our shielding · Your smart contracts, our shielding · Your smart c

shieldify

SteakHut

Liquidity V2

SECURITY REVIEW

Date: 2 February 2024

CONTENTS

1. About Shieldify

We are Shieldify Security – Revolutionizing Web3 security. Elevating standards with top-tier reports and a unique subscription-based auditing model.

Book a security review and learn more about us at shieldify.org or @ShieldifySec

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About SteakHut

SteakHut is a decentralized platform revolutionizing the liquidity landscape of DeFi, offering active liquidity management and market-making services across various concentrated liquidity AMMs (`CLMMs`). It caters to both retail and institutional liquidity providers, providing tailored solutions for on-chain liquidity with aggregated management across multiple DEXs and blockchains. Through its flexible and composable smart contracts, SteakHut empowers users with complete autonomy over their liquidity strategies.

Learn more about SteakHut's concept and the technicalities behind it here.

3.1. Observations

SteakHut's foundational components include the `Enigma` pool, its factory smart contract, the `Enigma Zapper` and helper libraries. All those display a layered structure with a clean separation of responsibilities. Each contract has a properly-defined role, enhancing modularity and upgradeability.

The protocol also focuses on efficient and optimized user experience. This is evident style of the code and the team's choice of chains on which the protocol will initially be deployed - `Arbitrum` and `Avalanche`.

3.2 Privileged Roles and Actors

The following section outlines the roles of the main contracts in scope:

- `EnigmaFactory.sol` - creates new Enigma pools, maintains the protocol fee percentage, blacklisting and whitelisting of pools for the UI.
- `Enigma.sol` - the heart of the protocol, containing the core logic. Handles all re-balancing, deposits and withdraws, and harvesting of the underlying v3 position. Inherits from `enigmaStorage.sol`.
- `EnigmaZapper.sol` - used as a helper to deposit into an underlying Enigma pool.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security assessment spanned 16 days, during which the four smart contract security researchers from the Shieldify team collectively dedicated 480 hours. The code exhibits professionalism and incorporates fundamental best practices aimed at mitigating common vulnerabilities. This security review contributed to the codebase quality by mitigating issues of varying severity. Some of those include risks of loss of user's funds, lack of slippage protection, and potential for price manipulation, among other, less detrimental attack vectors.

The contracts in scope utilize hooked-up requirements and libraries like OpenZeppelin, which is a positive indicator of code quality and security. The use of libraries like `SafeERC20` and `ReentrancyGuard` suggests a focus on safety.

The test coverage is robust. SteakHut Labs' prompt and effective communication significantly contributed to the quality of the audit report, and Shieldify expresses their appreciation for this aspect.

5.1 Protocol Summary

Project Name	**SteakHut**
Repository	enigma-contracts
Type of Project	DeFi, Liquidity Provider, CLMMs
Audit Timeline	16 days
Review Commit Hash	2ec77b1b38aa981143823568b1ccc6d668beeffe
Fixes Review Commit Hash	9ae5c079bcb0102c64dfc4ecfafe77151a2a1768

5.2 Scope

The following smart contracts were in the scope of the security review:

src/Enigma.sol	312
src/EnigmaFactory.sol	158
src/EnigmaZapper.sol	166
src/libs/EnigmaHelper.sol	177
src/abstract/EnigmaStorage.sol	149
Total	**962**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **2**
- **Medium** issues: **5**
- **Low** issues: **4**

