), 0x705f331c) // `AccountCreationFailed(bytes8)`
+ revert(add(ptr, 0x88), 0xc) // keyId is already at 0x8C
```

This assumes the **keyId** is stored at 0x8C.

**Team Response**

Fixed.

**Mitigation Review**

The issue is not fully addressed, only the offset change has been applied.

**Team Response**

Fixed.

**Mitigation Review #2**

Verified, the recommendation above has been implemented.

### TRST-L-6: GasPriceOracle should not be integrated with for TEA prices

- **Category:** Integration flaws
- **Source:** GasPriceOracle.sol
- **Status:** Fixed

**Description**

Although **GasPriceOracle** stores the price of TEA to ETH, the price could be unreliable. For example, *convertETHToTea()* and *teaPerETH()* do not check if the price is outdated and could use the hardcoded fallback price.

Therefore, protocols should not integrate with **GasPriceOracle** to fetch the price of TEA.

**Recommendation**

Document that **GasPriceOracle** is not meant to be used in integrations.

**Team Response**

Fixed.

**Mitigation Review**

Issue has been addressed as suggested.

### TRST-L-7: Velodrome pool's reserves can be manipulated to use the fallback price

- **Category:** MEV attacks
- **Source:** TeaWAPOracle.sol
- **Status:** Acknowledged

**Description**

In *teaPerETH()*, the price of TEA is queried from a Velodrome pool if it contains more than **MIN_WETH_BALANCE** of WETH reserves:

```
// Use WETH reserves for this reliability, because it's the more stable token price.
uint256 wethReserves = wethT0 ? r0 : r1;
if (wethReserves < MIN_WETH_BALANCE) return fallbackPrice;
```

Otherwise, the protocol's hardcoded fallback price is used. However, an attacker can manipulate the pool's reserves to force *teaPerETH()* to use the fallback price for a block. For example:

- Attacker sends two transactions at almost the same time, at the same gas price:
  - Transaction 1 buys WETH from the pool to reduce its WETH reserves below **MIN_WETH_BALANCE**.
  - Transaction 2 sells the WETH bought back to the pool.
- Transaction 1 ends up as the last transaction at block N.
- In block N+1:
  - System transaction calls *teaPerETH()* to determine the price of TEA for that block, which uses the fallback price.
  - Transaction 2 is executed as the first transaction in the block.

As a result, the fallback price will be used to calculate L1 DA costs in block N, instead of the actual price of TEA.

**Recommendation**

Refactor to use time-weighted reserves, or document the risk.

**Team Response**

Acknowledged.

## Additional recommendations

### TRST-R-1: Improvements to code and comments

GasPriceOracle:

- GasPriceOracle.sol#L75 - This should say "MUST NOT" instead of "CAN NOT"
- GasPriceOracle.sol#L92-L93 - These conditions can be combined into one if-condition
- GasPriceOracle.sol#L62 - This line is over-indented

TeaWAPOracle:

- TeaWAPOracle.sol#L26-L27 – Consider adding a comment stating that fallback price is stored as **uint160**, otherwise it's easy to assume it's a **uint256** since the entire slot is available
- TeaWAPOracle.sol#L56 - Can be declared **external**
- TeaWAPOracle.sol#L84 - Explicitly declare **uint256** instead of **uint**
- TeaWAPOracle.sol#L117 - There is an extra space after **external**

Airdropper:

- Airdropper.sol#L17, Airdropper.sol#L41 - Both functions can be declared **external**
- Airdropper.sol#L32-L33 - **require** is missing an error message
- Both functions should check **msg.value** is equal to the sum of **amounts**

GPGWalletDeployer:

- GPGWalletDeployer.sol#L6 - **implementation** can be declared immutable
- GPGWalletDeployer.sol#L8 - Comment is missing end quote
- GPGWalletDeployer.sol#L9 - Missing visibility
- GPGWalletDeployer.sol#L45 - Using **iszero** twice is redundant
- GPGWalletDeployer.sol#L52 - Use **xor** instead of **iszero(eq())**
- GPGWalletDeployer.sol#L65 - The argument is a keyId, not a public key as the comment suggests

GPGWalletImpl:

- GPGWalletImpl.sol#L11 - Missing visibility
- GPGWalletImpl.sol#L190-L199 - Consider using the scratch space at 0x00 instead of allocating new memory
- Natspec for *addSigner()*, *withdrawAll()* and *executeWithSig()* is missing **pubKey**
- *addSigner(), withdrawAll(), executeBySigner()* and *executeWithSig()* can be declared **external**
- EIP-712 type hashes should be constants

### TRST-R-2: Minor issues in tea-geth

In contracts.go#L1453-L1459, explicitly initialize the arguments with **Indexed** as false. This is an important implicit behavior as having **Indexed** as true would skip the types during the unpacking:

```
// Create ABI arguments
arguments := abi.Arguments{
    {Type: bytes32Type},
    {Type: bytes8Type},
    {Type: bytesType},
    {Type: bytesType},
}
```

In contracts.go#L1467-L1470, the error message wrongly states 3 arguments instead of 4:

```
// Ensure we have the correct number of elements
if len(unpacked) != 4 {
    return [32]byte{}, [8]byte{}, nil, nil, fmt.Errorf("unexpected number of decoded
arguments: got %d, want 3", len(unpacked))
}
```

### TRST-R-3: Minor issues in gpg-wallet

In **GPGWalletImpl**, *addSigner()* should check the signer being added is not an existing signer. This prevents the wallet from unnecessarily paying a paymaster fee to add a signer that already exists.

Additionally, the **signers** mapping should be updated before calling *_payPaymaster()* to adhere to the CEI pattern:

```
+ signers[signer] = true;

  if (paymasterFee > 0) _payPaymaster(paymasterFee);

- signers[signer] = true;
```

### TRST-R-3: Minor issues in teaWAPOracle

**MIN_WETH_BALANCE** and **MAX_ORACLE_DOWNTIME** should be configurable values instead of constants. For example, the **MIN_WETH_BALANCE** might need to be adjusted based on the price of ETH.

A different namespace should be used for computing the storage slots at TeaWAPOracle.sol#L14-L27. This avoids a potential storage collision if a future Optimism upgrade ever happens to use the “opstack.customgastoken” namespace.

**Mitigation Review**

While introducing storage for the minWethBalance, a type confusion issue was introduced. The data is stored as uint16 and uint80 for observations and minimal balance respectively, but they are read as uint8, uint72.

While changing the storage slot namespace on the tea-optimism side, it has not been copied over to tea-geth.

**Team Response**

Fixed.

**Mitigation Review #2**

Issue has been addressed as suggested.

### TRST-R-4: Incorrect ERC-1667 implementation in GPGWalletDeployer

In *deploy()*, the following bytecode is used to deploy a minimal proxy that points to **GPGWalletImpl**:

```
mstore(add(ptr, 0x6c), 0x5af43d82803e903d91602b57fd5bf3) // ERC-1167 footer
mstore(add(ptr, 0x5d), sload(implementation.slot)) // implementation
mstore(add(ptr, 0x49), 0x3d603f80600a3d3981f3363d3d373d3d3d363d73) // ERC-1167 ...
calldatacopy(add(ptr, 0x8d), 0x04, 0x20)
```

However, there are multiple errors in this implementation.

**keyId** is copied from calldata into offset 0x8D, which leaves an empty byte at offset 0x8C. It should be offset 0x8C instead.

The third byte of the ERC-1167 constructor, which is the size of the proxy's runtime bytecode, is wrongly set to 0x3F. This appends additional zero bytes to the end of the proxy's bytecode. It should be 0x35 instead:

- 10 bytes for ERC-1167 header
- 20 bytes for implementation address
- 15 bytes for ERC-1167 footer
- 8 bytes for **keyId**

## Centralization risks

### TRST-CR-1: L2 proxy admin owner can manipulate TEA price

In **TeaWAPOracle**, the L2 **ProxyAdmin** owner can manipulate the price of TEA stored in **GasPriceOracle**. For example, they can:

- Set a malicious **oracle** address with *setOracleConfig().*
- Set an incorrect fallback price when the oracle fails.

As such, users must trust that the L2 **ProxyAdmin** owner does not manipulate the price of TEA, and by extension, the L1 data fee for regular L2 transactions.

Additionally, the owner must be trusted to correctly configure **twapObservations** so that the TWAP price of TEA cannot be manipulated through the Velodrome pool.

## Systematic risks

### TRST-SR-1: TeaChainID must be ensured to be unique

If TeaChainID (6122) is used by other chains, a transaction can be replayed in Tea. It should be verified that the chain ID is unique and remains that way through sites such as chainlist.org.

In fact, since the ID is currently not reserved in chainlist.org, there is already a race-condition with other chains as of writing.