// Smart Contract Security Assessment 01.20.2025 - 01.24.2025

Guru Fund

Guru

HALBORN

Guru Fund - Guru

Prepared by: HALBORN

Last Updated 02/17/2025

Date of Engagement by: January 20th, 2025 - January 24th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
21	0	1	0	9	11

TABLE OF CONTENTS

1. Introduction

Guru engaged Halborn to conduct a security assessment on their Guru Fund project beginning on January 20th, 2025 and ending on January 24th, 2025. The security assessment was scoped to the GuruFund code provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided five days for the engagement and assigned one full-time security engineer to review the security of the smart contract in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Guru team. The main ones were the following:

- In claimAbandonedFundsForBuybackAndBurn(), use WETH balance instead of ETH balance when unwrapping WETH.
- Make sure to enforce the minimum deposit value.
- Ensure the nonReentrant modifier is always the first modifier in all functions.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the contracts' solidity code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic-related vulnerability classes.
- Local testing with custom scripts (Hardhat and Foundry).
- Fork testing against main networks (Hardhat and Foundry).
- Static analysis of security for scoped contract, and imported functions.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: halborn-audit

(b) Assessed Commit ID: 4367c4d

(c) Items in scope:

- src/helpers/EIP712Helper.sol
- src/helpers/SwapHelper.sol
- src/helpers/TransferHelper.sol
- src/interfaces/IWETH.sol
- src/lib/Error.sol
- src/lib/FundAction.sol
- src/FundFactory.sol
- src/GuruFund.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 768cf9c
- 84d7e5f

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	9	11

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
THE FUNCTIONALITY TO CLAIM ABANDONED FUNDS FOR BUYBACK AND BURN COULD BE DOSED	HIGH	SOLVED - 02/10/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MINIMUM DEPOSIT VALUE NOT ENFORCED	LOW	RISK ACCEPTED - 02/12/2025
INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD BE THE FIRST	LOW	SOLVED - 02/10/2025
NON-UPGRADEABLE REENTRANCYGUARD USED	LOW	PARTIALLY SOLVED - 02/10/2025
EXTENDGRACEPERIOD() CAN SHORTEN GRACE PERIOD	LOW	PARTIALLY SOLVED - 02/10/2025
MISSING _DISABLEINITIALIZERS() CALL IN THE CONSTRUCTOR	LOW	SOLVED - 02/10/2025
MIX USAGE OF BLOCK.NUMBER AND BLOCK.TIMESTAMP	LOW	SOLVED - 02/10/2025
SIGNED PAYLOADS TO CREATE NEW FUNDS CAN BE REPLAYED	LOW	SOLVED - 02/11/2025
MISSING INPUT VALIDATION	LOW	RISK ACCEPTED - 02/13/2025
CENTRALIZATION RISKS	LOW	RISK ACCEPTED - 02/13/2025
SINGLE-STEP OWNERSHIP TRANSFER PROCESS	INFORMATIONAL	ACKNOWLEDGED - 02/12/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
OWNER CAN RENOUNCE OWNERSHIP	INFORMATIONAL	PARTIALLY SOLVED - 02/10/2025
USE OF AN UNLICENSED SMART CONTRACT	INFORMATIONAL	SOLVED - 02/10/2025
INCOMPLETE NATSPEC DOCUMENTATION	INFORMATIONAL	ACKNOWLEDGED - 02/13/2025
UNLOCKED PRAGMA COMPILER	INFORMATIONAL	SOLVED - 02/10/2025
MAGIC NUMBERS IN USE	INFORMATIONAL	ACKNOWLEDGED - 02/13/2025
UNUSED CUSTOM ERRORS AND IMPORTS	INFORMATIONAL	PARTIALLY SOLVED - 02/11/2025
STYLE GUIDE OPTIMIZATIONS	INFORMATIONAL	PARTIALLY SOLVED - 02/11/2025
CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	ACKNOWLEDGED - 02/13/2025
CACHE ARRAY LENGTH OUTSIDE OF LOOP	INFORMATIONAL	ACKNOWLEDGED - 02/13/2025
POTENTIALLY UNSAFE OR UNNECESSARY CASTINGS	INFORMATIONAL	ACKNOWLEDGED - 02/13/2025

7. FINDINGS & TECH DETAILS

7.1 THE FUNCTIONALITY TO CLAIM ABANDONED FUNDS FOR BUYBACK AND BURN COULD BE DOSED

// HIGH

Description

In `GuruFund.sol`, the `claimAbandonedFundsForBuybackAndBurn()` function attempts to unwrap `ETH` using the contract's ETH balance instead of its `WETH` balance:

```
/**
  * @notice After the grace period ends, the protocol owner can claim any
  * remaining funds to buyback and burn $GURU.
  */
function claimAbandonedFundsForBuybackAndBurn() external {
    require(
        !isOpen &&
            msg.sender == fundFactory.owner() &&
            block.timestamp > gracePeriodEnd,
        Error.Unauthorized()
    );
    _unwrapETH(address(this).balance);
    _safeTransferETH(fundFactory.guruBurner(), address(this).balance);
    emit AbandonedFundsClaimed();
}
```

