2021-11-unlock (Unlock Protocol)

Benchmark Security Review Performed by Bug Hunter

27.06.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project 2021-11-unlock was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/code-423n4/2021-11-unlock
  Commit hash: `ec41eada1dd116bcccc5603ce342257584bec783`

Bug Hunter reviewed the following files:

Unlock.sol MixinTransfer.sol MixinPurchase.sol MixinLockCore.sol MixinRefunds.sol PublicLock.sol MixinGrantKeys.sol MixinKeys.sol MixinERC721Enumerable.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	2
Medium	8
Low	16

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
27.06.2025	Project submitted
27.06.2025	Security review concluded

Detailed Findings

1. MEV Manipulation of UDT Minting via tx.gasprice and Oracle Inputs

Severity	High
Finding ID	BH-2021-11-unlock-01
Target	smart-contracts/contracts/Unlock.sol
Function name	recordKeyPurchase

Description

The function `recordKeyPurchase` allows calculation of `tokensToDistribute` based on `tx.gasprice` and a dynamic oracle-fed `udtPrice`. In Solidity 0.8.2 and above, while overflows are checked, economic manipulation is still possible. A malicious MEV miner or bot can submit a transaction with artificially high gas price or manipulate oracle inputs to inflate `tokensToDistribute`. On mainnet (chainId == 1), tokens are minted without a hard upper limit, which may result in excessive UDT minting beyond expected supply growth. This behavior undermines the economic model of the protocol and could lead to dilution of UDT's value.

```
function recordKeyPurchase(
    uint _value,
    address _referrer
  )
    public
    onlyFromDeployedLock()
  {
    if(_value > 0) {
      uint valueInETH;
      address tokenAddress = IPublicLock(msg.sender).tokenAddress();
      if(tokenAddress != address(0) && tokenAddress != weth) {
        IUniswapOracle oracle = uniswapOracles[tokenAddress];
        if(address(oracle) != address(0)) {
          valueInETH = oracle.updateAndConsult(tokenAddress, _value, weth);
        }
      }
      else {
        valueInETH = _value;
      }
      grossNetworkProduct = grossNetworkProduct + valueInETH;
      locks[msg.sender].totalSales += valueInETH;

      if(_referrer != address(0))
      {
        IUniswapOracle udtOracle = uniswapOracles[udt];
        if(address(udtOracle) != address(0))
        {
          uint udtPrice = udtOracle.updateAndConsult(udt, 10 ** 18, weth);

          uint tokensToDistribute = (estimatedGasForPurchase * tx.gasprice) * (125
* 10 ** 18) / 100 / udtPrice;

          uint maxTokens = 0;
          if (chainId > 1)
          {
            maxTokens = IMintableERC20(udt).balanceOf(address(this)) * valueInETH /
(2 + 2 * valueInETH / grossNetworkProduct) / grossNetworkProduct;
          } else {
            maxTokens = IMintableERC20(udt).totalSupply() * valueInETH / 2 /
grossNetworkProduct;
          }

          if(tokensToDistribute > maxTokens)
          {
            tokensToDistribute = maxTokens;
          }

          if(tokensToDistribute > 0)
          {
            uint devReward = tokensToDistribute * 20 / 100;
            if (chainId > 1)
            {
              uint balance = IMintableERC20(udt).balanceOf(address(this));
              if (balance > tokensToDistribute) {
                IMintableERC20(udt).transfer(_referrer, tokensToDistribute -
devReward);
                IMintableERC20(udt).transfer(owner(), devReward);
              }
            } else {
              IMintableERC20(udt).mint(_referrer, tokensToDistribute - devReward);
              IMintableERC20(udt).mint(owner(), devReward);
            }
          }
```

```
            }
        }
    }
}
```

Recommendation

Implement safeguards against MEV-based manipulation by capping or normalizing `tx.gasprice` used in reward calculations (e.g., using block.basefee or a max threshold). Use a time-weighted average price (TWAP) or secure oracle (e.g., Chainlink) to prevent oracle manipulation. Impose a maximum token minting cap per transaction or block to limit the damage of unexpected inflation.

2. Logical Flaw in Key Transfer May Cause Key Ownership Conflicts and Time Manipulation

Severity	High
Finding ID	BH-2021-11-unlock-02
Target	smart-contracts/contracts/mixins/MixinTransfer.sol
Function name	transferFrom

Description

The `transferFrom` function in `MixinTransfer.sol` contains multiple logic flaws that can be exploited or lead to unintended key ownership states. Most critically, if `_recipient` already has an active key, the function attempts to extend their key's expiration by adding the remaining time from the sender's key. However, it does not validate if the combined expiration exceeds reasonable or capped values, potentially resulting in excessively long-lived keys. Additionally, tokenId reassignment and overwriting without burning or emitting corresponding events can cause unexpected behavior in UIs or integrations. The function also sets `fromKey.tokenId = 0` but does not emit an event to reflect this burning/reset, which may lead to data inconsistency.

```
function transferFrom(
    address _from,
    address _recipient,
    uint _tokenId
  )
    public
    onlyIfAlive
    hasValidKey(_from)
    onlyKeyManagerOrApproved(_tokenId)
  {
    require(ownerOf(_tokenId) == _from, 'TRANSFER_FROM: NOT_KEY_OWNER');
    require(transferFeeBasisPoints < BASIS_POINTS_DEN, 'KEY_TRANSFERS_DISABLED');
    require(_recipient != address(0), 'INVALID_ADDRESS');
    uint fee = getTransferFee(_from, 0);

    Key storage fromKey = keyByOwner[_from];
    Key storage toKey = keyByOwner[_recipient];

    uint previousExpiration = toKey.expirationTimestamp;
    _timeMachine(_tokenId, fee, false);

    if (toKey.tokenId == 0) {
      toKey.tokenId = _tokenId;
      _recordOwner(_recipient, _tokenId);
      _clearApproval(_tokenId);
    }

    if (previousExpiration <= block.timestamp) {
      toKey.expirationTimestamp = fromKey.expirationTimestamp;
      toKey.tokenId = _tokenId;
      _setKeyManagerOf(_tokenId, address(0));
      _recordOwner(_recipient, _tokenId);
    } else {
      toKey.expirationTimestamp = fromKey.expirationTimestamp + previousExpiration
- block.timestamp;
    }

    fromKey.expirationTimestamp = block.timestamp;
    fromKey.tokenId = 0;

    emit Transfer(
      _from,
      _recipient,
      _tokenId
    );
  }
```

Recommendation

Add strict checks on maximum expiration time that can be assigned to a key. Emit appropriate events when a key is effectively burned or reassigned. Ensure unique token IDs are not assigned to multiple owners at any point. Consider using ERC721 semantics more strictly, or document deviations clearly. Also validate that `toKey.expirationTimestamp` does not overflow in extreme edge cases.

