g8keep Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, 0xbepresent, Klaus

December 12th 2024 - December 18th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **g8keep/audit-v2** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About g8keep

The g8keep contracts facilitate a token sale using a bonding curve, starting with an initial liquidity setup that transitions to a secondary phase involving penalty-related token prices, while initializing a Uniswap V3 pool at the onset to prevent outside manipulation. The contract maintains strict conditions on liquidity and token volume and uses a scripted migration to ensure accurate pricing.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 7ed875cf8104c6267be505ca8c31372448a6f16e

fixes review commit hash - 7759b78293329222f64a196d332f5d185e74310e

Scope

The following smart contracts were in scope of the audit:

- `Tickmath`
- `g8keepBondingCurve`
- `g8keepBondingCurveCode`
- `g8keepBondingCurveFactory`
- `g8keepBondingCurveFactoryConfiguration`
- `g8keepLiquidityLocker`
- `g8keepLockerFactory`

7. Executive Summary

Over the course of the security review, 0xunforgiven, 0xbepresent, Klaus engaged with g8keep to review g8keep. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	g8keep
Repository	https://github.com/g8keep/audit-v2
Date	December 12th 2024 - December 18th 2024
Protocol Type	Bonding Curve Tokensale

Findings Count

Severity	Amount
High	3
Medium	4
Low	9
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Migration will fail trying to swap 1 wei of tokens to reach the target price	High	Resolved
[H-02]	Incorrect reserve assignment leads to reserve mismatch and fund mismanagement	High	Resolved
[H-03]	It's possible to add liquidity to UniswapV3 while token is not migrated	High	Resolved
[M-01]	Incorrect calculation in maxBuyWithoutPenalty	Medium	Resolved
[M-02]	Code should set ethB value in _curveBuy() when curveLiquidityMet is true	Medium	Resolved
[M-03]	Potential misuse of forceSafeTransferETH	Medium	Resolved
[M-04]	Liquidity shortfall during failed migration	Medium	Acknowledged
[L-01]	Forcing token deployer to purchase the initial supply	Low	Resolved
[L-02]	withdrawETH charges fees	Low	Resolved
[L-03]	Some tokens may be locked in token contract	Low	Resolved
[L-04]	STATUS_VOLUME_FLAG may not be set at the correct moment due to off by one error	Low	Resolved
[L-05]	getCurveStatus returns incorrect values	Low	Resolved

[L-06]	Function _getAmountIn() should round up when calculating the input token amount	Low	Acknowledged
[L-07]	Attacker can DOS token creation by front-running	Low	Acknowledged
[L-08]	Compromised G8KEEP_FEE_WALLET can disrupt buy/sell operations	Low	Resolved
[L-09]	Token deployer can avoid g8keep fees	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Migration will fail trying to swap 1 wei of tokens to reach the target price

Severity

Impact: Medium

Likelihood: High

Description

When the code wants to migrate tokens to Uniswap V3 pool, first it tries to change the pool's price by adding temporary liquidity and performing a swap in `_adjustPoolPrice()`:

```
uint256 tmpLPTokenId =
            _addTemporaryLiquidity(
              ethAmount/100,
              tokenAmount/100,
              constainedTickLower,
              constrainedTickUpper
            );

        if (sqrtPriceX96Start > sqrtPriceX96) {
            IUniswapV3Pool(poolAddress).swap(address(this), true, int256
              (uint256(tokenAmount)), sqrtPriceX96, "");
        } else {
            IUniswapV3Pool(poolAddress).swap(address(this), false, int256
              (uint256(1)), sqrtPriceX96, "");
        }
----snip
        (uint160 sqrtPriceX96New,,,,,,) = IUniswapV3Pool(poolAddress).slot0();
        adjustFailed = sqrtPriceX96 != sqrtPriceX96New;
```

And if the pool's price doesn't reach the target price then the code doesn't migrate. The issue is that the code tries to swap with 1 wei of ETH and it won't be enough to reach the target price the price would only reach the `constrainedTickUpper` and as a result the `adjustFailed` would be true and migration can't be completed and it would revert always.

Recommendations

Calculate the exact amount required to reach that target price level and use it in the swap and also allows for some percentage of error when checking the pool's price and target price.