The issue is that `_unwrapETH()` is called with `address(this).balance` (raw `ETH`) instead of `WETH.balanceOf(address(this))` (wrapped `ETH`). This is incorrect because:

- It is not possible to unwrap ETH that is not wrapped.
- The actual WETH balance might be different from the ETH balance.
- In the worst scenario, the function would revert if WETH balance < ETH balance.

Proof of Concept

An attacker could send some native funds to the GuruFund contract to prevent the protocol owner from successfully calling `claimAbandonedFundsForBuybackAndBurn()`.

In order to prove this, the "Should let the protocol owner claim the remaining funds (only after a 180 day grace period)" test was updated as follows:

```
it('DoS the protocol when trying to Claim Abandoned Funds For Buyback And B
    // Make sure the GuruFund is closed
    assert(await $OSHO.isOpen(), false)
```

```
    // Check how much ETH the GuruFund contract has
    const balanceETHBefore = await ethers.provider.getBalance($OSHO.target)
    console.log("balanceETHBefore: ", balanceETHBefore)

    // Check how much WETH the GuruFund contract has
    const wethBalance = await $WETH.balanceOf($OSHO.target)
    console.log("wethBalance: ", wethBalance)

    // Attacker sends some ETH so the GuruFund contract has just a bit more
    await anon1.sendTransaction({
        to: $OSHO.target,
        value: wethBalance + BigInt(1),
    })

    // Check how much WETH the GuruFund contract has now
    const balanceETHAfter = await ethers.provider.getBalance($OSHO.target)
    console.log("balanceETHAfter: ", balanceETHAfter)

    const GRACE_PERIOD_IN_SECONDS = 180 * 86400
    await ethers.provider.send('evm_increaseTime', [
        GRACE_PERIOD_IN_SECONDS,
    ])
    await ethers.provider.send('evm_mine')

    // factoryOwner tries to Claim Abandoned Funds For Buyback And Burn wit
    await expect(
        $OSHO
            .connect(factoryOwner)
            .claimAbandonedFundsForBuybackAndBurn()
    ).to.emit($OSHO, 'AbandonedFundsClaimed')
})
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:H/I:M/D:H/Y:N (7.1)

Recommendation

In `claimAbandonedFundsForBuybackAndBurn()`, use `WETH` balance instead of `ETH` balance:

```
/**
  * @notice After the grace period ends, the protocol owner can claim any
  * remaining funds to buyback and burn $GURU.
  */
function claimAbandonedFundsForBuybackAndBurn() external {
```

```
    require(
        !isOpen &&
            msg.sender == fundFactory.owner() &&
            block.timestamp > gracePeriodEnd,
        Error.Unauthorized()
    );

-   _unwrapETH(address(this).balance);
+   uint256 wethBalance = fundFactory.weth().balanceOf(address(this));
+   if (wethBalance > 0) _unwrapETH(wethBalance);

    _safeTransferETH(fundFactory.guruBurner(), address(this).balance);
    emit AbandonedFundsClaimed();
}
```

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by unwrapping the WETH balance as recommended.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.2 MINIMUM DEPOSIT VALUE NOT ENFORCED

// LOW

Description

The `GuruFund` contract declares a `minUserDepositValue` global state variable and sets it during fund initialization, but there is no on-chain enforcement to ensure that a user's deposit meets or exceeds this value when calling deposit functions. Although the protocol intends to collect a deposit fee as a percentage of the deposited amount, not having a minimum deposit check allows users to deposit extremely small amounts repeatedly. This can effectively minimize or circumvent meaningful fees, contrary to the intended fee structure. Consequently, the protocol may lose revenue and fail to discourage micro-deposits or spam.

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:N/I:L/D:M/Y:L (4.2)

Recommendation

Introduce a check in the deposit-related functions that reverts if the deposit's value falls below `minUserDepositValue`. This ensures the protocol's minimum deposit threshold is upheld, preventing trivial deposits from circumventing the intended fee model.

Remediation

RISK ACCEPTED: The **Guru team** accepted the risk of this finding.

7.3 INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD BE THE FIRST

// LOW

Description

The `nonReentrant` modifier is a critical security measure used in Solidity to prevent reentrancy attacks. It acts as a function-level lock, ensuring that a function cannot be called again before it finishes its execution. To maximize its effectiveness, the `nonReentrant` modifier must be placed as the first modifier in a function declaration. This ordering ensures that reentrancy protection is applied before any other logic or checks in other modifiers are executed.

Currently, some functions in the contract place other modifiers, such as `onlyOpen`, `onlyNotPaused` and `verifyingSignature`, before the `nonReentrant` modifier. This improper ordering can lead to scenarios where external calls in preceding modifiers might be exploited to bypass the reentrancy protection provided by `nonReentrant`.

For example, the `deposit()` function places `nonReentrant` after `onlyOpen` and `onlyNotPaused`, exposing it to a potential reentrancy risk:

```
function deposit(
    SignedPayload calldata _signedDepositPayload
)
    external
    payable
    onlyOpen
    onlyNotPaused
    nonReentrant
    verifyingSignature(_signedDepositPayload)
{
```

The `swapTokensForETH()` and `swapETHForTokens()`, `close()` and `depositAsset()` functions also follow this pattern. These functions interact with external addresses through the `onlyNotPaused` and `verifyingSignature` modifiers, further increasing the risk of reentrancy attacks.

BVSS

AO:A/AC:L/AX:M/R:P/S:C/C:N/A:M/I:M/D:M/Y:M (3.7)

Recommendation

Reorder the `nonReentrant` modifier to precede all other modifiers in the affected functions. This ensures that reentrancy protection is enforced before any additional checks or logic are applied.

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by making the `nonReentrant` modifier precede all other modifiers, as recommended.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.4 NON-UPGRADEABLE REENTRANCYGUARD USED

// LOW

Description

The `GuruFund` contract, which is meant to be deployed via cloning (EIP-1167 minimal proxy), inherits from OpenZeppelin's non-upgradeable `ReentrancyGuard` instead of `ReentrancyGuardUpgradeable`. In cloned contracts, constructors are not executed, which means the _status variable in ReentrancyGuard remains uninitialized.

See the contract declaration:

```
contract GuruFund is
  ReentrancyGuard,
  OwnableUpgradeable,
  SwapHelper,
  ERC20Upgradeable,
  TransferHelper
{
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

Replace the import and inheritance with the upgradeable version of the `ReentrancyGuard` module:

```
- import '@openzeppelin/contracts/security/ReentrancyGuard.sol';
+ import "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeal

contract GuruFund is
-   ReentrancyGuard,
+   ReentrancyGuardUpgradeable,
    OwnableUpgradeable,
    ...
{
    function initialize(...) external payable initializer {
        ...
+       __ReentrancyGuard_init();
        ...
    }
}
```

Remediation

PARTIALLY SOLVED: The **Guru team** partially fixed this finding in commit `768cf9cd` by using `ReentrancyGuardUpgradeable` instead of `ReentrancyGuard` as recommended. However, the old import of `ReentrancyGuard` was not removed, and the initializer function (`__ReentrancyGuard_init()`) is not called inside the `initialize()` function.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.5 EXTENDGRACEPERIOD() CAN SHORTEN GRACE PERIOD

// LOW

Description

In `GuruFund`, the `extendGracePeriod()` function allows the protocol owner to modify the grace period end time. Despite its name suggesting only extension is possible, the function actually allows setting any arbitrary timestamp, including past ones:

```
/**
  * @notice Extends the grace period.
  * @param _newGracePeriodEnd The new grace period end
  */
function extendGracePeriod(uint256 _newGracePeriodEnd) external {
  // Only protocol owner can extend the grace period
  require(msg.sender == fundFactory.owner(), Error.Unauthorized());
  gracePeriodEnd = _newGracePeriodEnd;
  emit GracePeriodExtended(_newGracePeriodEnd);
}
```

The affected function:

- Can set the new `gracePeriodEnd` to a timestamp lower than the current `gracePeriodEnd`.
- Can even set it to a past timestamp.
- Has a misleading name suggesting only extension is possible.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:H/Y:H (2.6)

Recommendation

Add validation to ensure the new grace period can only be extended and cannot be from the past.

Remediation

PARTIALLY SOLVED: The **Guru team** partially fixed this finding in commit `768cf9cd` by validating that the new grace period is greater than the current time. However, it is still possible to reduce the `gracePeriodEnd` global state variable by calling the `extendGracePeriod()` function. Consider ensuring that the new grace period is greater or modifying the function's name.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.6 MISSING _DISABLEINITIALIZERS() CALL IN THE CONSTRUCTOR

// LOW

Description

The `GuruFund` contract follows an upgradeable pattern, indirectly inheriting from the `Initializable` module from OpenZeppelin. In order to prevent leaving the contracts uninitialized, OpenZeppelin's documentation recommends adding the `_disableInitializers()` function in the `constructor` to automatically lock the contracts when they are deployed. However, all upgradeable contracts in scope are missing this function call.