3. Free UDT Arbitrage via Oracle Manipulation and Underpriced UDT

Severity	Medium
Finding ID	BH-2021-11-unlock-03
Target	smart-contracts/contracts/Unlock.sol
Function name	recordKeyPurchase

Description

The `recordKeyPurchase` function in `Unlock.sol` calculates the number of UDT tokens to mint and distribute based on the UDT/ETH price obtained from an on-chain oracle. If the oracle returns an artificially low or zero `udtPrice`, either due to manipulation or a broken data feed, the calculation `(estimatedGasForPurchase * tx.gasprice) * (125 * 10 ** 18) / 100 / udtPrice` results in a disproportionately large `tokensToDistribute` value. On mainnet (`chainId == 1`), this leads to excessive minting of UDT tokens without supply cap enforcement. An attacker can manipulate the UDT price on the oracle (e.g., Uniswap with low liquidity), trigger the `recordKeyPurchase` function, and receive a large amount of freshly minted UDT for little or no cost, enabling arbitrage by selling the UDT at market value.

```
function recordKeyPurchase(
    uint _value,
    address _referrer
  )
    public
    onlyFromDeployedLock()
  {
    if(_value > 0) {
      uint valueInETH;
      address tokenAddress = IPublicLock(msg.sender).tokenAddress();
      if(tokenAddress != address(0) && tokenAddress != weth) {
        IUniswapOracle oracle = uniswapOracles[tokenAddress];
        if(address(oracle) != address(0)) {
          valueInETH = oracle.updateAndConsult(tokenAddress, _value, weth);
        }
      }
      else {
        valueInETH = _value;
      }
      grossNetworkProduct = grossNetworkProduct + valueInETH;
      locks[msg.sender].totalSales += valueInETH;

      if(_referrer != address(0))
      {
        IUniswapOracle udtOracle = uniswapOracles[udt];
        if(address(udtOracle) != address(0))
        {
          uint udtPrice = udtOracle.updateAndConsult(udt, 10 ** 18, weth);

          uint tokensToDistribute = (estimatedGasForPurchase * tx.gasprice) * (125
* 10 ** 18) / 100 / udtPrice;

          uint maxTokens = 0;
          if (chainId > 1)
          {
            maxTokens = IMintableERC20(udt).balanceOf(address(this)) * valueInETH /
(2 + 2 * valueInETH / grossNetworkProduct) / grossNetworkProduct;
          } else {
            maxTokens = IMintableERC20(udt).totalSupply() * valueInETH / 2 /
grossNetworkProduct;
          }

          if(tokensToDistribute > maxTokens)
          {
            tokensToDistribute = maxTokens;
          }

          if(tokensToDistribute > 0)
          {
            uint devReward = tokensToDistribute * 20 / 100;
            if (chainId > 1)
            {
              uint balance = IMintableERC20(udt).balanceOf(address(this));
              if (balance > tokensToDistribute) {
                IMintableERC20(udt).transfer(_referrer, tokensToDistribute -
devReward);
                IMintableERC20(udt).transfer(owner(), devReward);
              }
            } else {
              IMintableERC20(udt).mint(_referrer, tokensToDistribute - devReward);
              IMintableERC20(udt).mint(owner(), devReward);
            }
          }
          }
        }
      }
    }
```

Recommendation

Implement a minimum valid threshold for `udtPrice` and revert if the price is 0 or unreasonably low. Use a secure, manipulation-resistant oracle such as Chainlink or a TWAP-based oracle from Uniswap v3. Add a hard cap on `tokensToDistribute` per transaction or per block. Replace `tx.gasprice` with `block.basefee` to limit MEV exploitation. Perform sanity checks on reward size relative to purchase value.

4. Missing Validation in ERC20 Token Purchase Path Enables Economic Abuse and Incorrect Key Pricing

Severity	Medium
Finding ID	BH-2021-11-unlock-04
Target	smart-contracts/contracts/mixins/MixinPurchase.sol
Function name	purchase

Description

In the `purchase` function of `MixinPurchase.sol`, when a purchase is made using an ERC20 token (i.e., `tokenAddress != address(0)`), the contract records the key price and mints the key before transferring the tokens via `token.transferFrom`. This opens multiple vulnerabilities: (1) If the `transferFrom` fails silently (via non-compliant ERC20 tokens), the function does not revert until the final `require(pricePaid >= inMemoryKeyPrice)` check. If the token does not correctly return a `bool`, as in some non-standard ERC20s, the contract may proceed as if the transfer was successful. (2) The value passed to `recordKeyPurchase` (`inMemoryKeyPrice`) may not match `pricePaid`, which is only verified after rewards are granted. (3) An attacker may exploit tokens with rebasing or fee-on-transfer mechanics to reduce their actual payment but still trigger full protocol reward logic. This can be used to game the UDT reward mechanism indirectly.

```
function purchase(
    uint256 _value,
    address _recipient,
    address _referrer,
    bytes calldata _data
  ) external payable
    onlyIfAlive
    notSoldOut
  {
    require(_recipient != address(0), 'INVALID_ADDRESS');

    Key storage toKey = keyByOwner[_recipient];
    uint idTo = toKey.tokenId;
    uint newTimeStamp;

    if (idTo == 0) {
      _assignNewTokenId(toKey);
      idTo = toKey.tokenId;
      _recordOwner(_recipient, idTo);
      newTimeStamp = block.timestamp + expirationDuration;
      toKey.expirationTimestamp = newTimeStamp;

      emit Transfer(
        address(0),
        _recipient,
        idTo
      );
    } else if (toKey.expirationTimestamp > block.timestamp) {
      newTimeStamp = toKey.expirationTimestamp + expirationDuration;
      toKey.expirationTimestamp = newTimeStamp;
      emit RenewKeyPurchase(_recipient, newTimeStamp);
    } else {
      newTimeStamp = block.timestamp + expirationDuration;
      toKey.expirationTimestamp = newTimeStamp;
      _setKeyManagerOf(idTo, address(0));
      emit RenewKeyPurchase(_recipient, newTimeStamp);
    }

    (uint inMemoryKeyPrice, uint discount, uint tokens) =
_purchasePriceFor(_recipient, _referrer, _data);
    if (discount > 0)
    {
      unlockProtocol.recordConsumedDiscount(discount, tokens);
    }

    unlockProtocol.recordKeyPurchase(inMemoryKeyPrice, _referrer);

    uint pricePaid;
    if(tokenAddress != address(0))
    {
      pricePaid = _value;
      IERC20Upgradeable token = IERC20Upgradeable(tokenAddress);
      token.transferFrom(msg.sender, address(this), pricePaid);
    }
    else
    {
      pricePaid = msg.value;
    }
    require(pricePaid >= inMemoryKeyPrice, 'INSUFFICIENT_VALUE');

    if(address(onKeyPurchaseHook) != address(0))
    {
      onKeyPurchaseHook.onKeyPurchase(msg.sender, _recipient, _referrer, _data,
inMemoryKeyPrice, pricePaid);
    }
  }
```