ID	Title	Severity
[C-01]	Users Could Experience Fund Loss Due To **`Front-Running`** When Multiple Fee Tiers Are Present During Deposit Process	Critical
[H-01]	Usage Of **`slot0`** To Get **`sqrtPriceLimitX96`** Is Extremely Prone To Manipulation	High
[M-01]	The **`deposit()`**, **`withdraw()`** and **`_depositToEnigma()`** Functions - Trade Transactions Lack Expiration Timestamp / Deadline Check	Medium
[M-02]	Malicious User Can DoS The deposit Functionality Temporarily, By Front-Running The **`setMaxTotalSupply()`** Function	Medium
[M-03]	Enigma Vault Does Not Work With **`Fee-On-Transfer`** Tokens	Medium
[M-04]	The **`removeAllowedRouter()`** Functionality Is Broken	Medium
[M-05]	Insufficient Validation	Medium
[L-01]	The **`safeApprove()`** Function Could Revert For Non-Standard Token Like **`USDT`**	Low
[L-02]	Unsafe Call to **`decimals()`**	Low
[L-03]	Hardcoded YakRouter Address Won't Work For Multi-Chain Deployments	Low
[L-04]	Returned Variables from Function Calls in **`UniswapV3Pool`** Can Cause Underflow Reverting	Low

7. Findings

[C-01] Users Could Experience Fund Loss Due To `Front-Running` When Multiple Fee Tiers Are Present During Deposit Process

Severity

Critical Risk

Description

The **`deposit()`** function essentially allows users to deposit **`token0`** and **`token1`** into the Enigma pool, receive shares in return, and add liquidity to the Uniswap V3 pools based on the contract's strategy. The function also handles fee distribution and ensures that the operations adhere to the contract's rules and limits.

The **`deposit()`** function takes a **`DepositParams`** struct as input, it contains:

- **`amount0Desired`** - the desired amount of token0 to be spent
- **`amount1Desired`** - the desired amount of token1 to be spent
- **`amount0Min`** - the minimum amount of token0 to spend
- **`amount1Min`** - the minimum amount of token1 to spend

The problem is that the **`DepositParams`** struct lacks a minimum number of shares parameter that the user expects to receive. This allows a malicious actor to restrict the number of shares a user obtains in the following manner:

1. A user, Alice, initiates a transaction to **deposit** tokens into Enigma where (**amount0Desired**, **amount0Min**) is greater than (**amount1Desired**, **amount1Min**).
2. A malicious actor, Bob can front-run this transaction if there are multiple fee tiers. He does this by:
   - initially manipulating the price in **feeTier1** to devalue **token0** significantly (resulting in a high quantity of token0 in the pool).
   - subsequently altering the price in **feeTier2** in the opposite direction to devalue token1 significantly (leading to a high quantity of token1 in the pool).
3. This manipulation results in a balancing of liquidity across different **feeTiers**.

The calculations in the **getTotalAmounts()** method reflect this balance across fee tiers, leading to the outcomes - **total0** and **total1** being equally balanced. Therefore, the validation of **amount0Desired** and **amount1Desired** in the **deposit()** function is successful.

Impact

The user loses funds as a result of maximum slippage. Users may end up receiving fewer shares for their deposit than expected. The attack scenario described above leads to a reduced minting of vault shares. This reduction occurs because all the calculations in **getTotalAmounts()** are exaggerated due to the sizable swap. Consequently, the final value of **_sharesToMint** is significantly smaller than it was before the large swap.

Location of Affected Code

File: src/Enigma.sol#L98

```
function deposit(DepositParams calldata params) external virtual override
    nonReentrant returns (uint256 shares, uint256 amount0, uint256
   amount1) {
// code

//calculate the amount of shares to mint the user
   (uint256 total0, uint256 total1) = getTotalAmounts();
   (uint256 _sharesToMint, uint256 amount0Actual, uint256 amount1Actual) =
       EnigmaHelper.calcSharesAndAmounts(
    totalSupply(), total0, total1, params.amount0Desired, params.
       amount1Desired
   );

// code
}
```

Recommendation

Add an extra parameter in the **DepositParams** struct for the minimum number of shares that the user expects, instead of just checking for a non-zero amount.