This omission can lead to potential security vulnerabilities, as an uninitialized implementation contract can be taken over by an attacker, which may impact the proxy.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:P/S:U (2.5)

Recommendation

Consider adding a constructor and calling the `_disableinitializers()` method within the contracts to prevent the implementation from being initialized.

```
/**
 * @dev This will only be called once when deploying the Fund Factory.
 * Clones initializers will be called by the FundFactory.
 */
constructor() {
  fundFactory = FundFactory(msg.sender);
  _disableInitializers();
}
```

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by calling the `_disableInitializers()` in the constructor as recommended.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.7 MIX USAGE OF BLOCK.NUMBER AND BLOCK.TIMESTAMP

// LOW

Description

The contracts in scope predominantly uses `block.timestamp` for time-based checks but utilizes `block.number` in the `_verifyEIP712()` function to validate the `SignedPayload.expiresAt` field. However, the `SignedPayload` struct includes a comment stating that `expiresAt` is an "expiration timestamp," suggesting it should align with `block.timestamp`:

```
struct SignedPayload {
    ...
    /**
     * @notice Expiration timestamp of the payload
     */
    uint256 expiresAt;
}
```

The current check in `_verifyEIP712()` enforces `require(_payload.expiresAt >= block.number, ExpiredSignature());`, which relies on block numbers rather than timestamps. This discrepancy can cause confusion for integrators and may lead to unintended expiration behavior.

```
function _verifyEIP712(
    bytes32 _typeHash,
    address _account,
    SignedPayload calldata _payload
) internal view {
    require(_payload.expiresAt >= block.number, ExpiredSignature());
  ...
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

Maintain a consistent source of time-based validations throughout the contract. If the intent is to rely on actual time, switch to `block.timestamp` for signature expiration checks. Alternatively, if block-based expiration is intentional, update the variable name (e.g., `expiresBlock`) and ensure corresponding documentation reflects the block-based logic. This prevents confusion and preserves clear, uniform semantics for time-dependent operations.

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by updating the `expiresAt` documentation indicating that `block number` is used.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.8 SIGNED PAYLOADS TO CREATE NEW FUNDS CAN BE REPLAYED

// LOW

Description

When creating a new fund via the `createFund()` function, the contract verifies an off-chain signature using `_verifyEIP712()`:

```
/**
  * @notice Creates a new fund with a minimum deposit of 1000 USDT worth of
  * @param _name Name of the fund token
  * @param _symbol Symbol of the fund token
  * @param _signedPayload Signed payload containing expiration, signature, 
  * the encoded price feed with latest WETH/USDT price
  */
function createFund(
  string calldata _name,
  string calldata _symbol,
  SignedPayload calldata _signedPayload
) public payable whenNotPaused {
  verifySignature(msg.sender, _signedPayload);
  ...
```

```
function verifySignature(
    address account,
    SignedPayload calldata _signedPayload
) public view {
    _verifyEIP712(SIGNED_ACTION_TYPEHASH, account, _signedPayload);
}
```

However, the validation does not include any nonce or mechanism to mark signatures as “used.” As a result, as long as the signature remains valid (i.e., before `expiresAt`), the same signature payload can potentially be replayed by the same user to create multiple funds.

```
/**
  * @dev Verifies the signature
  * @param _typeHash Type hash
  * @param _account Address of the user the signature was signed for
  * @param _payload Signed payload containing the data, signature and expir
  */
function _verifyEIP712(
```

