4rena

Unlock Protocol contest Findings & Analysis Report

2022-04-20

TABLE OF CONTENTS

Top

Overview

ABOUT C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Unlock Protocol smart contract system written in Solidity. The audit contest took place between November 18—November 24, 2021.

WARDENS

27 Wardens contributed reports to the Unlock Protocol contest:

1. WatchPug (jtp and ming)
2. cmichel
3. elprofesor
4. kenzo
5. pauliax
6. GiveMeTestEther
7. 0x0x0x
8. itsmeSTYJ
9. loop
10. Ruhum
11. defsec
12. Meta0xNull
13. harleythedog
14. Jujic
15. hagrid
16. jayjonah8
17. Reigada
18. HardlyDifficult

19. ye0lde
20. TomFrenchBlockchain
21. nathaniel
22. gzeon
23. BouSalman
24. mics
25. sabtikw
26. aga7hokakological

This contest was judged by 0xleastwood.

Final report assembled by itsmetechjay, CloudEllie, and liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 49 unique vulnerabilities and 151 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 13 received a risk rating in the category of MEDIUM severity, and 32 received a risk rating in the category of LOW severity.

C4 analysis also identified 54 non-critical recommendations and 48 gas optimizations.

Scope

The code under review can be found within the C4 Unlock Protocol contest repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 604 source lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (4)

[H-01] MEV MINER CAN MINT LARGER THAN EXPECTED UDT TOTAL SUPPLY

Submitted by elprofesor

`UnlockProtocol` attempts to calculate gas reimbursement using `tx.gasprice`, typically users who falsify `tx.gasprice` would lose gas to miners and therefore not obtain any advantage over the protocol itself. This does present capabilities for miners to extract value, as they can submit their own transactions, or cooperate with a malicious user, reimbursing a portion (or all) or the `tx.gasprice` used. As the following calculation is made;

```
        uint tokensToDistribute = (estimatedGasForPurchase * tx.gasprice) * (125 *
```

we can see that arbitrary `tx.gasprices` can rapidly inflate the `tokensToDistribute`. Though capped at maxTokens, this value can be up to half the total supply of UDT, which could dramatically affect the value of UDT potentially leading to lucrative value extractions outside of the pool.

Recommended Mitigation Steps

Using an oracle service to determine the average gas price and ensuring it is within some normal bounds that has not been subjected to arbitrary value manipulation.

julien51 (Unlock Protocol) disputed and commented:

>> we can see that arbitrary tx.gasprices can rapidly inflate the tokensToDistribute. Though capped at maxTokens, this value can be up to half the total supply of UDT, which could dramatically affect the value of UDT potentially leading to lucrative value extractions outside of the pool.
>
> As you noted it would be capped by the actual increase of the GDP transaction.
>
> However we could indeed use an oracle to determine the average gas price over a certain number of blocks to limit the risk even further.

0xleastwood (judge) commented:

> I think the warden has raised a valid issue of value extractions. Whether the value extracted is capped at a certain number of tokens, I don't think the issue is nullified as a result. Miners can realistically fill up blockspace by abusing this behaviour and then selling netted tokens on the open market. I'll consider marking this as `medium`, what do you think @julien51 ?

0xleastwood (judge) commented:

> I think `maxTokens` will be set to `IMintableERC20(udt).totalSupply() / 2` upon the first call to `recordKeyPurchase()`. If I'm not mistaken, this could allow a malicious miner could effectively distribute half of the token supply in one tx.

0xleastwood (judge) commented:

> After further offline discussions with @julien51. We agree that this is an issue that needs to be addressed.
>
> If we consider real-world values for `IMintableERC20(udt).totalSupply()` and `IMintableERC20(udt).totalSupply()` as `1_000_000e18` and `400e18` respectively. Then a miner could mint up to ~1247 `UDT` tokens valued at $USD 124,688 if they provide a single Ether as their purchase amount. Obviously this can be abused to generate a huge amount of profit for miners, so as this is a viable way to extract value from the protocol, I will be keeping this as `high` severity.

[H-02] WRONG DESIGN/IMPLEMENTATION OF FREETRIAL ALLOWS ATTACKER TO STEAL FUNDS FROM THE PROTOCOL

Submitted by WatchPug

The current design/implementation of `freeTrial` allows users to get full refund before the `freeTrial` ends. Plus, a user can transfer partial of their time to another user using `shareKey`.

This makes it possible for the attacker to steal from the protocol by transferring `freeTrial` time from multiple addresses to one address and adding up to `expirationDuration` and call refund to steal from the protocol.

Proof of Concept

Given:

- `keyPrice` is 1 ETH;
- `expirationDuration` is 360 days;
- `freeTrialLength` is 31 days.

The attacker can create two wallet addresses: Alice and Bob.

1. Alice calls `purchase()`, transfer 30 days via `shareKey()` to Bob, then calls `cancelAndRefund()` to get full refund; Repeat 12 times;
2. Bob calls `cancelAndRefund()` and get 1 ETH.

Recommendation

Consider disabling `cancelAndRefund()` for users who transferred time to another user.

julien51 (Unlock Protocol) confirmed and commented:

> I think this is valid! The free trial approach is indeed a risk on that front and we need to “warn” lock managers about this more.
>
> For lock manager who still want to offer free trials, the best approach would probably be to set a high transfer fee to make sure that free trials cannot be transfered.
>
> As a consequence of this, I am not sure this is as critical as indicated by the submitter.

0xleastwood (judge) commented:

> Nice find!
>
> From what I can tell at least, this does seem like a viable attack vector. Can I ask why this should not be treated as `high` risk? @julien51

julien51 (Unlock Protocol) commented:

> Sorry for the long delay here. In short: this is valid, but only an issue for locks which are enabling free trials (no one has done it) and we would make sure our

> UI shows this as a potential issue. In other words: a lock manager would need to *explicitly* enable free trials, despite our warning to put their own funds at risk. For that reason I don't think this is "High".

0xleastwood (judge) commented:

> While this is a valid issue pertaining only to lock managers who *explicitly* enable free trials, this may still lead to a loss of funds if `cancelAndRefund` is called by a user who has transferred their time to another account. I still believe this deserves a `high` severity rating.
>
> In my honest opinion, a warning isn't sufficient to prevent such abuse. I think on-chain enforcement ideal in this situation.