Recommendation

Ensure the `transferFrom` operation is safe and checks for both return success and actual token balance before and after transfer. Consider using OpenZeppelin's `SafeERC20` library. Validate that the actual `pricePaid` matches `inMemoryKeyPrice` before calling `unlockProtocol.recordKeyPurchase`, especially for ERC20 flows. Reject ERC20s with transfer hooks, rebase effects, or unusual mechanics unless explicitly supported.

5. Key Refunds Become Impossible If Lock Manager Withdraws All Funds

Severity	Medium
Finding ID	BH-2021-11-unlock-05
Target	smart-contracts/contracts/mixins/MixinLockCore.sol
Function name	withdraw

Description

The `withdraw` function in `MixinLockCore.sol` allows the lock manager or beneficiary to withdraw the entire contract balance (ETH or ERC20 tokens) without tracking or reserving any amount for potential key refunds. Since there is no internal accounting or segregation between refundable user funds and protocol profit, this enables a scenario where a lock manager can drain the contract balance entirely. As a result, key buyers who are eligible for refunds (e.g., within the refund window) may be unable to receive their funds, leading to denial-of-service and loss of trust in the protocol.

```
function withdraw(
    address _tokenAddress,
    uint _amount
  ) external
    onlyLockManagerOrBeneficiary
  {
    uint balance;
    if(_tokenAddress == address(0)) {
      balance = address(this).balance;
    } else {
      balance = IERC20Upgradeable(_tokenAddress).balanceOf(address(this));
    }

    uint amount;
    if(_amount == 0 || _amount > balance)
    {
      require(balance > 0, 'NOT_ENOUGH_FUNDS');
      amount = balance;
    }
    else
    {
      amount = _amount;
    }

    emit Withdrawal(msg.sender, _tokenAddress, beneficiary, amount);
    _transfer(_tokenAddress, beneficiary, amount);
  }
```

Recommendation

Implement internal accounting to track refundable balances separately from protocol earnings. Ensure that withdrawals do not reduce the balance below the amount needed to cover potential refunds. Alternatively, enforce a minimum reserve ratio or restrict full balance withdrawals unless all keys are expired or non-refundable. Include logic to block withdrawal if outstanding refundable keys exist.

6. Refund Calculation Ignores Historical Purchase Price

Severity	Medium
Finding ID	BH-2021-11-unlock-06
Target	smart-contracts/contracts/mixins/MixinRefunds.sol
Function name	_getCancelAndRefundValue

Description

The `_getCancelAndRefundValue` function in `MixinRefunds.sol` uses the current global `keyPrice` to calculate the refund amount when a key is canceled. This approach assumes that all users paid the same amount for their keys, but it does not account for dynamic pricing, discounts, promotions, or historical price changes. As a result, users who purchased at a lower price may be over-refunded if `keyPrice` increased, and those who paid more may be under-refunded if `keyPrice` decreased. This leads to incorrect economic accounting and potential financial loss to the protocol or unfair refunds to users.

```
function _getCancelAndRefundValue(
    address _keyOwner
  )
    private view
    hasValidKey(_keyOwner)
    returns (uint refund)
  {
    Key storage key = keyByOwner[_keyOwner];
    uint timeRemaining = key.expirationTimestamp - block.timestamp;
    if(timeRemaining + freeTrialLength >= expirationDuration) {
      refund = keyPrice;
    } else {
      refund = keyPrice * timeRemaining / expirationDuration;
    }

    if(freeTrialLength == 0 || timeRemaining + freeTrialLength < expirationDuration)
    {
      uint penalty = keyPrice * refundPenaltyBasisPoints / BASIS_POINTS_DEN;
      if (refund > penalty) {
        refund -= penalty;
      } else {
        refund = 0;
        }
      }
    }
```

Recommendation

Track the actual price paid per key (e.g., store it in the `Key` struct at purchase time) and use that value in the refund calculation instead of the current global `keyPrice`. This ensures refunds are fairly based on what the user actually paid.