File: `EnigmaStructs.sol`

```
struct DepositParams {
  uint256 amount0Desired;
  uint256 amount1Desired;
  uint256 amount0Min;
  uint256 amount1Min;
+ uint256 minSharesToMint;
  uint256 deadline;
  address recipient;
}
```

File: `Errors.sol`

```
+ error Enigma_MinimumSharesToMint();
```

File: `Enigma.sol`

```
function deposit(DepositParams calldata params)
  external
  virtual
  override
  nonReentrant
  returns (uint256 shares, uint256 amount0, uint256 amount1)
{
// code

//calculate the amount of shares to mint the user
  (uint256 total0, uint256 total1) = getTotalAmounts();

  (uint256 _sharesToMint, uint256 amount0Actual, uint256 amount1Actual) =
      EnigmaHelper.calcSharesAndAmounts(
      totalSupply(), total0, total1, params.amount0Desired, params.
         amount1Desired
  );
//check that shares are non-zero
  if (_sharesToMint == 0) revert Errors.Enigma_ZeroSharesAmount();

+ if (_sharesToMint < params.minSharesToMint) revert Errors.
   Enigma_MinimumSharesToMint();

// code
}
```

Team Response

Acknowledged and fixed as proposed.

[H-01] Usage Of `slot0` To Get `sqrtPriceLimitX96` Is Extremely Prone To Manipulation

Severity

High Risk

Description

In the `_calculatePriceFromLiquidity()` function in the **`EnigmaZapper.sol`** contract and **`getLiquidityForAmounts()`** and **`getAmountsForLiquidity()`** in **`EnigmaHelper`** the UniswapV3.slot0 is used to get the value of **`sqrtPriceX96`**, which is used to calculate the price of token0 and then performs the swap in **`performZap()`**.

The usage of **`slot0`** is extremely prone to manipulation. The slot0 in the pool stores many values, and is exposed as a single method to save gas when accessed externally. The data can change with any frequency including multiple times per transaction.

Impact

The **`sqrtPriceX96`** is pulled from **`Uniswap.slot0`**, which is the most recent data point and can be manipulated easily via MEV bots and Flashloans with sandwich attacks, which can cause the loss of funds when interacting with **`Uniswap.swap`** function. This could lead to wrong calculations and loss of funds for the protocol and other users.

Location of Affected Code

File: EnigmaZapper.sol

```
function _calculatePriceFromLiquidity(address _pool) internal view
   returns (uint256) {
  IUniswapV3Pool pool = IUniswapV3Pool(_pool);
// @audit-issue can be easily manipulated
  (uint160 sqrtPriceX96,,,,,,) = pool.slot0();

  uint256 _sqrtPriceX96_1 = uint256(sqrtPriceX96) * (uint256(sqrtPriceX96
     )) * (1e18) >> (96 * 2);
  return _sqrtPriceX96_1;
}
```

File: EnigmaHelper.sol

```
function getLiquidityForAmounts(address _pool, int24 tickLower, int24
   tickUpper, uint256 amount0, uint256 amount1) public view returns (
   uint128) {
// @audit-issue can be easily manipulated
  (uint160 sqrtRatioX96,,,,,,) = IUniswapV3Pool(_pool).slot0();
```

```
    return LiquidityAmounts.getLiquidityForAmounts(
      sqrtRatioX96,
      TickMath.getSqrtRatioAtTick(tickLower),
      TickMath.getSqrtRatioAtTick(tickUpper),
      amount0,
      amount1
    );
  }
```

```
function getAmountsForLiquidity(address _pool, int24 tickLower, int24
   tickUpper, uint128 liquidity) public view returns (uint256, uint256) {
// @audit-issue can be easily manipulated
  (uint160 sqrtRatioX96,,,,,,) = IUniswapV3Pool(_pool).slot0();
  return LiquidityAmounts.getAmountsForLiquidity(
    sqrtRatioX96, TickMath.getSqrtRatioAtTick(tickLower), TickMath.
       getSqrtRatioAtTick(tickUpper), liquidity
  );
}
```

Recommendation

Use the `TWAP` function instead of `slot0` to get the value of `sqrtPriceX96`. `TWAP` is a pricing algorithm used to calculate the average price of an asset over a set period. It is calculated by summing prices at multiple points across a set period and then dividing this total by the total number of price points.