```
    bytes32 _typeHash,
    address _account,
    SignedPayload calldata _payload
  ) internal view {
    require(_payload.expiresAt >= block.number, ExpiredSignature()); //@audit
    require(
      SignatureChecker.isValidSignatureNow(
        signer,
        _hashTypedDataV4(
        keccak256(
          abi.encode(
            _typeHash,
            _account,
            keccak256(_payload.data),
            _payload.expiresAt
          )
        )
      ),
      _payload.signature
    ),
    InvalidSignature()
    );
  }
```

The development team clarified they intend for every new fund creation to have a unique signature, but the contract does not technically enforce this. Due to the short signature validity window (10 blocks, which is approximately 2 minutes under certain network conditions), the risk of this finding was reduced to **low**. Nevertheless, without a nonce or one-time-use scheme, replay attacks remain theoretically possible while the signature is still valid.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:M/D:H/Y:M (2.3)

Recommendation

Incorporate replay protection by requiring each signature to include a nonce and keeping track of previously used nonces in a contract-level mapping. Once a signature with a particular nonce has been consumed, subsequent attempts to use the same signature and nonce should revert. This ensures each off-chain signature can only be used once, preventing replay attacks even within the payload's expiration window.

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by adding a nonce to prevent replay attacks.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.9 MISSING INPUT VALIDATION

// LOW

Description

During the security assessment, it was identified that some functions in the smart contracts lack proper input validation, allowing critical parameters to be set to undesired or unrealistic values. This can lead to potential vulnerabilities, unexpected behavior, or erroneous states within the contract. Examples include:

- `FundFactory.sol`:
  - `vault = _vault;` and `guruBurner = _guruBurner;` (in the constructor) have no check against `address(0)`.
  - `setVault()`, `setGuruBurner()`, and other setter functions similarly do not validate address inputs.
  - Numeric parameters such as the ones in `setProtocolDepositFee()` or `setProtocolSwapFee()` have no upper/lower bound checks.
- `GuruFund.sol`:
  - `_updateAssets()` assigns `assets[_updates[i].index] = _updates[i].asset;` without verifying whether `asset` is the zero address.
  - The `transferShares()` function is not ensuring that the `to` address is not equal to `msg.sender`.
  - `updateMinUserDepositValue()` and `updateMinDepositCooldown()` have no upper/lower bound checks.

This list is not exhaustive. It is recommended to conduct a comprehensive review of the codebase to identify and assess other functions that may require additional input validation. Ensuring appropriate checks are in place for critical parameters will enhance the overall reliability, security, and predictability of the contracts.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:L/R:P/S:U (2.2)

Recommendation

To mitigate these issues, implement input validation in all constructor functions and other critical functions to ensure that inputs meet expected criteria. This can prevent unexpected behaviors and potential vulnerabilities.

Remediation

RISK ACCEPTED: The **Guru team** accepted the risk of this finding.

7.10 CENTRALIZATION RISKS

// LOW

Description

The protocol relies heavily on centralized components that could pose risks to users and the system's integrity. Specifically:

- **Owner Role:** A single `owner` account retains broad privileges (e.g., pausing the protocol, updating critical parameters), allowing centralized control over the DeFi platform's core functionalities. If compromised or misused, this role could disrupt or manipulate core operations (e.g., fee configurations, deposit/withdrawal mechanics).
- **Off-Chain Signer:** Key operations require an off-chain EIP-712 signature, which the Guru Fund contracts verify. This architecture places substantial trust in the security of the private key controlling the off-chain signer. If that private key is compromised, attackers could craft valid signatures to execute unauthorized fund actions, affecting deposits, withdrawals, and fund management.

Although these design choices can offer operational efficiency and rapid updates, they also create a central point of failure and vest considerable power in a single entity or small group. The development team acknowledges that private key management for the off-chain signer is outside the scope of the Solidity audit; however, this remains a security concern.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:M/Y:N (2.1)

Recommendation

It is recommended the following:

1. **Adopt Multisig or DAO-Based Governance:** Transition the `owner` role to a multi-signature wallet or a DAO/governance model to distribute control and reduce reliance on a single party.
2. **Enhance Off-Chain Signer Security:**
   - Use an institutional-grade Hardware Security Module (HSM) or a threshold-signature scheme (e.g., MPC) for the off-chain signer key.
   - Implement robust key management policies and regular key rotations to reduce the risk of permanent compromise.
3. **On-Chain Validation of Critical Operations:** Where feasible, shift critical or sensitive logic on-chain to reduce reliance on off-chain logic.
4. **Emergency Procedures and Transparency:**
   - Provide transparent documentation or a public dashboard to inform users of any centralized intervention (e.g., pausing the protocol).

- Plan a contingency strategy (e.g., timelock for changes or a publicly broadcasted freeze period) to mitigate abrupt or unannounced governance decisions.

By distributing control and securing off-chain components, the protocol can significantly lower its centralization risk and foster greater trust among participants.

Remediation

RISK ACCEPTED: The **B14G team** accepted the risk of this finding.

7.11 SINGLE-STEP OWNERSHIP TRANSFER PROCESS

// INFORMATIONAL

Description

It was identified that the `GuruFund` contract inherits from OpenZeppelin's `OwnableUpgradeable` library and `FundFactory` indirectly inherits from `Ownable` through `EIP712Helper`. Ownership of the contracts that are inherited from the `OwnableUpgradeable` and `Ownable` modules can be lost, as the ownership is transferred in a single-step process.

The address that the ownership is changed to should be verified to be active or willing to act as the `owner`. `Ownable2Step` and `Ownable2StepUpgradeable` are safer than `Ownable` and `OwnableUpgradeable` for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

To mitigate the risks associated with single-step ownership transitions and enhance contract security, it is recommended to adopt a two-step ownership transition mechanism, such as OpenZeppelin's `Ownable2Step` and `Ownable2StepUpgradeable`. This approach introduces an additional step in the ownership transfer process, requiring the new owner to accept ownership before the transition is finalized. The process typically involves the current owner calling a function to nominate a new owner, and the nominee then calling another function to accept ownership.

Implementing `Ownable2Step` and `Ownable2StepUpgradeable` provides several benefits:
1. **Reduces Risk of Accidental Loss of Ownership**: By requiring explicit acceptance of ownership, the risk of accidentally transferring ownership to an incorrect or zero address is significantly reduced.
2. **Enhanced Security**: It adds another layer of security by ensuring that the new owner is prepared and willing to take over the responsibilities associated with contract ownership.
3. **Flexibility in Ownership Transitions**: Allows for a smoother transition of ownership, as the nominee has the opportunity to prepare for the acceptance of their new role.

By adopting `Ownable2Step` and `Ownable2StepUpgradeable`, contract administrators can ensure a more secure and controlled process for transferring ownership, safeguarding against the risks associated with accidental or unauthorized ownership changes.

Remediation

ACKNOWLEDGED: The **Guru team** acknowledged this finding.

7.12 OWNER CAN RENOUNCE OWNERSHIP

// INFORMATIONAL

Description

It was identified that the `GuruFund` contract inherits from OpenZeppelin's `OwnableUpgradeable` library and `FundFactory` indirectly inherits from `Ownable` through `EIP712Helper`. In the `Ownable` and `OwnableUpgradeable` contracts, the `renounceOwnership()` function is used to renounce the `Owner` permission. Renouncing ownership before transferring would result in the contract having no owner, eliminating the ability to call privileged functions.