[H-03] MIXINTRANSFER.SOL#TRANSFERFROM WRONG IMPLEMENTATION CAN POTENTIALLY ALLOWS ATTACKERS TO REVERSE TRANSFER AND CAUSE FUND LOSS TO THE USERS

Submitted by WatchPug

https://github.com/code-423n4/2021-11-unlock/blob/ec41eada1dd116bcccc5603ce342257584bec783/smart-contracts/contracts/mixins/MixinTransfer.sol#L131-L152

```
if (toKey.tokenId == 0) {
  toKey.tokenId = _tokenId;
  _recordOwner(_recipient, _tokenId);
  // Clear any previous approvals
  _clearApproval(_tokenId);
}

if (previousExpiration <= block.timestamp) {
  // The recipient did not have a key, or had a key but it expired. The new exp
  // An expired key is no longer a valid key, so the new tokenID is the sender'
  toKey.expirationTimestamp = fromKey.expirationTimestamp;

  toKey.tokenId = _tokenId;
```

```
      // Reset the key Manager to the key owner
      _setKeyManagerOf(_tokenId, address(0));

      _recordOwner(_recipient, _tokenId);
    } else {
      // The recipient has a non expired key. We just add them the corresponding re
      // SafeSub is not required since the if confirms `previousExpiration - block.
      toKey.expirationTimestamp = fromKey.expirationTimestamp + previousExpiration
    }
```

Based on the context, L131-136 seems to be the logic of handling the case of the recipient with no key, and L138-148 is handing the case of the recipient's key expired.

However, in L131-136, the key manager is not being reset.

This allows attackers to keep the role of key manager after the transfer, and transfer the key back or to another recipient.

Proof of Concept

Given:

- Alice owns a key that is valid until 1 year later.
- Alice calls `setKeyManagerOf()`, making herself the keyManager;
- Alice calls `transferFrom()`, transferring the key to Bob; Bob might have paid a certain amount of money to Alice upon receive of the key;
- Alice calls `transferFrom()` again, transferring the key back from Bob.

Recommendation

Consider resetting the key manager regardless of the status of the recipient's key.

julien51 (Unlock Protocol) confirmed:

> I *think* you are onto something here. We will need to investigate further and reproduce to fix!

0xleastwood (judge) commented:

> @julien51 Just following up if you were able to double-check this?

julien51 (Unlock Protocol) confirmed:

> This is indeed valid and I think we will need to “patch” this. We’re still unsure how but we’re exploring multiple ways.

[H-04] APPROVALS NOT CLEARED AFTER KEY TRANSFER

Submitted by cmichel

The locks implement three different approval types, see `onlyKeyManagerOrApproved` for an overview:

- key manager (map `keyManagerOf`)
- single-person approvals (map `approved`). Cleared by `_clearApproval` or `_setKeyManagerOf`
- operator approvals (map `managerToOperatorApproved`)

The `MixinTransfer.transferFrom` requires any of the three approval types in the `onlyKeyManagerOrApproved` modifier on the tokenId to authenticate transfers from `from`.

Notice that if the `to` address previously had a key but it expired only the `_setKeyManagerOf` call is performed, which does not clear `approved` if the key manager was already set to 0:

```
function transferFrom(
  address _from,
  address _recipient,
  uint _tokenId
)
  public
  onlyIfAlive
  hasValidKey(_from)
```

```
    hasValidKey(_from)
    onlyKeyManagerOrApproved(_tokenId)
  {
    // @audit this is skipped if user had a key that expired
    if (toKey.tokenId == 0) {
      toKey.tokenId = _tokenId;
      _recordOwner(_recipient, _tokenId);
      // Clear any previous approvals
      _clearApproval(_tokenId);
    }

    if (previousExpiration <= block.timestamp) {
      // The recipient did not have a key, or had a key but it expired. The new e
      // An expired key is no longer a valid key, so the new tokenID is the sende
      toKey.expirationTimestamp = fromKey.expirationTimestamp;
      toKey.tokenId = _tokenId;

      // Reset the key Manager to the key owner
      // @audit  doesn't clear approval if key manager already was 0
      _setKeyManagerOf(_tokenId, address(0));

      _recordOwner(_recipient, _tokenId);
    }
    // ...
  }

  //
  function _setKeyManagerOf(
    uint _tokenId,
    address _keyManager
  ) internal
  {
    // @audit-ok only clears approved if key manager updated
    if(keyManagerOf[_tokenId] != _keyManager) {
      keyManagerOf[_tokenId] = _keyManager;
      _clearApproval(_tokenId);
      emit KeyManagerChanged(_tokenId, address(0));
    }
  }
```

Impact

It’s possible to sell someone a key and then claim it back as the approvals are not always cleared.

Proof Of Concept

- Attacker A has a valuable key (`tokenId = 42`) with an expiry date far in the future.
- A sets approvals for their second attacker controlled account A' by calling `MixinKeys.setApprovalForAll(A', true)`, which sets `managerToOperatorApproved[A][A'] = true`.
- A clears the key manager by setting it to zero, for example, by transferring it to a second account that does not have a key yet, this calls the above `_setKeyManagerOf(42, address(0));` in `transferFrom`
- A sets single-token approval to A' by calling `MixinKeys.approve(A', 42)`, setting `approved[42] = A'`.
- A sells the token to a victim V for a discount (compared to purchasing it from the Lock). The victim needs to have owned a key before which already expired. The `transferFrom(A, V, 42)` call sets the owner of token 42 to `V`, but does not clear the `approved[42] == A'` field as described above. (`_setKeyManagerOf(_tokenId, address(0));` is called but the key manager was already zero, which then does not clear approvals.)
- A' can claim back the token by calling `transferFrom(V, A', 42)` and the `onlyKeyManagerOrApproved(42)` modifier will pass as `approved[42] == A'` is still set.

Recommended Mitigation Steps

The `_setKeyManagerOf` function should not handle clearing approvals of single-token approvals (`approved`) as these are two separate approval types. The `transferFrom` function should always call `_clearApproval` in the `(previousExpiration <= block.timestamp)` case.

julien51 (Unlock Protocol) confirmed and commented:

> Thanks for reporting this. This is valid and we will fix it.

Medium Risk Findings (13)

[M-01] UNLOCK: FREE UDT ARBITRAGE OPPORTUNITY

Submitted by itsmeSTYJ

Uniswap v2 made oracle attacks much more expensive to execute (since it needs to be manipulated over X number of blocks) however its biggest drawback is that it reacts slow to price volatility (depends on how far back you look). Depending on a single oracle is still very risky and can be exploited given the correct conditions.

Assuming the ideal conditions, it is possible to purchase many keys across many locks for the UDT token that is distributed to the referrer and sell them on some other exchanges where the price of UDT is higher; high enough such that the malicious user can still profit even after requesting for a refund (w/ or w/o a free trial).

Proof of Concept

This exploit is made possible because of:

- the over dependency on a single price oracle
- UDT token distribution logic is flawed

The following assumptions has to be true for this attack to work:

1. price of UDT on an exchange is much higher than that from the price retrieved from the `uniswapOracle`.
2. Since the price retrieved by `udtOracle.updateAndConsult()` only updates once per day, it is slow to react to the volatility of UDT price movements.
3. Malicious user creates a lock and buys many keys across multiple addresses.
4. Malicious user sells these UDT tokens on the exchanges w/ the higher price.