7. Key Transfer to Self Results in Key Expiration and Destruction

Severity	Medium
Finding ID	BH-2021-11-unlock-07
Target	smart-contracts/contracts/mixins/MixinTransfer.sol
Function name	transferFrom

Description

The `transferFrom` function in `MixinTransfer.sol` does not validate whether the `_from` and `_recipient` addresses are the same. If a key owner mistakenly or maliciously calls `transferFrom` with the same address for both `_from` and `_recipient`, the logic will proceed to expire the key (`fromKey.expirationTimestamp = block.timestamp`) and reset the `fromKey.tokenId` to 0. Since no net ownership change occurs, the result is the destruction of the key, despite no actual transfer taking place. This leads to unexpected loss of access for the key owner, breaking user expectations and posing a denial-of-service risk.

```
function transferFrom(
    address _from,
    address _recipient,
    uint _tokenId
  )
    public
    onlyIfAlive
    hasValidKey(_from)
    onlyKeyManagerOrApproved(_tokenId)
  {
    require(ownerOf(_tokenId) == _from, 'TRANSFER_FROM: NOT_KEY_OWNER');
    require(transferFeeBasisPoints < BASIS_POINTS_DEN, 'KEY_TRANSFERS_DISABLED');
    require(_recipient != address(0), 'INVALID_ADDRESS');
    uint fee = getTransferFee(_from, 0);

    Key storage fromKey = keyByOwner[_from];
    Key storage toKey = keyByOwner[_recipient];

    uint previousExpiration = toKey.expirationTimestamp;
    _timeMachine(_tokenId, fee, false);

    if (toKey.tokenId == 0) {
      toKey.tokenId = _tokenId;
      _recordOwner(_recipient, _tokenId);
      _clearApproval(_tokenId);
    }

    if (previousExpiration <= block.timestamp) {
      toKey.expirationTimestamp = fromKey.expirationTimestamp;
      toKey.tokenId = _tokenId;
      _setKeyManagerOf(_tokenId, address(0));
      _recordOwner(_recipient, _tokenId);
    } else {
      toKey.expirationTimestamp = fromKey.expirationTimestamp + previousExpiration
- block.timestamp;
    }

    fromKey.expirationTimestamp = block.timestamp;
    fromKey.tokenId = 0;

    emit Transfer(
      _from,
      _recipient,
      _tokenId
    );
  }
```

Recommendation

Add a `require(_from != _recipient, 'TRANSFER_TO_SELF_FORBIDDEN')` check at the beginning of the function to prevent self-transfers. This avoids unintended key expiration and data corruption caused by redundant logic execution.

8. No cap on number of keys that can be created via sharing

Severity	Medium
Finding ID	BH-2021-11-unlock-08
Target	smart-contracts/contracts/mixins/MixinTransfer.sol
Function name	shareKey

Description

The `shareKey` function in `MixinTransfer.sol` enables a key owner to share part of their key time with another address, creating a new key via `_assignNewTokenId`. However, the function does not enforce a cap on the number of times a key can be shared or validate that shared time has meaningful value. As a result, a malicious user can repeatedly share tiny or even negligible portions of key time to multiple addresses, effectively creating new keys without invoking `purchase()`. This bypasses the pricing logic, reduces protocol revenue, and undermines the economic model by flooding the system with 'free' keys.

```
function shareKey(
    address _to,
    uint _tokenId,
    uint _timeShared
  ) public
    onlyIfAlive
    onlyKeyManagerOrApproved(_tokenId)
  {
    require(transferFeeBasisPoints < BASIS_POINTS_DEN, 'KEY_TRANSFERS_DISABLED');
    require(_to != address(0), 'INVALID_ADDRESS');
    address keyOwner = _ownerOf[_tokenId];
    require(getHasValidKey(keyOwner), 'KEY_NOT_VALID');
    Key storage fromKey = keyByOwner[keyOwner];
    Key storage toKey = keyByOwner[_to];
    uint idTo = toKey.tokenId;
    uint time;
    uint timeRemaining = fromKey.expirationTimestamp - block.timestamp;
    uint fee = getTransferFee(keyOwner, _timeShared);
    uint timePlusFee = _timeShared + fee;

    if(timePlusFee < timeRemaining) {
      time = _timeShared;
      _timeMachine(_tokenId, timePlusFee, false);
    } else {
      fee = getTransferFee(keyOwner, timeRemaining);
      time = timeRemaining - fee;
      fromKey.expirationTimestamp = block.timestamp;
      emit ExpireKey(_tokenId);
    }

    if (idTo == 0) {
      _assignNewTokenId(toKey);
      idTo = toKey.tokenId;
      _recordOwner(_to, idTo);
      emit Transfer(
        address(0),
        _to,
        idTo
      );
    } else if (toKey.expirationTimestamp <= block.timestamp) {
      _setKeyManagerOf(idTo, address(0));
    }

    _timeMachine(idTo, time, true);
    emit Transfer(
      keyOwner,
      _to,
      idTo
    );

    require(_checkOnERC721Received(keyOwner, _to, _tokenId, ''),
'NON_COMPLIANT_ERC721_RECEIVER');
  }
```

Recommendation

Introduce a minimum `_timeShared` threshold and enforce a cap on the number of keys that can be generated from a single parent key. Consider charging a flat or variable fee per share to deter abuse. Track total time shared and enforce limits to prevent recursive or excessive key creation.

9. Missing Validation of msg.sender in initialize Allows Unauthorized Lock Initialization

Severity	Medium
Finding ID	BH-2021-11-unlock-09
Target	smart-contracts/contracts/PublicLock.sol
Function name	initialize

Description

The `initialize` function in `PublicLock.sol` does not validate that `msg.sender` is an expected contract (e.g., a factory or deployer). As a result, any user can directly call `initialize()` on an uninitialized lock clone if the `initializer()` modifier is bypassed (intentionally or via a proxy flaw). This could lead to malicious or accidental initialization with arbitrary parameters, such as a malicious `_lockCreator`, zero key price, or invalid token address. It undermines assumptions about lock ownership and trust during deployment.

```
function initialize(
    address payable _lockCreator,
    uint _expirationDuration,
    address _tokenAddress,
    uint _keyPrice,
    uint _maxNumberOfKeys,
    string calldata _lockName
  ) public
    initializer()
  {
    MixinFunds._initializeMixinFunds(_tokenAddress);
    MixinDisable._initializeMixinDisable();
    MixinLockCore._initializeMixinLockCore(_lockCreator, _expirationDuration,
_keyPrice, _maxNumberOfKeys);
    MixinLockMetadata._initializeMixinLockMetadata(_lockName);
    MixinERC721Enumerable._initializeMixinERC721Enumerable();
    MixinRefunds._initializeMixinRefunds();
    MixinRoles._initializeMixinRoles(_lockCreator);
    _registerInterface(0x80ac58cd);
  }
```

Recommendation

Add a validation that `msg.sender` is the expected factory or deployer contract, or emit a strict ownership claim mechanism post-initialization. Alternatively, use constructor-style initialization via factory-only deployment or authenticate initialization parameters cryptographically.