```
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby disabling any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Furthermore, the contract owner or single user with a role is not prevented from renouncing the role/ownership while the contract is paused, which would cause any user assets stored in the protocol, to be locked indefinitely.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:P/S:U (1.3)

Recommendation

It is recommended that the Owner cannot call `renounceOwnership()` without first transferring ownership to another address. In addition, if a multi-signature wallet is used, the call to the `renounceOwnership()` function should be confirmed for two or more users.

Remediation

PARTIALLY SOLVED: The **Guru team** partially fixed this finding in commit `768cf9cd` by using overriding the `renounceOwnership()` function in `GuruFund`. However, this was not modified in `FundFactory` (`EIP712Helper`).

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.13 USE OF AN UNLICENSED SMART CONTRACT

// INFORMATIONAL

Description

The `EIP712Helper` smart contract in scope is marked as unlicensed, as indicated by the SPDX license identifier at the top of the file:

```
1 | // SPDX-License-Identifier: UNLICENSED
```

Using unlicensed contracts can lead to legal uncertainties and potential conflicts regarding the usage, modification and distribution rights of the code. This may deter other developers from using or contributing to the project and could potentially lead to legal issues in the future.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is strongly recommended to apply the appropriate open-source license to the unlicensed smart contract.

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by correcting the license discrepancies.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.14 INCOMPLETE NATSPEC DOCUMENTATION

// INFORMATIONAL

Description

Although much of the code under review maintains a generally consistent NatSpec standard, there are notable gaps and inconsistencies in the documentation. Specifically:

- Several `public` functions across the codebase (e.g., all functions in `FundFactory` except `createFund()`) are missing NatSpec comments.
- Certain files, `Error.sol` and `IWETH.sol`, contain no NatSpec documentation at all.
- Files like `SwapHelper.sol`, `TransferHelper.sol`, and `FundFactory.sol` lack a standardized title and author format compared to `GuruFund.sol`.

These shortcomings in documentation can hinder a clear understanding of the contract logic, making it more difficult for external developers, users, and auditors to grasp the intended behavior, usage constraints, and potential edge cases.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Adopt a unified NatSpec documentation approach throughout the entire codebase. Specifically:
1. **Function-Level Annotations:** Provide full NatSpec annotations (`@notice`, `@dev`, `@param`, `@return`, etc.) for every function, detailing its purpose, parameter usage, and return data.
2. **File-Level Titles and Author Tags:** In all helper and core contracts, add a standardized title and author header for improved consistency and clarity.
3. **Comprehensive Coverage:** Incorporate NatSpec documentation for structs, errors, and interfaces (e.g., `Error.sol`, `IWETH.sol`) to ensure all code units are fully described.

By normalizing these practices, the overall clarity, maintainability, and auditability of the codebase is significantly enhanced.

Remediation

ACKNOWLEDGED: The **Guru team** acknowledged this finding.

7.15 UNLOCKED PRAGMA COMPILER

// INFORMATIONAL

Description

All files in scope currently use a fixed pragma version `=0.8.27`. However, the `TransferHelper` uses `^0.8.27`, which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.27` and less than `0.9.0`. It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Lock the pragma version of all files to the same version used during development and testing.

Remediation

SOLVED: The **Guru team** fixed this finding in commit `768cf9cd` by using a fixed version of Solidity in all contracts (`0.8.27`).