5. Malicious user requests for a refund on the keys owned.
6. Repeat until it is no longer profitable i.e. price on other exchanges become close to parity with the price retrieved by the `uniswapOracle`.

Recommended Mitigation Steps

- Use the average of multiple oracle sources so that the price of UDT tokens (from `Unlock.sol`'s PoV) reacts faster.
- UDT tokens distributed based on the duration of key ownership.

julien51 (Unlock Protocol) disagreed with severity and commented:

> As you noted this is pretty theoretical and given that the amount of UDT minted is capped to the gas spent, the user will need to 1) purchase a LOT of keys and 2) cancel them all and 3) find an exchange where the price is significantly different.

0xleastwood (judge) commented:

> Nice find!
>
> While, I do agree this is a difficult attack to perform, it is still a valid way of extracting value from the protocol. Hence, I believe this should be kept as `medium`.
>
> ```
> 2 — Med (M): vulns have a risk of 2 and are considered "Medium" severity
> when assets are not at direct risk, but the function of the protocol or
> its availability could be impacted, or leak value with a hypothetical
> attack path with stated assumptions, but external requirements.
> ```

julien51 (Unlock Protocol) commented:

> We will mitigate this in an upcoming upgrade by moving to Uniswap v3 for our oracles.

[M-02] POTENTIAL ECONOMIC ATTACK ON UDT GRANTS TO THE REFERRER

Submitted by WatchPug

In the current implementation, `Unlock.sol#recordKeyPurchase()` will send `estimatedGasForPurchase * tx.gasprice` worth of UDT to the referrer.

https://github.com/code-423n4/2021-11-unlock/blob/ec41eada1dd116bcccc5603ce342257584bec783/smart-contracts/contracts/Unlock.sol#L325-L325

```
    uint tokensToDistribute = (estimatedGasForPurchase * tx.gasprice) * (125 * 10 *
```

We believe there are multiple potential economic attack vectors to exploit this.

If `estimatedGasForPurchase` is misconfigured to a higher amount than the actual avg gas cost for a purchase call, or future network upgrades make the actual gas cost become lower than the configured `estimatedGasForPurchase`, it can be exploited simply by creating a lock and call `purchase()` many times to mint UDT.

Even if `estimatedGasForPurchase` is configured to an amount similar to the actual gas cost, a more sophisticated attack is still possible:

Proof of Concept

Given:

- `estimatedGasForPurchase` is configured as `200,000`;
- The gas cost of a regular purchase call is about `200,000`.

The attacker can create a lock contract and set the token address to a special gas saving token, which will SELFDESTRUCT to get a gas refund on `transfer`.

The attacker can:

1. Mint gas saving token with gas price: `1 gwei`;

2. Call `purchase()` and use 48 contract slots with gas price: `1000 gwei`;

Total gas saved will be ~0.8 ETH (or other native tokens, eg. BNB, MATIC). Therefore, the attacker will profit ~0.8 ETH worth of UDT.

See: https://gastoken.io/

Recommendation

Consider setting a global daily upper limit of total UDT grants to referrers, plus, an upper limit for UDT minted per purchase.

julien51 (Unlock Protocol) acknowledged, but disagreed with severity and commented:

> > If `estimatedGasForPurchase` is misconfigured to a higher amount than the actual avg gas cost for a purchase call, or future network upgrades make the actual gas cost become lower than the configured `estimatedGasForPurchase`, it can be exploited simply by creating a lock and call `purchase()` many times to mint UDT.
>
> Absolutely but considering the security model, the admin indeed have full control over the protocol. We are thinking about finding a mechanism to not hardcode gas spent but use the actual number eventually. When we do that we should consider the impact of things like gas-token (even though EIP1559 has probably made them mostly impractical?).
>
> At this point given that the gas spent is hardcoded, there is a de-facto cap on how much UDT they could earn (based on the token price).

0xleastwood (judge) commented:

> While I agree with the warden, there is potential for value extraction, however, it does require the admin to be unaware about upcoming network upgrades.
>
> As the sponsor has noted, they will be moving towards a dynamic `estimatedGasForPurchase` value, however, from the perspective of the c4 contest, this doesn’t change the outcome of my decision.

> As the protocol may leak value based on certain network assumptions, I'll mark this as `medium` severity.
>
> ```
> 2 — Med (M): vulns have a risk of 2 and are considered "Medium" severity
> when assets are not at direct risk, but the function of the protocol or
> its availability could be impacted, or leak value with a hypothetical
> attack path with stated assumptions, but external requirements.
> ```

Please note: the following additional discussions and re-assessment took place approximately 2 months after judging and awarding were finalized. As such, this report will leave this finding in its originally assessed risk category as it simply reflects a snapshot in time.

julien51 (Unlock Protocol) commented:

> I don't think the protocl can "leak" value based on that. The tokens that are used to compute GDP and distribute tokens have to be approved by the DAO (right now only USDC, DAI and BAT have been approved on mainnet, and only USDC on Polygon). I don't think the DAO would approve gas tokens givem that indeed they could result in leakage of UDT, so I think it is `minor`.

0xleastwood (judge) decreased severity to Low and commented:

> Considering the sponsor's comments, I actually agree that this is less likely than initially stated. Similar to the `SafeERC20` issue, it isn't expected that gas saving tokens will be approved to compute and distribute `UDT` tokens. I'll downgrade this to `low`.

[M-03] SUPPORT OF DIFFERENT ERC20 TOKENS

Submitted by pauliax, also found by cmichel, Reigada, kenzo, Ruhum, 0x0x0x, GiveMeTestEther, and WatchPug

The current version of the codebase does not handle special cases of tokens, e.g. deflationary, rebasing, or those that return true/false on success (see: https://github.com/d-xo/weird-erc20). Function purchase transfers tokens from msg.sender but it does not check the return value, nor how many tokens were actually transferred:

```
    token.transferFrom(msg.sender, address(this), pricePaid);
```

Recommended Mitigation Steps

I have 2 suggestions here:

1. Use SafeERC20 library to handle token transfers: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol
2. Consider checking the actual balances transferred (balance after-before) or clearly documenting that you do not support deflationary / rebasing / etc tokens.

julien51 (Unlock Protocol) disputed and commented:

> The only party that would be penalized in the examples you describe is the lock manager (and beneficiary) who has explicitly deployed the lock using the (noncompliant) ERC20. If we consider the threat model here then I think this is not really an issue, as additional checks would incur a gas cost for everyone.