Team Response

Acknowledged.

[M-01] The `deposit()`, `withdraw()` and `_depositToEnigma()` Functions - Trade Transactions Lack Expiration Timestamp / Deadline Check

Severity

Medium Risk

Description

In the `_depositToEnigma()` function the `deadline` parameter was initially hard-coded to `block.timestamp` which offers no protection since a validator can hold the transaction and the block it is eventually put into and the `deadline` parameter will still be `block.timestamp`.

After the initial pre-review, it is a user-provided parameter in the `performZap()` function but there is no validation for it. It is recommended to add the corresponding `deadline` parameter validation in the `deposit()` function.

In the `withdraw()` function in `Enigma.sol` there is no user-supplied `deadline` parameter. By not providing any deadline check, if the transaction has not been confirmed for a long time then the user might end up with a position that is not as interesting as it was. Eventually, the transaction can be confirmed but the position is not in profit anymore because the price changed during that time.

Location of Affected Code

File: src/EnigmaZapper.sol#L205

```
function _depositToEnigma(address _token0, address _token1, address
    _enigmaPool)
  internal
  returns (uint256 shares)
{
// code

DepositParams memory _depParams = DepositParams({
  amount0Desired: bal0,
  amount1Desired: bal1,
  amount0Min: 0,
  amount1Min: 0,
  deadline: block.timestamp,
  recipient: msg.sender
});

// code
}
```

File: src/Enigma.sol

```
function deposit(DepositParams calldata params) external virtual override
     nonReentrant returns (uint256 shares, uint256 amount0, uint256
    amount1) {
function withdraw(uint256 shares, address to) external nonReentrant
    returns (uint256 amount0, uint256 amount1) {
```

Recommendation

Implement the following changes:

File: `EnigmaZapper.sol`

```
- function _depositToEnigma(address _token0, address _token1, address
    _enigmaPool)
+ function _depositToEnigma(address _token0, address _token1, address
    _enigmaPool, uint256 _deadline)

// code
```

```
DepositParams memory _depParams = DepositParams({
  amount0Desired: bal0,
  amount1Desired: bal1,
  amount0Min: 0,
  amount1Min: 0,
- deadline: block.timestamp,
+ deadline: _deadline,
  recipient: msg.sender
});

// code
```

File: `Errors.sol`

```
+ error Deadline_Expired();
```

File: `Enigma.sol`

```
function deposit(DepositParams calldata params) external virtual override
    nonReentrant returns (uint256 shares, uint256 amount0, uint256
   amount1) {
// code

+ if (params.deadline < block.timestamp) revert Deadline_Expired();

// code
}

- function withdraw(uint256 shares, address to) external nonReentrant
   returns (uint256 amount0, uint256 amount1) {
+ function withdraw(uint256 shares, address to, uint256 deadline)
   external nonReentrant returns (uint256 amount0, uint256 amount1) {
// code

+ if (params.deadline < block.timestamp) revert Deadline_Expired();

// code
}
```

Team Response

Acknowledged and fixed as proposed.

[M-02] Malicious User Can DoS The deposit Functionality Temporarily, By Front-Running The `setMaxTotalSupply()` Function

Severity

Medium Risk

Description

The `maxTotalSupply` state variable is responsible for setting and limiting the maximum total supply of the Enigma Vault. The operator of the Enigma contract can change the max total supply cap via the `setMaxTotalSupply()` function.

Attack Scenario

The vulnerability revolves around the interaction between users executing the `deposit()` function and the operator - `setMaxTotalSupply()`, which allows the operator to adjust the maximum total supply limit/cap. The process unfolds as follows:

1. The current `maxTotalSupply` is set to 10_000.
2. Users deposit a total of 2_000.
3. The Operator initiates a transaction to update the `maxTotalSupply` cap to 5_000.
4. A malicious user identifies this transaction and front-runs it by depositing 5_000 into the Enigma Vault via the `deposit()` function.
5. The malicious user's transaction executes first, resulting in an actual total supply of 7_000 for Enigma Vault.
6. Subsequently, the Operator transaction executes, setting the `maxTotalSupply` to 5_000.

