byterocket

Smart Contract Audit Report

Unlock Protocol

11th of February 2022

Contents

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

1. Preface

The developers of **Unlock Protocol** contracted byterocket to conduct a smart contract audit of their smart contracts. Unlock Protocol is "*an access control protocol built on a blockchain. It enables creators to monetize their content or software without relying on a middleman. It lets consumers manage all of their subscriptions in a consistent way, as well as earn discounts when they share the best content and applications they use.*"

The team of byterocket reviewed and audited the above smart contracts in the course of this audit. We started on the 31st of January and finished on the 11th of February 2022.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Protocol/Logic Analysis*
- *Automated Code Review*
- *Formal Report*

byterocket gained access to the code via a public GitHub repository. The developers froze the code in a separate branch for us. We based the audit on our auditing branch state, deployed on January 28th, 2022 (*commit hash bfb4d59e5a4c5490fa8d dc9a7d235691de64da62*).

2. Manual Code Review

We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Three different people went through the smart contract independently and compared their results in multiple concluding discussions.

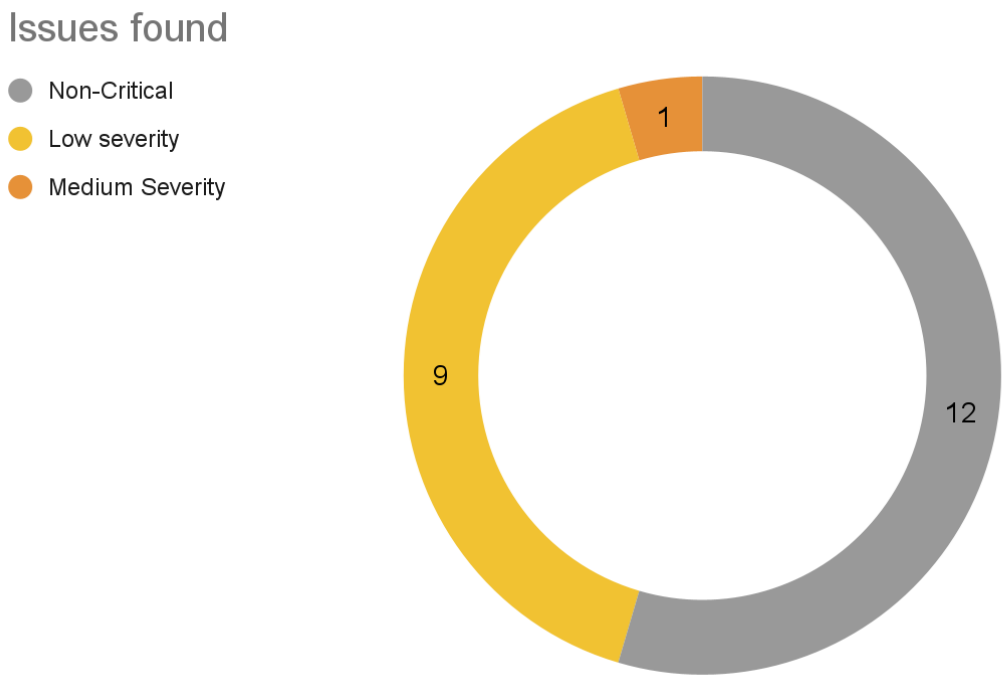
The manual review and analysis were additionally supported by multiple automated reviewing tools, like Slither, GasGauge, Manticore, and different fuzzing tools.

2.1 Severity Categories

We are categorizing our findings into four different levels of severity:

Non-Critical	Does not impose immediate risk but is relevant to security best practices. Includes issues with - Code style and clarity - Versioning - Off-chain monitoring
Low Severity	Imposes relatively small risks or could impose risks in the long-term but without assets being at risk in the current implementation. Includes issues with - State handling - Functions being incorrect as to specification - Faulty documentation or in-code comments
Medium Severity	Imposes risks on the function or availability of the protocol or imposes financial risk by leaking value from the protocol if external requirements are met.
High Severity	Imposes catastrophic risk for users and/or the protocol. Includes issues that could result in - Assets being stolen/lost/compromised - Contracts being rendered useless - Contracts being gained control of

2.2 Summary



On the code level, we **found 23 bugs or flaws, with** 12 non-critical**,** 9 of low severity**, and** 1 of medium severity.

The contracts are mostly written according to the latest standard used within the Ethereum community and the Solidity community's best practices, with some exceptions listed below. The naming of variables is very logical and understandable, which results in the contract being useful to understand. The code is very well documented, except for some occurrences listed below. The developers provided us with a test suite as well as deployment scripts.

It is, however, noticeable that different people worked on these contracts without a style guide or best practices that they agreed upon. There are parts of the code that look very different from others. This has no security implications, we are just noting it.