[H-02] Incorrect reserve assignment leads to reserve mismatch and fund mismanagement

Severity

Impact: High

Likelihood: Medium

Description

Upon handling a failed migration in the `g8keepBondingCurve::_handleMigrationFailed` function, the reserves for ETH and tokens are incorrectly assigned. The `tokenAmount` is wrongly assigned to `bReserve.reserve0`, which is intended for the native token (ETH), and the `ethAmount` is assigned to `bReserve.reserve1`, which should hold the token amount.

```
File: g8keepBondingCurve.sol
        function _handleMigrationFailed
  (uint256 ethAmount, uint112 tokenAmount) internal {
            curveStatus = curveStatus | STATUS_MIGRATION_FAILED_FLAG;
            WETH.withdraw(WETH.balanceOf(address(this)));

@>>         bReserve.reserve0 = tokenAmount;
@>>         bReserve.reserve1 = uint112(ethAmount);

            emit MigrationFailed();
        }
```

This reversed assignment can lead to incorrect reserve calculations and mismatches during buy and sell operations. Since buys and sells can still occur after a migration failure, this issue can create financial discrepancies and allow manipulation of trades.

Recommendations

Update the assignments in the `_handleMigrationFailed` function to correctly assign `ethAmount` to `bReserve.reserve0` and `tokenAmount` to `bReserve.reserve1`.

[H-03] It's possible to add liquidity to UniswapV3 while token is not migrated

Severity

Impact: High

Likelihood: Medium

Description

Code won't allow to transfer of tokens to the UniswapV3 pool when the token hasn't migrated yet so no one could add liquidity to the pool and manipulate the pool. The issue is that it's possible to bypass these checks and add liquidity to the Uniswap pool. The attacker can use `buy(to)` function to buy tokens for the Uniswap V3 pool address and increase the pool's token balance. When adding liquidity to Uniswap V3, it calls `uniswapV3MintCallback()` and expects that function to transfer the tokens to the pool's address:

```
if (amount0 > 0) balance0Before = balance0();
        if (amount1 > 0) balance1Before = balance1();
        IUniswapV3MintCallback(msg.sender).uniswapV3MintCallback
          (amount0, amount1, data);
        if (amount0 > 0) require(balance0Before.add(amount0) <= balance0
          (), 'M0');
        if (amount1 > 0) require(balance1Before.add(amount1) <= balance1
          (), 'M1');
```

The attacker's contract can call `mint()` to add liquidity to the pool and during the `uniswapV3MintCallback()` callback it would call `buy(pool)` to increase the pool's balance and as a result, Uniswap V3 pool's liquidity would increase. By performing this attacker can DOS the migration process and also manipulate the token price during the migration.

Recommendations

Won't allow buying tokens for the pool's address.

8.2. Medium Findings

[M-01] Incorrect calculation in maxBuyWithoutPenalty

Severity

Impact: Low

Likelihood: High

Description

The `maxBuyWithoutPenalty` function calculates the amount of tokens that can be purchased without penalty. But the formula for calculating `expectedBalance` is incorrect. It should use `SNIPE_PROTECTED_SUPPLY` instead of `TOTAL_SUPPLY`.

```
function maxBuyWithoutPenalty() external view returns (uint256) {
    ...

    uint256 elapsedSeconds = block.timestamp - GENESIS_TIME;
@>  uint256 expectedBalance = TOTAL_SUPPLY -
  (TOTAL_SUPPLY * elapsedSeconds) / SNIPE_PROTECTION_SECONDS;

    ...
}
```

Recommendations

Use `SNIPE_PROTECTED_SUPPLY` instead of `TOTAL_SUPPLY`.

```
function maxBuyWithoutPenalty() external view returns (uint256) {
    ...

    uint256 elapsedSeconds = block.timestamp - GENESIS_TIME;
-   uint256 expectedBalance = TOTAL_SUPPLY -
- (TOTAL_SUPPLY * elapsedSeconds) / SNIPE_PROTECTION_SECONDS;
+   uint256 expectedBalance = TOTAL_SUPPLY -
+ (SNIPE_PROTECTED_SUPPLY * elapsedSeconds) / SNIPE_PROTECTION_SECONDS;

    ...
}
```