As a result, the malicious user has effectively manipulated the `maxTotalSupply` cap for the Enigma Vault, causing it to be lower than the actual total supply.

Impact

The outcome of the scenario above is as follows:

1. Can manipulate the `maxTotalSupply` limit for the Enigma Vault, causing it to be lower than the actual total supply.
2. Can prevent legitimate users from depositing, which effectively breaks the deposit function, due to this check - `if (maxTotalSupply > 0 && totalSupply()> maxTotalSupply)revert Errors.Enigma_MaxTotalSupply();`.
3. Violate the common protocol and operator wants/requirements for the `maxTotalSupply` cap. (to restore the deposit function, the operator needs to increase `maxTotalSupply`).

Finally, yes, a malicious user puts liquidity into the `Enigma Vault`, but disrupts other users by disrupting the deposit functionality and also profits from swap fees.

Ultimately, a malicious user adds liquidity to the Enigma Vault, causing disruptions for other users by interfering with the deposit functionality, while profiting from swap fees during this disruptive activity.

Location of Affected Code

File: EnigmaStorage.sol#L203

```
function setMaxTotalSupply(uint256 _maxTotalSupply) external onlyOperator
    {
    if (maxTotalSupply != _maxTotalSupply) {
        maxTotalSupply = _maxTotalSupply;
        emit Log_MaxTotalSupplySet(_maxTotalSupply);
    }
}
```

File: src/Enigma.sol#L150

```
function deposit(DepositParams calldata params) external virtual override
     nonReentrant returns (uint256 shares, uint256 amount0, uint256
    amount1) {
// code

// Check total supply cap not exceeded. A value of 0 means no limit.
     if (maxTotalSupply > 0 && totalSupply() > maxTotalSupply) revert
         Errors.Enigma_MaxTotalSupply();

// code
}
```

Recommendation

Consider implementing a delay between changes made from both the **`deposit()`** and **`setMaxTotalSupply()`** functions.

Team Response

Acknowledged.

[M-03] Enigma Vault Does Not Work With `Fee-On-Transfer` Tokens

Severity

Medium Risk

Description

Currently, the contract is not compatible with tokens that have a fee-on-transfer. This problem affects not only the deposit function, where users might receive more shares than the actual number of tokens received by the Enigma Vault but also poses a problem for the withdrawal function.

Impact

It could lead to fund loss if a token with a fee-on-transfer mechanism is used and not properly handled in Enigma Vault, it can result in stuck balances of this token of users. Such tokens for example are **`PAXG`**, while **`USDT`** has a built-in fee-on-transfer mechanism that is currently switched off.

Attack Scenario

1. Alice attempts to withdraw **`10,000e18`** shares from the vault using the **`withdraw()`** function.
2. The strategy separates the corresponding amount of liquidity tokens into **`token0`** and **`token1`**.
3. The vault initiates a transfer of the respective **`amount0`** and **`amount1`** from the strategy contract to the user. However, if the tokens include a fee on transfer, there's a possibility that the strategy might not have enough tokens to complete the transfer.
4. Due to the shortfall in tokens caused by the transfer fee, the **`withdraw`** function ultimately fails and reverts.

Recommendation

As the protocol is intended to support any `ERC20` token, it is recommended to check the balance before and after the transfer and validate if the result is the same as the amount argument provided.

Team Response

Acknowledged and decided to update the whitepaper.

[M-04] The `removeAllowedRouter()` Functionality Is Broken

Severity

Medium Risk

Description

The `removeAllowedRouter()` function implements a check if the passed `_router` is included in `_allAllowedRouters`, but if it is included the function reverts with `Enigma__RouterWhitelisted`. In the other case when it does not exist the `_remove()` function from `EnumerableSet.sol` will return false and therefore the router can not be removed.