Please note: the following additional discussions and re-assessment took place approximately 2 months after judging and awarding were finalized. As such, this report will leave this finding in its originally assessed risk category as it simply reflects a snapshot in time.

julien51 (Unlock Protocol) commented:

> The fact that this requires an explicit action by the lock manager (ie using a buggy/malicious ERC20 token) and that it puts only *their* tokens at risk, I think this is `minor`.

0xleastwood (judge) decreased severity to Low and commented:

> Giving this a bit more thought, I think its always safe to enforce these checks rather than leave it up to the lock manage to potentially make the mistake and then be liable for this mistake later on. However, considering the threat model, I do think this is better suited as a `low` severity issue.

[M-04] KEY BUYERS WILL NOT BE ABLE TO GET REFUND IF LOCK MANAGER WITHDRAWS PROFITS

Submitted by kenzo

Unlock contains a feature in which a key buyer can ask for a refund. The refund is sent from the lock - where the purchase funds were sent. The lock manager can withdraw all funds from the lock. Therefore, if the lock manager withdraws enough profits from the lock, the user would not be able to cancel his key and request refund. Even if a lock manager is not malicious, if he would want to enable users to cancel their key, he would have to keep track of how much tokens need to be kept in the contract in order to enable this - not a trivial calculation. A naive lock manager might accidentally disable refunds for his clients.

Impact

Refunds are not guaranteed. A user might buy a key expecting to cancel it within some time, only to discover he can not cancel it. (This loss of user funds is why I consider this a high risk finding.) An unaware lock manager who just wants to withdraw all his profits might accidentally discover that he removed his users’ ability to cancel their key.

Notes

It seems the Unlock team is aware to some extent that withdrawing breaks refunds, as they state in the `withdraw` function:

```
* TODO: consider allowing anybody to trigger this as long as it goes to owne
*  -- however be wary of draining funds as it breaks the `cancelAndRefund` a
* use cases.
```

However, even if just the owner is allowed to call it, he may break the refund functionality - on purpose or accidentally. Looking on Unlock documentation I don't see a warning to creators about withdrawing their funds.

Proof of Concept

`withdraw` function has no limit on the amount withdrawn, therefore the owner can withdraw all funds: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinLockCore.sol#L133:#L162

`cancelAndRefund` transfers the funds from the same lock contract: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinRefunds.sol#L118 Therefore if there are not enough funds, the transfer will fail.

Recommended Mitigation Steps

Perhaps a sort of MasterChef-like shares system can be implemented in order to make sure the owner leaves enough funds in the lock to process refunds.

julien51 (Unlock Protocol) disagreed with severity and commented:

> As noted, this is actually documented. You are right though that we should make this more obvious on the UI. I would not classify this as High Risk.

0xleastwood (judge) decreased severity to Medium and commented:

> Nice find! I think this can be downgraded to `medium` as the availability of the protocol is impacted by this issue.

julien51 (Unlock Protocol) commented:

> This would not affect the whole protocol but only the "malicious" lock and it is impractice not only documented but also how these things work in the real world. If Netflix went out of business tomorrow, I could not get a refund on this month's membership fee…

0xleastwood (judge) commented:

> While I mostly agree with the sponsor, this may be intended behaviour as user's should not be entitled to a refund in this case. However, based on what was known at the time, it seemed like this broke the functionality of `cancelAndRefund` and `expireAndRefundFor` functions, hence why it was marked as `medium` severity.

[M-05] REFUND MECHANISM DOESN'T TAKE INTO ACCOUNT THAT KEY PRICE CAN CHANGE

Submitted by kenzo, also found by WatchPug and 0x0x0x

Lock manager can change key pricing. The refund mechanism calculates refund according to current key price, not price actually paid.

Impact

A user refunding can get less (or more) funds than deserved.

Proof of Concept

Refund only takes the current price into account: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinRefunds.sol#L144:#L152

Lock manager can update key price at any point, and the old price is not saved anywhere: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinLockCore.sol#L183

So if for example a key price has gone down, a user who tried to refund will get less funds than deserved.

Recommended Mitigation Steps

Consider saving the amount the user paid, and refund according to that. Or having a kind of a price snapshot/version mechanism.

julien51 (Unlock Protocol) acknowledged, but disagreed with severity and commented:

> This is a known issue… but indeed we should show things in the UI to indicate things to users.

0xleastwood (judge) decreased severity to Medium and commented:

> Agree this sounds like an issue! However, I don't think this can be justified as a `high` risk issue. But it does seem that the protocol could leak value and impact users, so marking this as `medium`.

julien51 (Unlock Protocol) commented:

> We actually are adding a new mechanism to keep track of the last price paid by any user which means we could use it in the next version to solve this issue!

[M-06] KEY TRANSFER WILL DESTROY KEY IF FROM==TO

Submitted by kenzo, also found by GiveMeTestEther and cmichel

If calling `transferFrom` with `_from == _recipient`, the key will get destroyed (meaning the key will be set as expired and set the owner's key to be 0).

Impact

A key manager or approved might accidentally destroy user's token.

Note: this requires user error and so I'm not sure if this is a valid finding. However, few things make me think that it is valid:

- Unlock protocol checks for transfer to 0-address, so some input validation is there
- Since other entities other than the owner can be allowed to transfer owner's token, it might be best to make sure such accidental mistake could not happen.

- This scenario manifests a unique and probably unintended behavior

Proof of Concept

By following `transferFrom`'s execution: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinTransfer.sol#L109:#L166 One can see that in the case where `_from == _recipient` with a valid key:

- The function will deduct transfer fee from the key
- The function will incorrectly add more time to the key's expiration (L151)
- The function will expire and reset the key (L155)

Therefore, the user will lose his key without getting a refund.

Recommended Mitigation Steps

Add a require statement in the beginning of `transferFrom`:
`require(_from != _recipient, 'TRANSFER_TO_SELF');`

julien51 (Unlock Protocol) confirmed

julien51 (Unlock Protocol) commented:

> Fixed since then :)

[M-07] MIXINPURCHASE:SHAREKEY ALLOWS TO GENERATE KEYS WITHOUT PURCHASING

Submitted by GiveMeTestEther, also found by kenzo

The `shareKey` function allows a user to share some time with another user that doesn't already has/had a key and this generates a new key. This even allows the user to generate more keys than `\_maxNumberOfKeys`.

Attacker generates a lot of EOA addresses, buys a key, share the minimum necessary time with each address and in each “sharing” a new key gets generated. This allows cheaply to allocate alot of “keys” with out really purchasing them and a lot of user can’t get a “key” because purchase has a modifier notSoldOut, that limits the max purchasable to “keys” to `maxNumberOfKeys`

Proof of Concept

Provide direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept.