[M-02] Code should set `ethB` value in `_curveBuy()` when `curveLiquidityMet` is true

Severity

Impact: Low

Likelihood: High

Description

in `_curveBuy()` when `curveLiquidityMet` is true, code buys all the tokens from pool B but it sets value of the of `ethA` instead of `ethB`:

```
function _curveBuy(uint112 buyValue, bool curveLiquidityMet)
        internal
        returns (
          uint112ethA,
          uint112ethB,
          uint112classA,
          uint112classB,
          boolupdatedCurveLiquidityMet
        )
    {
        updatedCurveLiquidityMet = curveLiquidityMet;
        uint112 remainingBuyValue;

        if (updatedCurveLiquidityMet) {
            remainingBuyValue = buyValue;
            ethA = buyValue;
        } else {
```

As `ethB` shows how much ETH is spent for pool B the value with be wrong and those functions that rely on this function's return value won't work properly. The same issue exists in `_curveBuyCalculation()` too.

Recommendations

Set `ethB` value when `curveLiquidityMet` is true.

[M-03] Potential misuse of `forceSafeTransferETH`

Severity

Impact: High

Likelihood: Low

Description

The `g8keepBondingCurve` contract utilizes the `SafeTransferLib.forceSafeTransferETH` function at line 840 to transfer Ether safely, as part of a reward mechanism. This function operates by attempting to send the value to the address, and if that fails, it creates a temporary contract holding the balance and uses self-destruction to forcibly send the ETH to the address, this is done to ensure the migration does not fail under any circumstances.

```
File: g8keepBondingCurve.sol
840:          SafeTransferLib.forceSafeTransferETH
  (DEPLOYER, DEPLOYER_REWARD, 50_000);
```

However, there is a potential issue regarding the determination of the `DEPLOYER` address, which is set as `msg.sender` when the `deployToken` function is called in the `g8keepBondingCurveFactory` contract.

```
File: g8keepBondingCurveFactory.sol
187:                        DEPLOYER: msg.sender,
```

The problem arises when the `deployToken` function is invoked by a `Multicaller` or any intermediate contract. In such cases, the intermediary contract becomes `msg.sender` and consequently the designated `DEPLOYER`, the ETH would be forcibly sent into the contract's balance. This results in the intermediary contract receiving the Ether transfer, rather than the intended user. This can lead to a race condition where malicious actors, such as MEV bots, exploit this situation to capture the Ether.

Additionally, when the token is migrated, the Uniswap position is sent to the **DEPLOYER**, which could potentially result in the loss of this position.

```
File: g8keepBondingCurve.sol
837:          UNISWAP_POSITION_MANAGER.approve(address
  (LOCKER_FACTORY), lpTokenId);
838:          LOCKER_FACTORY.deploy(lpTokenId, DEPLOYER);
```

Recommendations

Instead of using `msg.sender` as the **DEPLOYER** when calling `g8keepBondingCurveFactory::deployToken`, consider adding a parameter to specify the deployer reward recipient, ensuring it is eligible to receive the tokens.

[M-04] Liquidity shortfall during failed migration

Severity

Impact: High

Likelihood: Low

Description

When `reserves0` or the native token reaches the value of `MIGRATION_MINIMUM_LIQUIDITY` and the volume is met, it is possible to execute `_migrateToken`:

```
File: g8keepBondingCurve.sol
        if (curveLiquidityMet && curveVolumeMet && !migrationFailed) {
            _migrateToken(0);
        }
```

The problem is that if the migration fails, buying and selling remain available. If the `_curveSell` function is called with `curveLiquidityMet` set to true, it could utilize the `bReserve` for processing the sale. This situation might lead to a scenario where the sale occurs under the `bReserves`, potentially impacting the liquidity balance.

```
File: g8keepBondingCurve.sol
708:        if (remainingToSell > 0) {
709:            Reserves storage sellFromReserves = aReserve;
710:@>>         if (curveLiquidityMet) {
711:@>>             sellFromReserves = bReserve;
712:@>>         }
713:            uint112 reserve0 = sellFromReserves.reserve0;
714:            uint112 reserve1 = sellFromReserves.reserve1;
715:
716:            uint112 amountToSell = remainingToSell;
717:            if (classA < amountToSell) {
718:                revert InsufficientBalance();
719:            } else {
720:                classA -= amountToSell;
721:            }
722:
723:            uint112 reserve0Out = _getAmountOut
  (amountToSell, reserve1, reserve0);
724:            amountOut += reserve0Out;
725:@>>         sellFromReserves.reserve0 -= reserve0Out;
726:            sellFromReserves.reserve1 += amountToSell;
727:        }
```