Location of Affected Code

File: src/EnigmaFactory.sol#L211

```
// @notice removes a allowed router for rebalancing swaps
function removeAllowedRouter(address _router) external onlyOwner {
  if (_allAllowedRouters.contains(_router)) revert
     Enigma__RouterWhitelisted(_router);
  _allAllowedRouters.remove(_router);
  emit RouterAllowed(_router, false);
}
```

Recommendation

Fix the check in the following way:

```
// @notice removes a allowed router for rebalancing swaps
function removeAllowedRouter(address _router) external onlyOwner {
- if (_allAllowedRouters.contains(_router)) revert
   Enigma__RouterWhitelisted(_router);
+ if (!_allAllowedRouters.contains(_router)) revert
   Enigma__RouterWhitelisted(_router);
  _allAllowedRouters.remove(_router);
  emit RouterAllowed(_router, false);
}
```

Team Response

Acknowledged and fixed as proposed.

[M-05] Insufficient Validation

Severity

Medium Risk

Description

Some functions lack complete validation for input parameters, specifically for address-type parameters. Here is a detailed explanation of these functions:

- The `initialize()` and `deployEnigmaPool()` functions lack validation for the `_operator` parameter.
- The `setSelectedFee()` function allows the operator to set the `SELECTED_FEE` which is the max fee that can be taken by the enigma, however, the `FEE_LIMIT` is 100%.
- The `_setSelectedFee()` function in the `EnigmaStorage.sol` contract does not check that the `_newSelectedFee` value is different from the old `SELECTED_FEE`.
- The `setEnigmaFee()` function does not check that the new enigma fee is different from the old `ENIGMA_TREASURY_FEE`.
- The `setFactoryCap()` function does not check that the `_factoryCap` value is different from the old `ENIGMA_CAP`.
- The `setMaxPositions()` function does not check that the `_maxPositions` value is different from the old `ENIGMA_MAX_POS`.
- The `setEnigmaFee()` function allows the owner of the `EnigmaFactory` contract to set `ENIGMA_TREASURY_FEE` to up to 100%. This manipulation could potentially divert up to 100% of the pending fees in favor of the Enigma Treasury in the `_applyFeesDistribute()` method.
- In the `deposit()` function, a user will not be able to deposit the max amount of any of the tokens via `deposit0Max` or `deposit1Max`, because the if check will revert if the desired amount is `>=`.
- In the `sortTokens()` function in the `EnigmaHelper.sol` contract there is a redundant zero-address check.
- The `setFactoryCap()`, `setMaxPositions()`, `setMaxTotalSupply()` and `setDepositMax()` lacks any lower and upper-bound validation checks.

Recommendations

- Ensure that all parameters are properly validated in all methods, and use the setter functions when possible (for example `setOperator()` function).
- Consider implementing a check that the new values are different from the old ones in the `setEnigmaFee()`, `setFactoryCap()` and `setMaxPositions()` and `_setSelectedFee()` methods.
- Consider setting a lower upper boundary for the `FEE_LIMIT` for `SELECTED_FEE` and `ENIGMA_TREASURY_FEE` .
- Fix the check for `deposit0Max` and `deposit1Max` amounts in the `deposit()` function as follows:

```
- if (params.amount0Desired >= deposit0Max || params.amount1Desired >=
  deposit1Max)
+ if (params.amount0Desired > deposit0Max || params.amount1Desired >
  deposit1Max)
```

- Consider removing the redundant check in `sortTokens()`.
- Consider adding validation in `setFactoryCap()`, `setMaxPositions()`, `setMaxTotalSupply()` and `setDepositMax()`.

Team Response

Acknowledged and fixed most of the bullets.

[L-01] The `safeApprove()` Function Could Revert For Non-Standard Token Like `USDT`

Severity

Low Risk

Description

Some non-standard tokens like USDT will revert when a contract or a user tries to approve an allowance when the spender allowance has already been set to a non-zero value.