Tools Used

Manual Analysis

Recommended Mitigation Steps

- rethink the whole `shareKey` thingy,

julien51 (Unlock Protocol) acknowledged and commented:

> I am not sure this is a bug or even a risk. Someone could actually achieve the same thing by purchasing keys at the full price and cancelling them immediately getting an almost full refund (or even full refund when there is a free trial) and could quickly get the lock to “sell out”. It is actually the case with any NFT project where there is a cap/limit to number of tokens and someone can easily “Capture” them all.
>
> One way to limit the impact for the lock manager would be to set a cancellation penalty AND a transfer fee.

0xleastwood (judge) commented:

> Nice find! While I understand what the sponsor is saying, this does seem like a valid way to deny a lock from selling membership to honest users.

julien51 (Unlock Protocol) commented:

> Note that a lock manager can easily increase supply to mitigate that (and even could *delete* existing keys/NFT to reduce the outstanding supply)

0xleastwood (judge) commented:

> While the lock manager can restore the contract to some valid state, this will still impact protocol availability, even in the short-term.

[M-08] FRONTRUNNING `PUBLICLOCK.INITIALIZE()` CAN PREVENT UPGRADES DUE TO INSUFFICIENT ACCESS CONTROL

Submitted by elprofesor, also found by kenzo

The unlock protocols base contract `Unlock.sol` uses `setLocktemplate()` to initialize the implementation contract for the `PublicLock` proxy. This function will initialize the relevant `PublicLock` contract which has been deployed separately. `PublicLock.initialize()` does not have any relevant access control and does not prevent arbitrary users from initialising. This means that a malicious user could front run the `setLocktemplate()` forcing the deployer of `PublicLock`'s implementation to redeploy. The process can be repeated, which costs the malicious user less than it would the owner of the Unlock Protocol, potentially unnecessarily draining funds from the development team.

Proof of Concept

Lack of access control on initialize

Recommended Mitigation Steps

Implement valid access control on the `PublicLock` contract to ensure only the relevant deployer can `initialize()`.

julien51 (Unlock Protocol) disagreed with severity

0xleastwood (judge) commented:

> I agree with the warden, `_publicLockAddress` is deployed separately and hence `initialize` can be called before `setLockTemplate` is called.

[M-09] REFERRER DISCOUNT TOKEN AMOUNT CAN BE MANIPULATED

Submitted by cmichel

The `Unlock.recordKeyPurchase` function is called on each key purchase (`MixinPurchase.purchase`) and mints UDT tokens to the referrer. The amount to mint is based on the transaction's gas price which is controlled by the caller (purchaser):

```
uint tokensToDistribute = (estimatedGasForPurchase * tx.gasprice) * (125 * 10 *
```

Impact

Tokens can be minted by purchasing a key with themself as the referrer at a high transaction gas price. Depending on the UDT price on external markets, it could be profitable to buy a key at a high gas price, receive UDT and then sell them on a market for a profit.

Recommended Mitigation Steps

The amount minted should be more predictable and not depend on the user's gas price input. Consider declaring an *average gas price* storage variable that is set by a trusted party and use this one instead.

julien51 (Unlock Protocol) disagreed with severity and commented:

> Depending on the UDT price on external markets, it could be profitable to buy a key at a high gas price, receive UDT and then sell them on a

>   market for a profit.
>
> Since we get the token price from the Uniswap oracle, the amount of tokens received is always at most equal to what they would have spent to acquire them on Uniswap.

0xleastwood (judge) commented:

> As the uniswap oracle provides averaged price data, if there is any discrepancy between the spot price and the TWAP price, this can definitely be abused to extract value from the protocol. Keeping this as `medium`.

[M-10] INACCURATE FEES COMPUTATION

Submitted by cmichel, also found by 0x0x0x

The `MixinTransfer.shareKey` function wants to compute a fee such that `time + fee * time == timeRemaining (timePlusFee)`:

```
uint fee = getTransferFee(keyOwner, _timeShared);
uint timePlusFee = _timeShared + fee;
```

However, if the time remaining is less than the computed fee time, the computation changes and a different formula is applied. The fee is now simply taken on the remaining time.

```
if(timePlusFee < timeRemaining) {
  // now we can safely set the time
  time = _timeShared;
  // deduct time from parent key, including transfer fee
  _timeMachine(_tokenId, timePlusFee, false);
} else {
  // we have to recalculate the fee here
  fee = getTransferFee(keyOwner, timeRemaining);
```

```
        // @audit want it such that time + fee * time == timeRemaining, but fee is ta
        time = timeRemaining - fee;
      }
```

It should compute the `time` without fee as `time = timeRemaining / (1.0 + fee_as_decimal)` instead, i.e., `time = BASIS_POINTS_DEN * timeRemaining / (transferFeeBasisPoints + BASIS_POINTS_DEN)`.

Proof Of Concept

To demonstrate the difference with a 10% fee and a `_timeShared = 10,000s` which should be credited to the `to` account.

The correct time plus fee which is reduced from `from` (as in the `timePlusFee < timeRemaining` branch) would be `10,000 + 10% * 10,000 = 11,000`.

However, if `from` has not enough time remaining and `timePlusFee >= timeRemaining`, the entire time remaining is reduced from `from` but the credited `time` is computed wrongly as: (Let's assume `timeRemaining == timePlusFee`): `time = 11,000 - 10% * 11,000 = 11,000 - 1,100 = 9900`.

They would receive 100 seconds less than what they are owed.

Impact

When transferring more time than the `from` account has, the credited time is scaled down wrongly and the receiver receives less time (a larger fee is applied).

Recommended Mitigation Steps

It should change the first `if` branch condition to `timePlusFee <= timeRemaining` (less than or equal). In the `else` branch, it should compute the time without fee as

```
time = BASIS_POINTS_DEN * timeRemaining / (transferFeeBasisPoints +
BASIS_POINTS_DEN)
```

.

julien51 (Unlock Protocol) confirmed and commented:

> Great find!

[M-11] MISSING SCALING FACTOR IN RECORDKEYPURCHASE?

Submitted by cmichel

The `Unlock.recordKeyPurchase` function computes the `maxTokens` as:

```
maxTokens = IMintableERC20(udt).balanceOf(address(this)) * valueInETH / (2 + 2
```

Note that `grossNetworkProduct` was already increased by `valueInETH` in the code before. Meaning, the `(2 + 2 * valueInETH / grossNetworkProduct)` part of the computation will almost always be `2` as usually `grossNetworkProduct > 2 * valueInETH`, and thus the `2 * valueInETH / grossNetworkProduct` is zero by integer division.

Impact

The `maxTokens` curve might not be computed as intended and lead to being able to receive more token rewards than intended.