Consider the following scenario:

1. The curve is configured and ready for migration, but an error or condition leads to a failed migration.
2. Users, reacting to the failed migration, might sell tokens, which reduces the native token/ETH liquidity below the required `MIGRATION_MINIMUM_LIQUIDITY`.
3. Subsequently, the `executeMigration` function is called again, and this time the migration succeeds. However, due to the reduced liquidity from the user sales, the migration occurs with less liquidity than initially intended, potentially below the `MIGRATION_MINIMUM_LIQUIDITY`.

This could result in a compromised migration state where the expected liquidity guarantees are not met.

Recommendations

Implement additional checks in the `executeMigration` function to ensure that liquidity levels meet or exceed the `MIGRATION_MINIMUM_LIQUIDITY` before proceeding with the migration.

8.3. Low Findings

[L-01] Forcing token deployer to purchase the initial supply

The token deployer may not want to purchase the initial supply. However, during deployment, the `buy` function is always called, forcing them to purchase.

If the deployer sets the token amount (`msg.value`) to zero when calling the `buy` function, the `g8keepBondingCurve._buy` function will revert the transaction, so the deployer must purchase tokens.

```
function deployToken(
    ...
) external payable returns (address _tokenAddress) {
    uint256 bundleBuyAmount = msg.value;
    uint256 _deploymentFee = deploymentFee;
    if (msg.value < _deploymentFee) revert InvalidDeploymentFee();
    if (_deploymentFee > 0) {
        bundleBuyAmount -= _deploymentFee;
        SafeTransferLib.safeTransferETH(g8keepFeeWallet, _deploymentFee);
    }
    if (bundleBuyAmount > maxBundleBuyAmount) {
        revert InvalidBundleBuy();
    }
    ...
@>  g8keepBondingCurve(payable(_tokenAddress)).buy{value: bundleBuyAmount}
  (msg.sender, 0);
    ...
}
```

Additionally, since `elapsedSeconds` is 0, the token deployer gets the maximum penalty, so most initial tokens are purchased as classB.

```
function _applySnipeProtection(uint112 amountOut) internal view returns
  (uint112 adjustedAmountOut) {
    ...

    // Calculate expected balance based on elapsed time
@>  uint112 elapsedSeconds = uint112(block.timestamp - GENESIS_TIME); // 0
@>  uint112 expectedBalance = TOTAL_SUPPLY - SNIPE_PROTECTED_SUPPLY *
// elapsedSeconds / SNIPE_PROTECTION_SECONDS; // TOTAL_SUPPLY

    // Apply snipe penalties if adjusted balance is less than expected
    if (expectedBalance > adjustedBalance) {
        uint256 exponentPenalty = SNIPE_PENALTY_BASE_EXPONENT;
        if (elapsedSeconds < SNIPE_PROTECTION_HEAVY_PENALTY_SECONDS) {
            unchecked {
@>              exponentPenalty += SNIPE_PROTECTION_HEAVY_EXPONENT_START // get
// maximum penalty
@>                  *
  (SNIPE_PROTECTION_HEAVY_PENALTY_SECONDS - elapsedSeconds) / SNIPE_PROTECTION_HEAVY_P
            }
        }
        for (uint256 i; i < exponentPenalty; ++i) {

                         tmpAdjustedAmountOut = tmpAdjustedAmountOut * adjustedBalance
        }
    }
    adjustedAmountOut = uint112(tmpAdjustedAmountOut);
}
```

By calling the `buy` function only when the remaining `bundleBuyAmount` is greater than 0, we can give the option not to buy the initial supply.