10. Missing maxNumberOfKeys Check in shareKey Allows Key Supply Inflation

Severity	Medium
Finding ID	BH-2021-11-unlock-10
Target	smart-contracts/contracts/mixins/MixinTransfer.sol
Function name	shareKey

Description

The `shareKey` function in `MixinTransfer.sol` allows the creation of new keys by assigning a new tokenId to recipients who do not already have one. However, the function does not enforce a check against `maxNumberOfKeys`, which is meant to limit the total number of keys that can be issued by a lock. This can result in more keys being issued than intended, breaking economic assumptions and potentially leading to lock abuse or denial of service via key inflation.

```
function shareKey(
    address _to,
    uint _tokenId,
    uint _timeShared
  ) public
    onlyIfAlive
    onlyKeyManagerOrApproved(_tokenId)
  {
    require(transferFeeBasisPoints < BASIS_POINTS_DEN, 'KEY_TRANSFERS_DISABLED');
    require(_to != address(0), 'INVALID_ADDRESS');
    address keyOwner = _ownerOf[_tokenId];
    require(getHasValidKey(keyOwner), 'KEY_NOT_VALID');
    Key storage fromKey = keyByOwner[keyOwner];
    Key storage toKey = keyByOwner[_to];
    uint idTo = toKey.tokenId;
    uint time;
    uint timeRemaining = fromKey.expirationTimestamp - block.timestamp;
    uint fee = getTransferFee(keyOwner, _timeShared);
    uint timePlusFee = _timeShared + fee;

    if(timePlusFee < timeRemaining) {
      time = _timeShared;
      _timeMachine(_tokenId, timePlusFee, false);
    } else {
      fee = getTransferFee(keyOwner, timeRemaining);
      time = timeRemaining - fee;
      fromKey.expirationTimestamp = block.timestamp;
      emit ExpireKey(_tokenId);
    }

    if (idTo == 0) {
      _assignNewTokenId(toKey);
      idTo = toKey.tokenId;
      _recordOwner(_to, idTo);
      emit Transfer(
        address(0),
        _to,
        idTo
      );
    } else if (toKey.expirationTimestamp <= block.timestamp) {
      _setKeyManagerOf(idTo, address(0));
    }

    _timeMachine(idTo, time, true);
    emit Transfer(
      keyOwner,
      _to,
      idTo
    );

    require(_checkOnERC721Received(keyOwner, _to, _tokenId, ''),
'NON_COMPLIANT_ERC721_RECEIVER');
  }
```

Recommendation

Before assigning a new tokenId in `shareKey`, add a check to ensure the current number of issued keys is less than `maxNumberOfKeys`. Revert if the limit would be exceeded. Also audit the `grantKey` function for the same missing check.

11. Arbitrary Version Gaps in addLockTemplate May Break Upgrade Logic

Severity	Low
Finding ID	BH-2021-11-unlock-11
Target	smart-contracts/contracts/Unlock.sol
Function name	addLockTemplate

Description

The `addLockTemplate` function allows the contract owner to assign arbitrary version numbers to new lock templates without enforcing sequential or incremental ordering. This can create large gaps between versions (e.g., adding version 1000 before 2 or 3), which can lead to inconsistent upgrade logic, missed upgrades, or difficulty in iterating over valid versions. The `publicLockLatestVersion` is only updated if the new version is greater than the current one, so gaps in `_publicLockImpls` can be left untracked.

```
function addLockTemplate(address impl, uint16 version) public onlyOwner {
    _publicLockVersions[impl] = version;
    _publicLockImpls[version] = impl;
    if (publicLockLatestVersion < version) publicLockLatestVersion = version;

    emit UnlockTemplateAdded(impl, version);
  }
```

Recommendation

Enforce sequential versioning or require that new versions must be exactly one greater than the current `publicLockLatestVersion`. Additionally, validate that a version is not already assigned, and optionally track a list of explicitly registered versions.

12. Insufficient Version Validation in upgradeLock() Can Cause Denial of Service

Severity	Low
Finding ID	BH-2021-11-unlock-12
Target	smart-contracts/contracts/Unlock.sol
Function name	upgradeLock

Description

The `upgradeLock` function in `Unlock.sol` retrieves the implementation address for a given version from `_publicLockImpls` without validating whether the address is non-zero or points to a valid contract. If the specified version has not been registered correctly, this will result in the upgrade call being made with a zero address. Upgrading a proxy to the zero address would brick the lock contract, rendering it non-functional and causing a denial of service for any key holders or managers relying on that lock.

```
function upgradeLock(address payable lockAddress, uint16 version) public
returns(address) {
    require(proxyAdminAddress != address(0), "proxyAdmin is not set");

    require(_isLockManager(lockAddress, msg.sender) == true, "caller is not a
manager of this lock");

    IPublicLock lock = IPublicLock(lockAddress);
    uint16 currentVersion = lock.publicLockVersion();
    require( version == currentVersion + 1, 'version error: only +1 increments are
allowed');

    address impl = _publicLockImpls[version];
    TransparentUpgradeableProxy proxy = TransparentUpgradeableProxy(lockAddress);
    proxyAdmin.upgrade(proxy, impl);

    emit LockUpgraded(lockAddress, version);
    return lockAddress;
  }
```

Recommendation

Add a check to ensure the retrieved implementation address is not the zero address and is a valid contract using `AddressUpgradeable.isContract(impl)`. Additionally, ensure that only explicitly registered and validated versions are upgradeable targets.