Recommended Mitigation Steps

The comment “we distribute tokens using asymptotic curve between 0 and 0.5” should be more clear to indicate how exactly the curve looks like. It could be that a floating-point number was desired instead of the integer division in

`2 * valueInETH / grossNetworkProduct`. In that case, consider adding a scaling factor to this term and divide by it at the end of the computation again.

julien51 (Unlock Protocol) commented:

> I am not fully sure I understand what the problem is here?

0xleastwood (judge) commented:

> I think the warden is raising an issue where `2 * valueInEth / grossNetworkProduct` will more than likely truncate and return `0`. I think this is a valid finding.

julien51 (Unlock Protocol) commented:

> Hum, we did some tests and could not reproduce here.

0xleastwood (judge) commented:

> I'm not sure how `2 * valueInETH / grossNetworkProduct` does not always lead to some truncation. `grossNetworkProduct` is equal to `valueInETH` in the first call but always greater than `valueInETH` in any subsequent calls.

[M-12] MISSING MAXNUMBEROFKEYS CHECKS IN SHAREKEY AND GRANTKEY

Submitted by kenzo

More keys can be minted than maxNumberOfKeys since `shareKey` and `grantKey` do not check if the lock is sold out.

Impact

More keys can be minted than intended.

Proof of Concept

In both `shareKey` and `grantKey`, if minting a new token, a new token is simply minted (and `_totalSupply` increased) without checking it against `maxNumberOfKeys`. This is unlike `purchase`, which has the `notSoldOut` modifier.

`grantKey`: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinGrantKeys.sol#L41:#L42

`shareKey`: https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinTransfer.sol#L83:#L84 Both functions call `_assignNewTokenId` which does not check maxNumberOfKeys.

https://github.com/code-423n4/2021-11-unlock/blob/main/smart-contracts/contracts/mixins/MixinKeys.sol#L311:#L322 So you can say that `_assignNewTokenId` is actually the root of the error, and this is why I am submitting this as 1 finding and not 2 (for grantKey/shareKey).

Recommended Mitigation Steps

Add a check to `_assignNewTokenId` that will revert if we need to record a new key and `maxNumberOfKeys` has been reached.

julien51 (Unlock Protocol) confirmed and commented:

> This is actually intentional. We want the lock manager to be able to grant keys even if the lock is sold out. Note that the lock manager could also increase the supply if they needed anyway. However, we should take that into account in the `shareKey` flow so I'll mark as confirmed for that flow.

[M-13] MALICIOUS USER CAN GET INFINITE FREE TRIAL BY REPEATEDLY REFUND AND REPURCHASE RIGHT BEFORE THE FREETRIAL ENDS

Submitted by WatchPug

The current design/implementation allows users who are refunded before to get another `freeTrial`. This can be exploited by malicious users to get an infinite free trial.

Proof of Concept

Given:

- `keyPrice` is 1 ETH;
- `freeTrialLength` is 31 days.

A malicious user can:

1. Call `purchase()`, pay 1 ETH and get 31 days of `freeTrial` on day 1;
2. Call `cancelAndRefund()` on day 30 and get 1 ETH of refund; then call `purchase()` again, pay 1 ETH and get 31 days of `freeTrial` again.

Repeat the steps above and the user can get infinite `freeTrial`.

Impact

A malicious third party may provide a service named `freeUnlock`, which will call `cancelAndRefund()` and `purchase()` automatically right before the end of the `freeTrial`. This can cause fund loss to all the owners that provide a `freeTrial`.

Recommendation

Consider adding a `mapping(address => uint256) freeTrialEnds` and make sure each address can only get 1 `freeTrial`.

julien51 (Unlock Protocol) disputed and commented:

> Isn’t that the case with every free trial system? If they use the same address the lock manager could easily use the hook system to keep track of who already had received a full refund and not grant it on the 2nd cancellation. The user could still use new addresses all the time, and in that case that would be valid, but that is

> actually the case with a lot of systems like that :) One of my roommates in college was just subscribing to newspaper and getting the full risk-free refund by using a different name every time (but used the same address)

0xleastwood (judge) commented:

> While I agree with the warden, there is potential for unlimited free trials. Limiting a free trial to a single address does not resolve the issue as an attacker can generate any number of addresses from a single seed. However, I do understand this is a tricky issue to workaround.
>
> So I'm not sure how this should be treated as it does affect how the protocol is intended to operate. Is there any reason for users to not abuse this @julien51 ? Typically with newspapers, you have to provide credit card details, so an individual is really limited by the number of cards they hold.

julien51 (Unlock Protocol) commented:

> As you noted, there is *no way* to prevent free trials from being abused which is why by default, locks do *not* have a free trial: they have to be manually explicitly configured. From there, since it's trivial to just create an infinite number of accounts, anyone could just claim free trials over and over from new accounts.

0xleastwood (judge) decreased severity to Medium and commented:

> As per sponsor, trials are not enabled by default. But seeing as this impacts protocol availability through abuse if enabled. I'll mark this as `medium`.

[M-14] MIXINREFUNDS: FRONTRUN UPDATEKEYPRICING() FOR FREE PROFIT

Submitted by itsmeSTYJ

A malicious user is able to withdraw all payments that were paid to a lock owner if the owner increases the keyPrice.

Proof of Concept

When `updateKeyPricing()` is called to increase the price of a key, it is possible to frontrun this call and buy many keys at the cheaper price then request for a refund at the higher price.

Recommendation

Keep track of the price at which keys are purchased so that when you issue a refund, you use the original keyPrice to refund instead of the updated keyPrice

julien51 (Unlock Protocol) acknowledged, but disagreed with High severity and commented:

> This is only true for locks where there is no penalty. We should make it clear on the front-end that when changing the price it is recommended to set up a penalty (at least temporarily) for the price difference so that no key can be refunded for the full price.

0xleastwood (judge) commented:

> Circling back on this, I'm not sure how a penalty would be correctly applied to all locks. Wouldn't users who wanted to get a refund for their key get penalised if they purchase after the change in key price? I think it would also be safer to update the key price and apply the penalty in the one transaction.

julien51 (Unlock Protocol) commented:

> I think that is a good finding, but there again (like often) I think this is pretty edgy. The cancellation penalty is pretty easy to apply just to a single lock from the lock manager's perspective. Before changing the lock price, a lock manager can easily apply a penalty for the difference in price. IE if I change the price from 10 to 12, I apply a penalty for 2 and anyone who tries to abuse this will only get a refund of 12-2 = 10.
>
> On top of that we're actually storing the amount paid for the latest key as part of our next upgrade to support automatically recurring memberships, which should

> make things even more robust as anyone will only get re-imbursed based on what they paid…

0xleastwood (judge) decreased severity to Medium and commented:

> Considering the sponsor's comments and after some further discussion on Discord. I think it is more correct to downgrade this to `medium` severity. While it isn't clear, the lock manager is expected to apply a penalty before updating the cost of a membership such that users cannot game the price difference. However, this isn't enforced on-chain or documented anywhere so based on the judge's and warden's context at the time, this seemed like a valid `high` severity issue. It is important to note that users who refund their membership after purchasing a membership post price change will be refunded less than users who purchased their memberships before the price change. The sponsor is looking to integrate these fixes in their next upgrade.