[L-02] `withdrawETH` charges fees

The g8keepLiquidityLocker contract has functions to withdraw tokens deposited in the contract. Unlike ERC20, the ETH withdrawal function charges g8keep fees, even after the vesting period. Since the tokens transferred to the contract are not Uniswap V3 fees, charging g8keep fees appears to be an incorrect implementation.

```
function withdrawERC20(address _token) external onlyOwner {
    if (_token == address(UNISWAP_POSITION_MANAGER)) revert();
    SafeTransferLib.safeTransferAll(_token, msg.sender);
}

function withdrawETH(address _token) external onlyOwner {
    if (_token == address(UNISWAP_POSITION_MANAGER)) revert();
    uint256 protocolFee = (address(this).balance * G8KEEP_FEE) / BPS;
    uint256 ownerAmount = address(this).balance - protocolFee;

@>  SafeTransferLib.forceSafeTransferETH
  (G8KEEP_FEE_RECIPIENT, protocolFee, 50_000);
    SafeTransferLib.forceSafeTransferETH(msg.sender, ownerAmount, 50_000);
}
```

The issue can be resolved by removing the g8keep fee in the `withdrawETH` function.

[L-03] Some tokens may be locked in token contract

Code uses `_balances[address(this)].classA` to get the contract's token balance during the migration and withdrawal of excess funds. The issue is that the contract address may have `classB` balance too and those funds won't be used in migration and the deployer can't withdraw them too. It's possible to transfer `classB` balance to the contract address or buy tokens for the contract address which would increase the `classB` balance of the contract address. It would be better to use `balanceOf(This)` to get the contract's balance to make sure there would be no leftovers.

[L-04] STATUS_VOLUME_FLAG may not be set at the correct moment due to off by one error

Each time tokens are bought or sold, `_applyLiquiditySupplement` is called, updating `liquiditySupplement` and `updatedCurveVolumeMet`. If `liquiditySupplement` is reached its maximum value due to this transaction, the `STATUS_VOLUME_FLAG` flag is set.

When comparing them, it uses `<` instead of `<=`. Therefore, the flag is not set when `maxSupplementFee == liquiditySupplementFee`. This may cause the migration to be delayed once.

The same issue exists with the _applyLiquiditySupplementCalculation function.

```
function _applyLiquiditySupplement(uint112 value, bool curveVolumeMet)
    internal
    returns (uint112 remainingValue, bool updatedCurveVolumeMet)
{
    ...

    if (!updatedCurveVolumeMet) {
        unchecked {
            uint112 _liquiditySupplement = liquiditySupplement;

                         uint112 maxSupplementFee = MIGRATION_MINIMUM_LIQUIDITY_SUPPLE

                         uint112 liquiditySupplementFee = value * LIQUIDITY_SUPPLEMENT
@>          if (maxSupplementFee < liquiditySupplementFee) {
                liquiditySupplementFee = maxSupplementFee;
                curveStatus = curveStatus | STATUS_VOLUME_FLAG;
                updatedCurveVolumeMet = true;
            }
            ...
        }
    }
}
```

Use `<=` instead of `<` when comparing `maxSupplementFee` and `liquiditySupplementFee`.

[L-05] `getCurveStatus` returns incorrect values

The function redeclares variables with the same names to receive the results of the `_getCurveStatus` function. Since the named return variables of the `getCurveStatus` function are not updated, it cannot return the correct status.

```
function getCurveStatus() external view returns (
    Reserves memory _aReserve,
    Reserves memory _bReserve,
@>  bool _curveLiquidityMet,
@>  bool _curveVolumeMet,
@>  bool _curveMigrated,
@>  bool _migrationFailed
) {
    _aReserve = aReserve;
    _bReserve = _bReserve;

    (
@>      bool _curveLiquidityMet,
@>      bool _curveVolumeMet,
@>      bool _curveMigrated,
@>      bool _migrationFailed
    ) = _getCurveStatus();
}
```

Set values directly to named return variables.

```
function getCurveStatus() external view returns (
    Reserves memory _aReserve,
    Reserves memory _bReserve,
    bool _curveLiquidityMet,
    bool _curveVolumeMet,
    bool _curveMigrated,
    bool _migrationFailed
) {
    _aReserve = aReserve;
    _bReserve = _bReserve;

    (
-       bool _curveLiquidityMet,
-       bool _curveVolumeMet,
-       bool _curveMigrated,
-       bool _migrationFailed
+       _curveLiquidityMet,
+       _curveVolumeMet,
+       _curveMigrated,
+       _migrationFailed
    ) = _getCurveStatus();
}
```