2.3 Findings

[MEDIUM SEVERITY] M.1 - Dangerous Management of Key Manager

Location: MixinPurchase.sol - Line 116

Description:
A user can leverage the purchase function to extend another user's key. While doing so, they can set themselves (or any address for that matter) as the key manager. This should not be possible.

These are the affected lines of code:

```
_setKeyManagerOf(idTo, _keyManager);
```

Recommendation:
Consider wrapping the `_setKeyManager()` call in line 116 in an if-clause that verifies that the caller (`msg.sender`) is the `_recipient`.

[LOW SEVERITY] L.1 - Gas refund value can be higher than price

Location: MixinPurchase.sol - Line 34 - 36 & 151 - 160

Description:
The user can set the gas refund value to an arbitrary value. Hence, it is possible to set the gas refund to very high amounts, even higher than the actual value of the key. The user can lose money with every sale because of this.

These are the affected lines of code:

```
function setGasRefundValue(uint256 _refundValue) external
         onlyLockManager {
  _gasRefundValue = _refundValue;
}
```

Recommendation:
With values that can be changed due to user input, we suggest adding safety and sanity checks. In this case, we would suggest ensuring that the gas refund value is lower than or equal to the actual key price.

[LOW SEVERITY] L.2 - Setting a new template doesn't validate the array

Location: Unlock.sol - Line 476 - 491

Description:
When the owner sets a new lock template for the subsequent calls to `createLock()`, the implementation address for this is being defined. It is not, however, verified whether this address is the current or latest address of the `_publicLockImpls` array. It's not even verified whether it is part of the array at all.

These are the affected lines of code:

```
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:
Validate that the newly provided address is already part of the `_publicLockImpls` array, or add it to the array at that moment if that is not the case.

[LOW SEVERITY] L.3 - Adding a new lock template doesn't remove a faulty one

Location: Unlock.sol - Line 187 - 193

Description:
When a faulty implementation has been added, the version number could later be reduced to jump back to the latest working version. In this case, the highest version number of the faulty implementation still exists.

These are the affected lines of code:

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

```
  if (publicLockLatestVersion < version)
     publicLockLatestVersion = version;
  emit UnlockTemplateAdded(impl, version);
}
```

Recommendation:
Consider removing any versions that are higher than what is currently being added.

[LOW SEVERITY] L.4 - Users can upgrade to a lock version that is "too high"

Location: Unlock.sol - Line 187 - 193

Description:
The `upgradeLock()` function does not verify whether the version that the user upgrades their lock to is the current latest version stored in `publicLockLatestVersion`. Due to the finding in L.3, it would be possible to still have faulty implementations in the array of implementations.

These are the affected lines of code:

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

Recommendation:
Consider checking whether the version that a user upgrades to is safe. This can be done by fixing L.3 or adding another code section that addresses this somehow.

[LOW SEVERITY] L.5 - Deprecated function delivers unexpected returns

Location: Unlock.sol - Line 208 - 228

Description:
The legacy function createLock() disregards the `_salt` parameter since it has been deprecated. This leads to the fact that the output of this function is not deterministic anymore, which old implementations could rely on.

These are the affected lines of code:

```
function createLock(
  uint _expirationDuration,
  address _tokenAddress,
  uint _keyPrice,
  uint _maxNumberOfKeys,
  string calldata _lockName,
  bytes12 // _salt
) public returns(address) {
```

Recommendation:
We would think that this function should just revert since otherwise unaware developers could run into issues here due to the now non-deterministic nature of the implementation of the function.

[LOW SEVERITY] L.6 - Unextendable key after the duration becomes infinite

Location: MixinPurchase.sol - Line 100 - 103

Description:
If the manager of the key sets a key with a previously not-infinite expiration date to an expiration date with an infinite value, the user won't be able to extend the key while it's not expired. It's valid to check if the expiration timestamp of a key is infinite to prohibit unnecessary extensions, but users might run into issues here due to this edge case.

These are the affected lines of code:

```
require(toKey.expirationTimestamp != type(uint).max, 'A valid
        non-expiring key can not be purchased twice');
[...]
newTimeStamp = toKey.expirationTimestamp + expirationDuration;
```

Recommendation:
Allow users that have a timestamp that is running out to extend their purchased lock to also be of infinite length.

[LOW SEVERITY] L.7 - Data Argument has overloaded Use-Cases

Location: MixinPurchase.sol - Line 58 - 161

Description:
The `_data` argument given by the user in the `purchase()` function is used for the `onKeyPurchases`-Hook and the `keyPurchasePrice`-Hook. This makes it impossible for a lock to use both hooks, as the data argument would be overloaded with use-cases. Additionally, the functions support inputs of different lengths.

These are the affected lines of code:

```
uint inMemoryKeyPrice = _purchasePriceFor(_recipient, _referrer, _data);
[...]
onKeyPurchaseHook.onKeyPurchase(msg.sender, _recipient, _referrer,
_data, inMemoryKeyPrice, pricePaid);
```

Recommendation:
Consider adding one argument per hook to separate concerns.

[LOW SEVERITY] L.8 - FeeOnTransfer tokens not supported

Location: Throughout the project

Description:
In the creation of a lock, a user can set the token address used for payments. This includes FeeOnTransfer tokens, however, the purchase function in `MixinPurchase` does not handle these correctly.

These are the affected lines of code:

```
uint inMemoryKeyPrice = _purchasePriceFor(_recipient, _referrer, _data);
[...]
onKeyPurchaseHook.onKeyPurchase(msg.sender, _recipient, _referrer,
_data, inMemoryKeyPrice, pricePaid);
```

Recommendation:
Consider adding either a **big warning** stating that these tokens are not supported (yet) or disabling payments with these tokens by using an if check, verifying the transferred balances before and after the calls.

[LOW SEVERITY] L.9 - Users can create uninitialized locks

Location: Unlock.sol - Line 246 - 265

Description:
Users can create uninitialized locks, as the `data` argument is provided by the user. Such an initialized lock would still be added to the `locks` array and from there on taken as initialized. A user would be able to create such an uninitialized lock by encoding a call to the fallback function of the contract or any other function that executes without a revert.

These are the affected lines of code:

```
TransparentUpgradeableProxy proxy = new
```

```
TransparentUpgradeableProxy(publicLockAddress, proxyAdminAddress, data);
```

Recommendation:
Consider verifying whether the created lock is actually initialized. This could for example be done by wrapping a call to the initialize function in a try-catch after it's initialization.

[NON-CRITICAL] **NC.1 - Using SafeMath with Solidity >= 0.8.0**
Location: DiscountCodeHook.sol - Line 17

Description:
The smart contract uses `SafeMath` while at the same time being implemented in Solidity version 0.8.2. Solidity has had an inbuilt overflow handling since version 0.8.0. The corresponding calls in lines 73 & 74 below are not unsafe but lead to more gas spendings than necessary.

These are the affected lines of code:

```
uint discount = minKeyPrice.mul(discountBP).div(10000);
minKeyPrice = minKeyPrice.sub(discount);
```

Recommendation:
Remove the `SafeMath` library and replace the `mul` and `sub` calls with the regular arithmetic calls.

[NON-CRITICAL] **NC.2 - Emitting an event twice**
Location: MixinGrantKeys.sol - Line 47 - 48

Description:
The function `_setKeyManagerOf()` is called in line 47, which in itself emits the event `KeyManagerChanged()`. This event, however, is additionally emitted after this call in line 48. Furthermore, if the key's manager is not changed in `_setKeyManagerOf()`, the event would be erroneously emitted anyway.

These are the affected lines of code:

```
emit KeyManagerChanged(idTo, keyManager);
```

Recommendation:
Remove the emitted event in line 48, since it is unnecessary.

[NON-CRITICAL] NC.3 - Lock name can be updated to empty string

Location: MixinLockMetadata.sol - Line 54 - 60

Description:
The updateLockName() function does not check whether the _lockName parameter is an empty string. Henceforth, the lock can be updated to have no name.

These are the affected lines of code:

```
function updateLockName(string calldata _lockName) external
         onlyLockManager {
  name = _lockName;
}
```

Recommendation:
It is usually best practice to prevent users from doing something that shouldn't be done. We would argue that there shouldn't be a lock that has no name.

[NON-CRITICAL] NC.4 - Documentation is wrong

Location: MixinLockMetadata.sol - Line 27 - 28

Description:
The documentation notes that the lock name defaults to a certain value, which is not the case.
These are the affected lines of code:

```
/// A descriptive name for a collection of NFTs in this contract.
    Defaults to 'Unlock-Protocol' but is settable by lock owner
string public name;
```

Recommendation:
Either add a function that does provide a default value for the name or adapt the documentation to what's actually the case.

[NON-CRITICAL] NC.5 - Use of deprecated library functions

Location: MixinRoles.sol - Line 30 - 35

Description:
The `_initializeMixinRoles()` function is using the `_setupRole()` function to set up roles. This function has been deprecated, as referenced here. The new function is called `_grantRole()`.

These are the affected lines of code:

```
_setupRole(LOCK_MANAGER_ROLE, sender);
```

Recommendation:
While `_setupRole()` internally just calls `_grantRole()`, it just makes sense to directly call `_grantRole()`.

[NON-CRITICAL] **NC.6 - Inconsistent error messages**
Location: Throughout the project

Description:
The smart contracts are using different, inconsistent error messages throughout the project. Additionally, some of the error messages are quite long.

These are some affected lines of code:

```
“MixinRoles: caller does not have the LockManager role”
“INVALID_ADDRESS”
“Transfer to xDAI disabled”
“TRANSFER_FROM: NOT_KEY_OWNER”
```

Recommendation:
Decide on one format for the error messages and implement it throughout the project if possible. Consider shortening revert strings to fit in 32 bytes, since this will decrease gas costs for deployment and gas costs when the revert condition has been met.

[NON-CRITICAL] **NC.7 - Reference link is off**
Location: UnlockUtils.sol - Line 6

Description:
The provided link where the implementation is borrowed from is not directed to the right code section anymore.

https://github.com/oraclize/ethereum-api/blob/master/oraclizeAPI_0.5.sol#L943

Recommendation:
Update the link, and if possible to one that does not reference the ever-changing master branch.

[NON-CRITICAL] NC.8 - Excessive gas usage in edge case

Location: Unlock.sol - Line 345

Description:
In the edge case of the oracle being less than 24 hours old after its deployment, it will always return zero, hence the `valueInETH` will be zero. In this edge case, all of the following code can be skipped, since it doesn't do anything meaningful.

These are the affected lines of code:

```
valueInETH = oracle.updateAndConsult(tokenAddress, _value, weth);
```

Recommendation:
If there is ever a possibility of the oracle being less than 24 hours old, we would suggest catching this behavior.

[NON-CRITICAL] NC.9 - BaseFee is hardcoded

Location: Unlock.sol - Line 371

Description:
For chains that have no `baseFee` implemented yet, it is hardcoded to 100. The nature and parameters of current blockchains are changing rapidly, so it might become necessary for this to change at some point.

These are the affected lines of code:

```
try this.networkBaseFee() returns (uint _basefee) {
  // no assigned value
  if(_basefee == 0) {
    baseFee = 100;
  } else {
    baseFee = _basefee;
  }
} catch {
  // block.basefee not supported
  baseFee = 100;
}
```

Recommendation:
Allow the contract owner to update the `baseFee` if necessary.

[NON-CRITICAL] NC.10 - Use of constant instead of immutable

Location: MixinRoles.sol - Line 12 - 13

Description:
Access roles marked as `constant` result in computing the `keccak256` operation each time the variable is used because assigned operations for constant variables are re-evaluated every time.

These are the affected lines of code:

```
bytes32 public constant LOCK_MANAGER_ROLE = keccak256("LOCK_MANAGER");
bytes32 public constant KEY_GRANTER_ROLE = keccak256("KEY_GRANTER");
```

Recommendation:
Changing the variables to `immutable` results in computing the hash only once on deployment, leading to gas savings.

[NON-CRITICAL] NC.11 - Wrong function name

Location: Unlock.sol - Line 511

Description:
The name of the function does not reflect the documentation as well as what the function actually does. It does not reset the values, it allows the owner to set them to arbitrary values.

These are the affected lines of code:

```
// Allows the owner to change the value tracking variables as needed.
function resetTrackedValue(
  uint _grossNetworkProduct,
  uint _totalDiscountGranted
) external onlyOwner {
  grossNetworkProduct = _grossNetworkProduct;
  totalDiscountGranted = _totalDiscountGranted;
  emit ResetTrackedValue(_grossNetworkProduct, _totalDiscountGranted);
}
```

Recommendation:
Consider changing the function name in future updates/deployments.

[NON-CRITICAL] **NC.12 - Unnecessary naming of return values**

Location: MixinPurchase.sol - Line 41, UnlockUtils.sol - Line 16 & 24

Description:

There are multiple occurrences where return variables are named but not used with their names. This allocates an unused storage slot during the execution, which is unnecessary.

Recommendation:

Consider removing the name of the return values or use them accordingly.

3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. The byterocket team went through the implementation and documentation of the implemented protocol.

The repository itself contained tests and documentation. Even if there was a crucial part of the documentation missing at the time of our audit (a necessary `yarn build` in the root folder), we found the documentation to be very helpful. The online documentation, as well as the inline documentation, is sufficient to fully understand the intended protocol that is being implemented.

We found the provided unit tests that are coming with the repository execute without any issues and cover the most important parts of the protocol.

According to our analysis, the protocol and logic are working as intended, given that the findings listed in section (2) with the severity of low or medium are fixed.

We were **not able to discover any additional problems** in the protocol implemented in the smart contract.

4. Summary

During our code review (*which was done manually and automated*), we found **23 bugs or flaws, with** 12 non-critical**,** 9 of low severity**, and** 1 of medium severity. Our automated systems and review tools did **not find any additional ones**.

The protocol review and analysis did neither uncover any game-theoretical nature problems nor any other functions prone to abuse.

In general, there are some improvements that can be made, but we are **very happy** with the overall quality of the code and its documentation. The developers have been very responsive and were able to answer any questions that we had.