Low Risk Findings (32)

- [L-01] `Unlock.addLockTemplate` does not adequately increment version, leading to gaps in version *Submitted by elprofesor, also found by loop, pauliax, 0x0x0x, and harleythedog*
- [L-02] Insufficient version validation causes denial of service for `PublicLock` during lock upgrades *Submitted by elprofesor*
- [L-03] MixinGrantKeys:grantKeys possible DoS with (Unexpected) revert *Submitted by GiveMeTestEther*
- [L-04] No ERC20 safeApprove called & not success check *Submitted by cmichel, also found by Reigada and 0x0x0x*
- [L-05] MixinLockCore: use safeApprove from SafeERC20, and do approve(0) before approve(amount) *Submitted by GiveMeTestEther*
- [L-06] getTransferFee() Fee Could Be 0 *Submitted by Meta0xNull*

- [L-07] The function MixinLockCore.approveBeneficiary is susceptible to a race condition *Submitted by Ruhum*
- [L-08] Unlock has incomplete fallback function which may cause loss of funds *Submitted by elprofesor*
- [L-09] `initialize` functions can be frontrun *Submitted by cmichel, also found by WatchPug*
- [L-10] _cancelAndRefund is not protected from re-entrancy *Submitted by pauliax*
- [L-11] setKeyManagerOf has no address-0 check *Submitted by kenzo*
- [L-12] Distribution of tokens in recordKeyPurchase *Submitted by pauliax*
- [L-13] a single user can become owner of multiple token ids *Submitted by GiveMeTestEther*
- [L-14] Setting the admin in initialize initializeProxyAdmin can be frontrun by an attacker *Submitted by Jujic*
- [L-15] Scenario where variable in Unlock.recordKeyPurchase() is not initialized *Submitted by Ruhum*
- [L-16] Consider adding storage gaps to `Mixin***` contracts *Submitted by WatchPug*
- [L-17] Lock template versions can be overwritten *Submitted by cmichel*
- [L-18] Can set arbitrary lock templates *Submitted by cmichel*
- [L-19] DoS when `onKeyPurchaseHook` reverts *Submitted by cmichel*
- [L-20] PREVENT DIV BY 0 *Submitted by defsec*
- [L-21] Confliction on double `initialize` functions front-run `minter` *Submitted by hagrid*
- [L-22] Unimplemented function computeAvailableDiscountFor *Submitted by harleythedog*
- [L-23] Unused function parameters *Submitted by jayjonah8*
- [L-24] Wrong event parameter emitted at _setKeyManagerOf *Submitted by kenzo*
- [L-25] Potential division by 0 in `recordKeyPurchase` *Submitted by loop*

- [L-26] Function spec and implementation difference / strict comparison *Submitted by loop*
- [L-27] msg.value should be 0 when token is not native *Submitted by pauliax*
- [L-28] tokenByIndex returns wrong token id *Submitted by pauliax*
- [L-29] Interface and implementation differ *Submitted by pauliax*
- [L-30] onKeyPurchase hook expects amount + discount *Submitted by pauliax*
- [L-31] Validations *Submitted by pauliax*
- [L-32] `MixinRefunds.sol#cancelAndRefund()` Potential fund loss on `cancelAndRefund()` for users who purchased multiple times *Submitted by WatchPug, also found by GiveMeTestEther*

Non-Critical Findings (54)

- [N-01] Use safeTransfer consistently instead of transfer *Submitted by Jujic*
- [N-02] `freeTrialLength` is used as full refund period *Submitted by 0x0x0x*
- [N-03] Open TODOs *Submitted by mics, also found by Meta0xNull, loop, pauliax, ye0lde, hagrid, and defsec*
- [N-04] Function grantKeys() - Bulk Send Free Keys Are Not Practical & Gas May Over Block Size Limit *Submitted by Meta0xNull*
- [N-05] ERC20 return values not checked *Submitted by cmichel*
- [N-06] Unable to change token approval when tokenAddress changed *Submitted by gzeon*
- [N-07] input validation *Submitted by sabtikw*
- [N-08] Input validation of Zero address on addLockTemplate *Submitted by BouSalman*
- [N-09] Input validation of Zero address on function initialize() *Submitted by BouSalman*
- [N-10] Input validation Zero address *Submitted by BouSalman*

- [N-11] Insufficient input validation *Submitted by WatchPug*
- [N-12] Missing input validation on array lengths (MixinGrantKeys.sol) *Submitted by ye0lde*
- [N-13] transferOwnership should be two step process *Submitted by defsec, also found by Meta0xNull*
- [N-14] USE OF DEPRECATED _SETUPROLE FUNCTION *Submitted by Reigada*
- [N-15] USE OF FLOATING PRAGMA *Submitted by Reigada, also found by jayjonah8*
- [N-16] MixinERC721Enumerable.tokenOfOwnerByIndex - parameter _index can be removed *Submitted by Reigada*
- [N-17] Missing events for critical operations *Submitted by WatchPug*
- [N-18] Initializer modifiers should be called in the same way everywhere *Submitted by jayjonah8*
- [N-19] Wrong comment in recordKeyPurchase *Submitted by kenzo*
- [N-20] Inconsistent code and comment *Submitted by gzeon*
- [N-21] MixinLockCore.sol has wrong comments *Submitted by GiveMeTestEther*
- [N-22] Incorrect or confusing comments or missing code in tokenOfOwnerByIndex *Submitted by ye0lde*
- [N-23] named return issue *Submitted by mics*
- [N-24] safeApprove is deprecated. *Submitted by mics*
- [N-25] Use explicit variables type *Submitted by BouSalman*
- [N-26] Function type from public to external *Submitted by BouSalman*
- [N-27] Missing event for critical updateBeneficiary function *Submitted by BouSalman*
- [N-28] Function type from public to external tokenByIndex() *Submitted by BouSalman*
- [N-29] Fix event params for `KeyManagerChanged` *Submitted by HardlyDifficult*
- [N-30] Reduce rounding error when minting UDT in Unlock *Submitted by HardlyDifficult*
- [N-31] shareKey onERC721Received tokenId *Submitted by HardlyDifficult*
- [N-32] Remove fallback function *Submitted by HardlyDifficult*