[L-06] Function `_getAmountIn()` should round up when calculating the input token amount

Code use function `_getAmountIn()` to calculate swap results during the buy transactions:

```
if (amount1Out > classA) {
                uint112 classACost = _getAmountIn(classA, reserve0, reserve1);
                remainingBuyValue = remainingBuyValue + reserveAIn - classACost;
                reserveAIn = classACost;
                ethA = classACost;
            }
```

The issue is that `_getAmountIn()` rounds down in favor of user when calculating the input tokens amounts and as result users pays less tokens. This can open new attacker vectors that attacker could inflates the price and then use this rounding direction issue to extract value.

In `_getAmountIn()` rounds up in favor of the pool.

[L-07] Attacker can DOS token creation by front-running

When users want to create a new token they call `deploy()` function of the factory contract. The issue is that the attacker can DOS the token creation by front-running.

```
IUniswapV3Factory factory = IUniswapV3Factory
          (UNISWAP_POSITION_MANAGER.factory());
        if (factory.getPool(address(this), address
          (WETH), UNISWAP_FEE_TIER) != address(0)) {
            revert PoolAlreadyCreated();
        }
```

The attacker can front-run and create a Uniswap V3 pool for the token address. As a result attacker can DOS other users' deployment transactions.

There's no easy fix for this. It's better to ask users to use private mempools.

[L-08] Compromised `G8KEEP_FEE_WALLET` can disrupt buy/sell operations

The `g8keepBondingCurve::_applyFee` function is responsible for sending fees to the `G8KEEP_FEE_WALLET`. This function is invoked during buy and sell operations.

```
File: g8keepBondingCurve.sol
         function _applyFee(uint112 value) internal returns
  (uint112 remainingValue) {
             unchecked {
                 uint112 fee = value * G8KEEP_FEE / FEE_DENOMINATOR;
@>>              SafeTransferLib.safeTransferETH(G8KEEP_FEE_WALLET, fee);
                 remainingValue = value - fee;
             }
         }
```

If the `G8KEEP_FEE_WALLET` is compromised, an attacker can exploit this by executing a high gas consumption operation that causes the `safeTransferETH` call to revert. This can result in the disruption of buy and sell operations, effectively allowing the attacker to manipulate and control trading activities.

```
File: g8keepBondingCurve.sol
             uint112 sellValue = _applyFee(_curveSell
  (msg.sender, amount, curveLiquidityMet));
...
             uint112 buyValue = _applyFee(uint112(msg.value));
```

Use `forceSafeTransferETH` to ensure the pool can continue operating even if the `G8KEEP_FEE_WALLET` is compromised.

```
function _applyFee(uint112 value) internal returns
      (uint112 remainingValue) {
        unchecked {
            uint112 fee = value * G8KEEP_FEE / FEE_DENOMINATOR;
-           SafeTransferLib.safeTransferETH(G8KEEP_FEE_WALLET, fee);
+           SafeTransferLib.forceSafeTransferETH(G8KEEP_FEE_WALLET, fee);
            remainingValue = value - fee;
        }
    }
```

[L-09] Token deployer can avoid g8keep fees

In the g8keepLiquidityLocker contract, calling `collectFees` allows collecting fees earned from UniV3. n% of the UniV3 fee must be given as g8keep fees. This function can only be called by the administrator(token deployer).

```
@>  function collectFees
  (address _recipient, uint256 _tokenId) external onlyOwner {
```

However, the `release` function, which returns the UniV3 NFT to the administrator(token deployer) after the vesting period, does not settle the fees. Therefore, token deployers can avoid paying g8keep fees by waiting until the end of the vesting period without calling `collectFees`, and then collect the fees after receiving the NFT back.

```
function release(uint256 tokenId) external onlyOwner {
    if (block.timestamp < VESTING_END) {
        revert();
    }

    UNISWAP_POSITION_MANAGER.transferFrom(address(this), msg.sender, tokenId);
    emit LiquidityReleased(tokenId);
}
```

Call `collectFees` in the `release` function to collect and distribute fees before returning the NFT.