13. grantKeys May Revert Unexpectedly Due to Array Length Mismatch

Severity	Low
Finding ID	BH-2021-11-unlock-13
Target	smart-contracts/contracts/mixins/MixinGrantKeys.sol
Function name	grantKeys

Description

The `grantKeys` function assumes that the `_recipients`, `_expirationTimestamps`, and `_keyManagers` arrays are of equal length, but does not validate this assumption. If any of the arrays are shorter than the others, accessing out-of-bounds indexes will cause the function to revert unexpectedly. This results in a denial of service (DoS), where none of the key grants are processed, even if earlier entries were valid. Attackers or misconfigured frontends could exploit this to halt key distribution.

```
function grantKeys(
    address[] calldata _recipients,
    uint[] calldata _expirationTimestamps,
    address[] calldata _keyManagers
  ) external
    onlyKeyGranterOrManager
  {
    for(uint i = 0; i < _recipients.length; i++) {
      address recipient = _recipients[i];
      uint expirationTimestamp = _expirationTimestamps[i];
      address keyManager = _keyManagers[i];

      require(recipient != address(0), 'INVALID_ADDRESS');

      Key storage toKey = keyByOwner[recipient];
      require(expirationTimestamp > toKey.expirationTimestamp, 'ALREADY_OWNS_KEY');

      uint idTo = toKey.tokenId;

      if(idTo == 0) {
        _assignNewTokenId(toKey);
        idTo = toKey.tokenId;
        _recordOwner(recipient, idTo);
      }
      _setKeyManagerOf(idTo, keyManager);
      emit KeyManagerChanged(idTo, keyManager);

      toKey.expirationTimestamp = expirationTimestamp;
      emit Transfer(
        address(0),
        recipient,
        idTo
      );
    }
  }
```

Recommendation

Add a require statement at the start of the function to ensure that all input arrays have the same length, such as: `require(_recipients.length == _expirationTimestamps.length && _recipients.length == _keyManagers.length, 'ARRAY_LENGTH_MISMATCH');`

14. Missing Return Value Check on ERC20 approve()

Severity	Low
Finding ID	BH-2021-11-unlock-14
Target	smart-contracts/contracts/mixins/MixinLockCore.sol
Function name	approveBeneficiary

Description

The `approveBeneficiary` function directly calls `IERC20Upgradeable(tokenAddress).approve(_spender, _amount)` and returns its result without checking whether the call succeeded. Many ERC20 implementations return `false` instead of reverting on failure, which can lead to silent errors if the approval fails but no exception is thrown. Relying on the return value alone without validating it can introduce inconsistencies and unexpected behavior in downstream logic.

```
function approveBeneficiary(
    address _spender,
    uint _amount
  ) public
    onlyLockManagerOrBeneficiary
    returns (bool)
  {
    return IERC20Upgradeable(tokenAddress).approve(_spender, _amount);
  }
```

Recommendation

Explicitly check the return value of the `approve()` call and ensure it is `true`. Consider using OpenZeppelin's `SafeERC20.safeApprove()` utility for safer handling of token approvals.

15. approveBeneficiary Should Use SafeERC20.safeApprove for Compatibility and Safety

Severity	Low
Finding ID	BH-2021-11-unlock-15
Target	smart-contracts/contracts/mixins/MixinLockCore.sol
Function name	approveBeneficiary

Description

The `approveBeneficiary` function uses the raw `approve()` method of the ERC20 interface. However, not all ERC20 token implementations adhere strictly to the standard — some may return `false` on failure instead of reverting, or behave unexpectedly. This can lead to undetected approval failures and potential inconsistencies in token allowances. Using OpenZeppelin's `SafeERC20.safeApprove` ensures proper error handling and compatibility across diverse ERC20 implementations.

```
function approveBeneficiary(
    address _spender,
    uint _amount
  ) public
    onlyLockManagerOrBeneficiary
    returns (bool)
  {
    return IERC20Upgradeable(tokenAddress).approve(_spender, _amount);
  }
```

Recommendation

Import OpenZeppelin's `SafeERC20` library and use `safeApprove()` instead of `approve()` to ensure proper handling of token approvals. Example:

```
using SafeERC20 for IERC20Upgradeable;
...
IERC20Upgradeable(tokenAddress).safeApprove(_spender, _amount);
```

16. getTransferFee() May Return Zero Fee in Edge Cases

Severity	Low
Finding ID	BH-2021-11-unlock-16
Target	smart-contracts/contracts/mixins/MixinTransfer.sol
Function name	getTransferFee

Description

The `getTransferFee()` function can return a fee of zero under certain edge conditions. Specifically, if `_time == 0` and the key has expired or just about to expire, the calculated `timeToTransfer` becomes zero, resulting in zero fee. Additionally, if the key is deemed invalid by `getHasValidKey`, the function returns zero immediately. This opens up the potential for fee bypasses and unintended economic behavior.

```
function getTransferFee(
    address _keyOwner,
    uint _time
  )
    public view
    returns (uint)
  {
    if(! getHasValidKey(_keyOwner)) {
      return 0;
    } else {
      Key storage key = keyByOwner[_keyOwner];
      uint timeToTransfer;
      uint fee;
      if(_time == 0) {
        timeToTransfer = key.expirationTimestamp - block.timestamp;
      } else {
        timeToTransfer = _time;
      }
      fee = timeToTransfer * transferFeeBasisPoints / BASIS_POINTS_DEN;
      return fee;
    }
  }
```

Recommendation

Introduce a minimum fee threshold or explicitly disallow operations where the computed transfer fee is zero. Also, validate that `_time` is non-zero and meaningful before proceeding. Consider tightening `getHasValidKey()` usage to avoid granting zero-fee sharing options.

17. approveBeneficiary is Vulnerable to ERC20 Approval Race Condition

Severity	Low
Finding ID	BH-2021-11-unlock-17
Target	smart-contracts/contracts/mixins/MixinLockCore.sol
Function name	approveBeneficiary

Description

The `approveBeneficiary` function directly uses `approve()` on an ERC20 token without resetting the existing allowance to zero. This is known to introduce a race condition where a spender could front-run the approval change and spend both the old and new allowance unexpectedly. Many ERC20 tokens require the allowance to be set to 0 before updating it to a new value for this reason.

```
function approveBeneficiary(
    address _spender,
    uint _amount
  ) public
    onlyLockManagerOrBeneficiary
    returns (bool)
  {
    return IERC20Upgradeable(tokenAddress).approve(_spender, _amount);
  }
```

Recommendation

Use the standard mitigation pattern of first setting the allowance to 0 before updating it to a new value, or preferably use OpenZeppelin's `SafeERC20.safeApprove()` with appropriate checks. Example:

```
IERC20Upgradeable(tokenAddress).approve(_spender, 0);
IERC20Upgradeable(tokenAddress).approve(_spender, _amount);
```

Or:

```
using SafeERC20 for IERC20Upgradeable;
token.safeApprove(_spender, _amount);
```