- [N-33] Unnecessary function parameter in Unlock.upgradeLock() function *Submitted by Ruhum*
- [N-34] Changes that affect access control should be accompanied by an event *Submitted by Ruhum*
- [N-35] Constants are not explicitly declared *Submitted by WatchPug*
- [N-36] Code Style: Unnecessary public function visibility *Submitted by WatchPug*
- [N-37] Consider adding `initializer` modifier to _initialize** functions *Submitted by WatchPug*
- [N-38] Incomplete implementation *Submitted by WatchPug*
- [N-39] `MixinPurchase#purchase()` Consider checking if _referrer equals _recipient *Submitted by WatchPug*
- [N-40] Race condition on ERC20 approval *Submitted by WatchPug*
- [N-41] Typos *Submitted by WatchPug*
- [N-42] Critical changes should use two-step procedure *Submitted by WatchPug*
- [N-43] Order of layout is wrong in ERC20Patched.sol *Submitted by aga7hokakological*
- [N-44] Order of function is wrong in contracts ERC20PermitUpgradeable, ERC20VotesCompUpgradeable, EIP712Upgradeable *Submitted by aga7hokakological*
- [N-45] Missing `_beforeTokenTransfer` Token Transfer Handle *Submitted by hagrid*
- [N-46] Initialization parameters of new lock template are hardcoded *Submitted by kenzo*
- [N-47] Unconventional log emittance confuses Etherscan *Submitted by kenzo*
- [N-48] Commented lines of code *Submitted by loop*
- [N-49] Use of access control require statement when modifier exists *Submitted by loop*
- [N-50] grantKeys no check on parameter array lengths and values *Submitted by nathaniel*
- [N-51] Store owners in EnumerableSet *Submitted by pauliax*
- [N-52] `== true` doesn’t bring anything *Submitted by 0x0x0x*

- [N-53] Avoiding Initialization of Loop Index If It Is 0 *Submitted by Meta0xNull*
- [N-54] Upgrade pragma to at least 0.8.4 *Submitted by defsec*

Gas Optimizations (48)

- [G-01] MixinTransfer:getTransferFee gas optimization with unchecked *Submitted by GiveMeTestEther*
- [G-02] Setters of `UnlockProtocolGovernor.sol` can be implemented more efficiently *Submitted by 0x0x0x*
- [G-03] Cache length at for loop to save gas *Submitted by 0x0x0x*
- [G-04] `MixinGrantKeys.sol` apply requiere statements earlier *Submitted by 0x0x0x*
- [G-05] Use unchecked operation to save gas *Submitted by 0x0x0x*
- [G-06] 4 variables are cached and used only once at `Unlock.sol#upgradeLock` *Submitted by 0x0x0x*
- [G-07] `UnlockUtils.sol#address2Str` can be implemented much cheaper *Submitted by 0x0x0x*
- [G-08] Unnecessary fallback function *Submitted by BouSalman*
- [G-09] Adding unchecked directive can save gas *Submitted by WatchPug, also found by GiveMeTestEther, mics, and pauliax*
- [G-10] MixinGrantKeys:grantKeys gas optimizations *Submitted by GiveMeTestEther, also found by Reigada and defsec*
- [G-11] Unlock:createLock no need to define the newLock as payable *Submitted by GiveMeTestEther, also found by WatchPug*
- [G-12] Unlock:_deployProxyAdmin return value is not used *Submitted by GiveMeTestEther*
- [G-13] MixinFunds:_initializeMixinFunds move the require statement to the beginning of the function so save gas in the case of a revert *Submitted by GiveMeTestEther, also found by loop*

- [G-14] Gas: Assume 0 when creating struct *Submitted by HardlyDifficult*
- [G-15] Gas: Cast instead of creating new variables *Submitted by HardlyDifficult, also found by TomFrench and cmichel*
- [G-16] Avoid On Chain Computation That Have Known Answer to Save Gas *Submitted by Meta0xNull*
- [G-17] Long Revert Strings *Submitted by Reigada, also found by WatchPug*
- [G-18] Using uint16 for lock versions increases gas costs for no reason. *Submitted by TomFrench*
- [G-19] Unused named returns *Submitted by WatchPug*
- [G-20] `UnlockUtils.sol#uint2Str()` Implementation can be simpler and save some gas *Submitted by WatchPug*
- [G-21] `MixinLockCore.sol#updateKeyPricing()` Check of `_tokenAddress` can be done earlier to save gas *Submitted by WatchPug*
- [G-22] Remove unnecessary variables can make the code simpler and save some gas *Submitted by WatchPug*
- [G-23] Redundant check of `owner() != address(0)` *Submitted by WatchPug*
- [G-24] Changing function visibility from public to external can save gas *Submitted by WatchPug, also found by loop, mics, nathaniel, and Jujic*
- [G-25] Avoid unnecessary storage reads can save gas *Submitted by WatchPug*
- [G-26] Remove unnecessary function can make the code simpler and save some gas *Submitted by WatchPug*
- [G-27] `MixinRefunds.sol#_getCancelAndRefundValue` Cache and read storage variables from the stack can save gas *Submitted by WatchPug*
- [G-28] Gas: `_recordOwner` pushes duplicates *Submitted by cmichel*
- [G-29] Inconsistent use of _msgSender() *Submitted by defsec*
- [G-30] Gas improvement on the nonce increment *Submitted by defsec*
- [G-31] Less than 256 uints are not gas efficient *Submitted by defsec*
- [G-32] Gas optimization: Unused variable `yieldedDiscountTokens` *Submitted by gzeon*

- [G-33] MixinRefunds: use variable to save gas *Submitted by itsmeSTYJ*
- [G-34] MixinPurchase: gas optimisation by relying on 0.8.0 auto revert on underflow. *Submitted by itsmeSTYJ*
- [G-35] Redundant check of freeTrialLength == 0 *Submitted by nathaniel*
- [G-36] Precalculate expressions *Submitted by pauliax*
- [G-37] Unnecessary checks *Submitted by pauliax*
- [G-38] Refund amount and penalty calculation *Submitted by pauliax*
- [G-39] timePlusFee = timeRemaining *Submitted by pauliax*
- [G-40] assigned operations to constant variables *Submitted by pauliax, also found by Reigada*
- [G-41] 0 valueInETH *Submitted by pauliax*
- [G-42] ++/— are cheapest *Submitted by pauliax*
- [G-43] address(this).address2Str() *Submitted by pauliax*
- [G-44] Use existing memory version of state variables *Submitted by ye0lde*
- [G-45] Unused Named Returns *Submitted by ye0lde*
- [G-46] Gas: remove owners array *Submitted by HardlyDifficult*
- [G-47] `Unlock.sol#RecordKeyPurchases` can be implemented cheaper *Submitted by 0x0x0x*
- [G-48] Gas: Merge callbacks to Unlock on purchase *Submitted by HardlyDifficult*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

TWITTER // DISCORD // GITHUB

0XC2BC2F890067C511215F9463A064221577A53E10 //