In the current code we have not seen any real problem with this fact because after transferring the amount, the `safeApprove()` function was called again and set to 0. However, if the approval is not lowered to exactly 0 (due to a rounding error or another unforeseen situation) then the next approval will fail (assuming a token like `USDT` is used), blocking all further deposits.

We also should note that `OpenZeppelin` has officially deprecated the `safeApprove()` function, suggesting to use instead of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`.

Location of Affected Code

File: src/Enigma.sol

File: src/EnigmaZapper.sol

Recommendation

Consider replacing deprecated functions, the official `OpenZeppelin` documentation recommends using `safeIncreaseAllowance()` & `safeDecreaseAllowance()`.

Team Response

Acknowledged.

[L-02] Unsafe Call to `decimals()`

Severity

Low Risk

Description

The `decimals()` function is optional in the initial ERC20 and might fail for old tokens that do not implement it.

Location of Affected Code

File: src/EnigmaZapper.sol#L113-L114

```
uint256 _token0Decimals = ERC20(zapParams.token0).decimals();
uint256 _token1Decimals = ERC20(zapParams.token1).decimals();
```

Recommendation

Here is how to fix the issue by leveraging BoringSolidity's `safeDecimals()` function:

```
/// @notice Provides a safe ERC20.decimals version which returns '18' as
   fallback value.
/// @param token The address of the ERC-20 token contract.
/// @return (uint8) Token decimals.
function safeDecimals(IERC20 token) internal view returns (uint8) {
  (bool success, bytes memory data) = address(token).staticcall(abi.
     encodeWithSelector(SIG_DECIMALS));
  return success && data.length == 32 ? abi.decode(data, (uint8)) : 18;
}
```

Team Response

Acknowledged and fixed as proposed.

[L-03] Hardcoded YakRouter Address Won't Work For Multi-Chain Deployments

Severity

Low Risk

Description

The `yakRouter` address is hardcoded in the `EnigmaZapper.sol` contract:

```
IYakRouter public yakRouter = IYakRouter(0
   xC4729E56b831d74bBc18797e0e17A295fA77488c);
```

However, the StakeHut documentation states, that the protocol will be deployed on other chains like `Arbitrum` (currently, the only supported chain is `Avalanche`).

So, the protocol functionality will not work because of the hardcoded `yakRouter` address in the `EnigmaZapper.sol` contract.

Location of Affected Code

File: src/EnigmaZapper.sol#43

```
IYakRouter public yakRouter = IYakRouter(0
   xC4729E56b831d74bBc18797e0e17A295fA77488c);
```

Recommendation

To address this vulnerability, it is highly recommended to remove the hardcoded `yakRouter` address and replace it with an immutable variable contract that is passed as an argument in the constructor during contract deployment.

Team Response

Acknowledged and fixed as proposed.

[L-04] Returned Variables from Function Calls in `UniswapV3Pool` Can Cause Underflow Reverting

Severity

Low Risk

Description

In the `_burnLiquidity()` function, the tokens owed for the liquidity are represented as `collect0` or `collect1`. If any of those variables is `0`, one of the `withdrawPayload.fee` will underflow, causing a revert during its calculation. The same issue is valid in the `_applyFeesDistribute()` function for its `enigmaFee0` `enigmaFee1`, `operatorFee0` and `operatorFee1` variables.

Location of Affected Code

File: src/Enigma.sol

```
function _burnLiquidity(BurnParams memory _burnParams) internal returns (
    WithdrawParams memory withdrawPayload) {
function _applyFeesDistribute(uint256 fee0_, uint256 fee1_) internal
    returns (uint256 fee0, uint256 fee1) {
```

Recommendation

Refactor the code to prevent potential overflow in `_burnLiquidity()` and `_applyFeesDistribute()` methods. These functions should be non-blocking as it is crucial for the `rebalance` functionality.

Team Response

Acknowledged and fixed as proposed.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!