18. Missing Reentrancy Guard in _cancelAndRefund

Severity	Low
Finding ID	BH-2021-11-unlock-18
Target	smart-contracts/contracts/mixins/MixinRefunds.sol
Function name	_cancelAndRefund

Description

The `_cancelAndRefund` function performs external calls — including `_transfer` and `onKeyCancelHook.onKeyCancel` — without any reentrancy guard. This exposes the contract to a potential reentrancy vulnerability, where a malicious hook or recipient contract could call back into the contract before state changes are finalized or re-enter other sensitive functions.

```
function _cancelAndRefund(
    address payable _keyOwner,
    uint refund
  ) internal
  {
    Key storage key = keyByOwner[_keyOwner];

    emit CancelKey(key.tokenId, _keyOwner, msg.sender, refund);
    key.expirationTimestamp = block.timestamp;

    if (refund > 0) {
      _transfer(tokenAddress, _keyOwner, refund);
    }

    if(address(onKeyCancelHook) != address(0))
    {
      onKeyCancelHook.onKeyCancel(msg.sender, _keyOwner, refund);
    }
  }
```

Recommendation

Add a `nonReentrant` modifier from OpenZeppelin's ReentrancyGuard to the `_cancelAndRefund` function, or implement a custom reentrancy lock pattern to prevent recursive calls. External calls should occur only after all internal state changes are finalized.

19. Missing Zero Address Validation for Key Manager

Severity	Low
Finding ID	BH-2021-11-unlock-19
Target	smart-contracts/contracts/mixins/MixinKeys.sol
Function name	_setKeyManagerOf

Description

The `_setKeyManagerOf` function allows setting the key manager to the zero address (`address(0)`) without any restriction. While this may be intentional to represent 'no manager', the lack of validation or clear intention may lead to confusion or unexpected behaviors, particularly if downstream logic assumes a valid non-zero key manager address.

```
function _setKeyManagerOf(
    uint _tokenId,
    address _keyManager
  ) internal
  {
    if(keyManagerOf[_tokenId] != _keyManager) {
      keyManagerOf[_tokenId] = _keyManager;
      _clearApproval(_tokenId);
      emit KeyManagerChanged(_tokenId, address(0));
    }
  }
```

Recommendation

Add an explicit check in `setKeyManagerOf` or `_setKeyManagerOf` to prevent assigning `address(0)` as a key manager unless this is an intended behavior. If allowed, document this clearly in the function comment or emit a separate event to indicate a reset.

20. Missing Access Control on initializeProxyAdmin Function

Severity	Low
Finding ID	BH-2021-11-unlock-20
Target	smart-contracts/contracts/Unlock.sol
Function name	initializeProxyAdmin

Description

The `initializeProxyAdmin` function lacks access control, allowing any user to call it and deploy the ProxyAdmin contract. This can lead to a denial of service, where a malicious user frontruns the contract owner and initializes the ProxyAdmin address, preventing legitimate administrative control over proxy upgrades.

```
function initializeProxyAdmin() public {
    require(proxyAdminAddress == address(0), "ProxyAdmin already deployed");
    _deployProxyAdmin();
  }

  /**
  * @dev Deploy the ProxyAdmin contract that will manage lock templates upgrades
  * This deploys an instance of ProxyAdmin used by PublicLock transparent proxies.
  */
  function _deployProxyAdmin() private returns(address) {
    proxyAdmin = new ProxyAdmin();
    proxyAdminAddress = address(proxyAdmin);
    return address(proxyAdmin);
  }
```

Recommendation

Add an `onlyOwner` modifier or equivalent role-based restriction to the `initializeProxyAdmin` function to ensure that only authorized users can call it.

21. Overwriting Previous Implementations in addLockTemplate

Severity	Low
Finding ID	BH-2021-11-unlock-21
Target	smart-contracts/contracts/Unlock.sol
Function name	addLockTemplate

Description

The function addLockTemplate allows overwriting an existing lock template implementation for a given version without any validation or warning. This could lead to inconsistent behavior in lock upgrades, especially if a version expected to point to a specific implementation is later silently changed. Such unexpected changes might introduce security vulnerabilities or break client integrations.

```
function addLockTemplate(address impl, uint16 version) public onlyOwner {
    _publicLockVersions[impl] = version;
    _publicLockImpls[version] = impl;
    if (publicLockLatestVersion < version) publicLockLatestVersion = version;

    emit UnlockTemplateAdded(impl, version);
  }
```

Recommendation

Add a check to ensure that a version is not already associated with an existing implementation, or emit a warning/log when an overwrite occurs. Alternatively, enforce immutability for version-to-implementation mappings once set.

22. Missing Validation on Template Contract in setLockTemplate

Severity	Low
Finding ID	BH-2021-11-unlock-22
Target	smart-contracts/contracts/Unlock.sol
Function name	setLockTemplate

Description

The function `setLockTemplate` does not validate whether the provided `_publicLockAddress` is a valid and compatible implementation of `IPublicLock`. Without checks (e.g., verifying that the address is a contract using `extcodesize` or calling a known selector), a malformed or malicious contract can be set as a template. This could result in downstream failures during lock deployment or unexpected behavior during interactions with the lock template.

```
function setLockTemplate(
    address _publicLockAddress
  ) external
    onlyOwner
  {
    // First claim the template so that no-one else could
    // this will revert if the template was already initialized.
    IPublicLock(_publicLockAddress).initialize(
      address(this), 0, address(0), 0, 0, ''
    );
    IPublicLock(_publicLockAddress).renounceLockManager();

    publicLockAddress = _publicLockAddress;

    emit SetLockTemplate(_publicLockAddress);
  }
```

Recommendation

Add validation to ensure `_publicLockAddress` is a contract and properly implements the expected interface. Consider verifying with `extcodesize` and checking for the presence of critical functions (like `initialize` and `renounceLockManager`).