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/768cf9cd6d12462058d1fe70ea12357fe8dec65a

7.16 MAGIC NUMBERS IN USE

// INFORMATIONAL

Description

In programming, `magic numbers` refers to the use of unexplained numerical or string values directly in code, without any clear indication of their purpose or origin. The use of magic numbers can lead to confusion and make your code more difficult to understand, maintain, and update.

To improve the readability and maintainability of your smart contracts, it is recommended to avoid using magic numbers and instead use named constants or variables to represent these values. By doing so, you provide clear context for the values, making it easier for developers to understand their purpose and significance.

Three examples of magic numbers were found in `GuruFund.sol`:

```
fees <= (msg.value * fundFactory.protocolDepositFee()) / 100_000 &&
...
gracePeriodEnd = block.timestamp + 180 days;
...
return 6;
```

In `FundFactory.sol`, the use of magic numbers without scientific notation was also found in:

```
uint64 public minimumGuruInitialDepositValue = 1000_000000;
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

To improve code maintainability, readability, and reduce the risk of potential errors, it is recommended to replace magic numbers with well-defined constants. By using constants, developers can provide clear and descriptive names for specific values, making the code easier to understand and maintain. Additionally, updating the values becomes more straightforward, as changes can be made in a single location, reducing the risk of errors and inconsistencies. For large numbers, consider using scientific notation (e.g., `1e4`).

Remediation

ACKNOWLEDGED: The **Guru team** acknowledged this finding.

7.17 UNUSED CUSTOM ERRORS AND IMPORTS

// INFORMATIONAL

Description

During the security assessment of the smart contracts, some instances of unused custom errors and unused imports were found. Unused errors and imports can clutter the codebase, reducing readability and potentially leading to confusion during development or auditing. Additionally, unnecessary imports can slightly increase the compiled contract's bytecode size, potentially affecting deployment and execution costs.

Unused Errors

- In `FundFactory.sol`:

```
error InsufficientFirstDeposit(uint256 received, uint256 required);
```

- In `GuruFund.sol`:

```
error DepositMustIncreaseTVL();
```

Unused Imports

- In `FundFactory.sol`:

```
import '@openzeppelin/contracts/token/ERC20/ERC20.sol';
import '@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol';
import '@openzeppelin/contracts/proxy/utils/UUPSUpgradeable.sol';
import '@openzeppelin/contracts/utils/Context.sol';
```

- In `GuruFund.sol`:

```
import '@openzeppelin/contracts/utils/Context.sol';
...
import './interfaces/IWETH.sol';
import './helpers/EIP712Helper.sol';
```

- In `EIP712Helper.sol`:

```
import '@openzeppelin/contracts/token/ERC20/ERC20.sol';
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

For better clarity, consider using or removing all unused code. Keeping the code clean and relevant helps in maintaining a secure and efficient codebase.

Remediation

PARTIALLY SOLVED: The **Guru team** partially fixed this finding in commit `84d7e5f5` by reintroducing and using the custom error `DepositMustIncreaseTVL`. However, the `InsufficientFirstDeposit` is still unused and the unused imports are still present.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/84d7e5f599af8bde9bd6902a0ca23a0a24e43cb8

7.18 STYLE GUIDE OPTIMIZATIONS

// INFORMATIONAL

Description

The project codebase contains several stylistic inconsistencies and deviations from Solidity best practices, which, while not directly impacting functionality, reduce code readability, maintainability, and adherence to standard conventions. Addressing these inconsistencies can enhance the clarity and professionalism of the code.

Examples:

- **Use of `public` Where `external` Could Be Used:** Certain `public` functions could be declared as `external` to potentially save gas and adhere to best practices. Example in `FundFactory.sol`:

```
function setProtocolSwapFee(uint16 _newProtocolSwapFee) public onlyOwner {
```

- **Use of Whole File Imports Instead of Named Imports:** Full file imports are used across the codebase, which may include unnecessary code and reduce clarity. Example from `FundFactory.sol`:

```
import '@openzeppelin/contracts/token/ERC20/ERC20.sol';
```

- **Use of single quotes:** In Solidity, when working with strings, it is recommended to use double quotes rather than single quotes. Example from `FundFactory.sol`:

```
import '@openzeppelin/contracts/token/ERC20/ERC20.sol';
...
Ownable(msg.sender) EIP712Helper('GURU.FUND', 'v0.1.0', _offchainSigner) {
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended the following:

- **Reassess Function Visibility:**
  - Change public functions to `external` where appropriate for gas efficiency.
- **Use Named Imports:**
  - Change whole path imports to named imports to improve readability.

- **Strings should use double quotes rather than single quotes.**

Remediation

PARTIALLY SOLVED: The **Guru team** partially fixed this finding in commit `84d7e5f5` by double quotes rather than single quotes. However, multiple functions are still using `public` instead of `external` and no named imports are used.

Remediation Hash

https://gitlab.com/guru-fund/halborn-audit/-/commit/84d7e5f599af8bde9bd6902a0ca23a0a24e43cb8

7.19 CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

The project accepts using a Solidity compiler version greater than `0.8.18`, which supports named mappings. Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice helps developers and auditors understand the mappings' intent more easily.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit.

For example, on `GuruFund.sol`, instead of declaring:

```
mapping(address => uint256) public investedCapital;
```

It could be declared as:

```
mapping(address user => uint256 totalInvested) public investedCapital;
```

Remediation

ACKNOWLEDGED: The **Guru team** acknowledged this finding.

7.20 CACHE ARRAY LENGTH OUTSIDE OF LOOP

// INFORMATIONAL

Description

When the length of an array is not cached outside of a loop, the Solidity compiler reads the length of the array during each iteration. For **storage** arrays, this results in an extra `sload` operation (100 additional gas for each iteration except the first). For **memory** arrays, this results in an extra `mload` operation (3 additional gas for each iteration except the first).

Detecting loops that use the length member of a storage array in their loop condition without modifying it can reveal opportunities for optimization. See the following example in `SwapHelper.sol`:

```
function _executeSwaps(Swap[] memory _swaps) internal {
    for (uint8 i = 0; i < _swaps.length; i++) {
        _executeSingleSwap(_swaps[i]);
    }
}
```

Another instance in `GuruFund.sol` was found:

```
function _updateAssets(AssetIndex[] memory _updates) internal {
  for (uint8 i = 0; i < _updates.length; i++) {
    assets[_updates[i].index] = _updates[i].asset;
  }
  ...
```

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Cache the length of the (storage and memory) arrays in a local variable outside the loop to optimize gas usage. This reduces the number of read operations required during each iteration of the loop. See the example fix below:

```
function _executeSwaps(Swap[] memory _swaps) internal {
    uint256 _swapsLength = _swaps.length;
    for (uint8 i = 0; i < _swapsLength; i++) {
        _executeSingleSwap(_swaps[i]);
    }
}
```

Remediation

ACKNOWLEDGED: The **Guru team** acknowledged this finding.

7.21 POTENTIALLY UNSAFE OR UNNECESSARY CASTINGS

// INFORMATIONAL

Description

There are instances of both potentially unsafe and redundant type castings in the codebase. One instance of potential unsafe casting is:

```
int256(_initialDeposit.value),
```

This casting from `uint256` to `int256` could result in unexpected negative values if `_initialDeposit.value` is larger than `type(int256).max`.

One instance of redundant casting is the following:

```
investedCapital[owner()] = uint256(_initialDeposit.value);
```

Notice `_initialDeposit.value` is already of type `uint256`.

Score

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider using OpenZeppelin's `SafeCast` library to safely downcast integers. Alternatively, consider adding the following check:

```
require(rewardAmount <= type(uint192).max, "Amount exceeds uint192");
return uint192(rewardAmount);
```

For unnecessary casting, it would be recommended to remove it for better code clarity and gas optimization.

Remediation

ACKNOWLEDGED: The **Guru team** acknowledged this finding.

STATIC ANALYSIS REPORT

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Output

```
INFO:Detectors:
Contract locking ether found:
        Contract FundFactory (src/FundFactory.sol#17-180) has payable functions:
         - FundFactory.createFund(string,string,SignedPayload) (src/FundFactory.sol#150-179)
        But does not have a function to withdraw the ether
Contract locking ether found:
        Contract GuruFund (src/GuruFund.sol#34-805) has payable functions:
         - GuruFund.initialize(address,string,string,InitialDeposit) (src/GuruFund.sol#240-283)
         - GuruFund.deposit(SignedPayload) (src/GuruFund.sol#331-405)
         - GuruFund.receive() (src/GuruFund.sol#804)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether
INFO:Detectors:
FundFactory.constructor(address,address,address,IWETH)._vault (src/FundFactory.sol#55) lacks a zero-check on :
                - vault = _vault (src/FundFactory.sol#59)
FundFactory.constructor(address,address,address,IWETH)._guruBurner (src/FundFactory.sol#56) lacks a zero-check on :
                - guruBurner = _guruBurner (src/FundFactory.sol#60)
FundFactory.setVault(address)._newVault (src/FundFactory.sol#74) lacks a zero-check on :
                - vault = _newVault (src/FundFactory.sol#75)
FundFactory.setGuruBurner(address)._newGuruBurner (src/FundFactory.sol#80) lacks a zero-check on :
                - guruBurner = _newGuruBurner (src/FundFactory.sol#81)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.