23. Division by zero risk in UDT reward calculation

Severity	Low
Finding ID	BH-2021-11-unlock-23
Target	smart-contracts/contracts/Unlock.sol
Function name	recordKeyPurchase

Description

The function `recordKeyPurchase()` calculates the number of UDT tokens to distribute by dividing by the UDT price obtained from a Uniswap oracle. If the oracle returns a price of 0, this results in a division by zero, which causes the transaction to revert. This could happen if the oracle is misconfigured, has no liquidity, or fails to return a valid price, leading to denial-of-service in UDT reward distribution.

```
function recordKeyPurchase(
    uint _value,
    address _referrer
  )
    public
    onlyFromDeployedLock()
  {
    if(_value > 0) {
      uint valueInETH;
      address tokenAddress = IPublicLock(msg.sender).tokenAddress();
      if(tokenAddress != address(0) && tokenAddress != weth) {
        IUniswapOracle oracle = uniswapOracles[tokenAddress];
        if(address(oracle) != address(0)) {
          valueInETH = oracle.updateAndConsult(tokenAddress, _value, weth);
        }
      }
      else {
        valueInETH = _value;
      }

      grossNetworkProduct = grossNetworkProduct + valueInETH;
      locks[msg.sender].totalSales += valueInETH;

      if(_referrer != address(0))
      {
        IUniswapOracle udtOracle = uniswapOracles[udt];
        if(address(udtOracle) != address(0))
        {
          uint udtPrice = udtOracle.updateAndConsult(udt, 10 ** 18, weth);

          uint tokensToDistribute = (estimatedGasForPurchase * tx.gasprice) * (125
* 10 ** 18) / 100 / udtPrice;

          uint maxTokens = 0;
          if (chainId > 1)
          {
            maxTokens = IMintableERC20(udt).balanceOf(address(this)) * valueInETH /
(2 + 2 * valueInETH / grossNetworkProduct) / grossNetworkProduct;
          } else {
            maxTokens = IMintableERC20(udt).totalSupply() * valueInETH / 2 /
grossNetworkProduct;
          }

          if(tokensToDistribute > maxTokens)
          {
            tokensToDistribute = maxTokens;
          }

          if(tokensToDistribute > 0)
          {
            uint devReward = tokensToDistribute * 20 / 100;
            if (chainId > 1)
            {
              uint balance = IMintableERC20(udt).balanceOf(address(this));
              if (balance > tokensToDistribute) {
                IMintableERC20(udt).transfer(_referrer, tokensToDistribute -
devReward);
                IMintableERC20(udt).transfer(owner(), devReward);
              }
            } else {
              IMintableERC20(udt).mint(_referrer, tokensToDistribute - devReward);
              IMintableERC20(udt).mint(owner(), devReward);
            }
          }
        }
      }
    }
  }
```

Recommendation

Add a check to ensure that `udtPrice > 0` before performing the division. This prevents division by zero and allows for graceful error handling or fallback logic.

24. Incorrect token enumeration logic in tokenByIndex

Severity	Low
Finding ID	BH-2021-11-unlock-24
Target	smart-contracts/contracts/mixins/MixinERC721Enumerable.sol
Function name	tokenByIndex

Description

The tokenByIndex function incorrectly assumes that token IDs are sequential and tightly packed starting from 0. It simply returns the index as the tokenId, which violates the ERC-721 Enumerable specification. This can lead to returning invalid or non-existent token IDs, especially if tokens are minted non-sequentially, burned, or if gaps exist.

```
function tokenByIndex(
    uint256 _index
  ) public view
    returns (uint256)
  {
    require(_index < _totalSupply, 'OUT_OF_RANGE');
    return _index;
  }
```

Recommendation

Maintain a mapping (e.g., tokenByIndex[index] = tokenId) to track valid token IDs in the order of minting. Ensure the function returns the actual token ID at the given index in the list of valid tokens.

25. Improper Input Validation on Refund Penalty Parameters

Severity	Low
Finding ID	BH-2021-11-unlock-25
Target	smart-contracts/contracts/mixins/MixinRefunds.sol
Function name	updateRefundPenalty

Description

The `updateRefundPenalty` function lacks validation on both `_refundPenaltyBasisPoints` and `_freeTrialLength` parameters. This could result in misconfiguration: for example, setting `_refundPenaltyBasisPoints` greater than `BASIS_POINTS_DEN` (100%) would result in full penalty with no refund. Additionally, `_freeTrialLength` could be set to a value that exceeds the lock's expiration duration, leading to logical inconsistencies and unexpected refund behavior.

```
function updateRefundPenalty(
    uint _freeTrialLength,
    uint _refundPenaltyBasisPoints
  ) external
    onlyLockManager
  {
    emit RefundPenaltyChanged(
      _freeTrialLength,
      _refundPenaltyBasisPoints
    );

    freeTrialLength = _freeTrialLength;
    refundPenaltyBasisPoints = _refundPenaltyBasisPoints;
  }
```

Recommendation

Add input validation to ensure `_refundPenaltyBasisPoints <= BASIS_POINTS_DEN` and `_freeTrialLength <= expirationDuration` or another sensible upper bound. This prevents accidental or malicious misconfiguration.

26. Refund Based on Current Key Price May Lead to Inconsistent User Compensation

Severity	Low
Finding ID	BH-2021-11-unlock-26
Target	smart-contracts/contracts/mixins/MixinRefunds.sol
Function name	_getCancelAndRefundValue

Description

The `_getCancelAndRefundValue` function calculates the refund using the current `keyPrice`. However, if the `keyPrice` has changed since the key was originally purchased, the user may be refunded an amount that does not reflect their original purchase price, leading to over- or under-compensation.

```
function _getCancelAndRefundValue(
    address _keyOwner
  )
    private view
    hasValidKey(_keyOwner)
    returns (uint refund)
  {
    Key storage key = keyByOwner[_keyOwner];
    uint timeRemaining = key.expirationTimestamp - block.timestamp;
    if(timeRemaining + freeTrialLength >= expirationDuration) {
      refund = keyPrice;
    } else {
      refund = keyPrice * timeRemaining / expirationDuration;
    }

    if(freeTrialLength == 0 || timeRemaining + freeTrialLength < expirationDuration)
    {
      uint penalty = keyPrice * refundPenaltyBasisPoints / BASIS_POINTS_DEN;
      if (refund > penalty) {
        refund -= penalty;
      } else {
        refund = 0;
      }
    }
  }
```

Recommendation

Store the key price at the time of purchase for each key and use that stored value for refund calculations instead of the current global `keyPrice`.