Least Authority

PRIVACY MATTERS

xx coin

Final Security Audit Report (v2)

# Praxxis.io

Report Version: 3 February 2020

# Table of Contents

# Overview

## Background

Praxxis.io has requested that Least Authority perform a security audit of the xx coin, the xx network's smart contract token ERC 1404 structure, in preparation for the token sale launch

The ERC 1404 standard is developed by Tokensoft to manage regulatory compliance of the tokens via an explicit whitelist for sending and receiving the tokens, which will be traded for native tokens on the xx network via an intermediate smart contract that will be developed following the token sale.

## Project Dates

- January 16 - 20: Initial Review *(Completed)*
- January 21: Initial Audit Report delivered *(Completed)*
- January 22: Verification Review *(Scheduled)*
- January 23: Final Audit Report delivered *(Scheduled)*

## Review Team

- Emery Rose Hall, Security Researcher and Engineer
- Alexander Leitner, Security Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer

# Coverage

## Target Code and Revision

For this audit, we performed research, investigation, and review of the xx coin followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- xx coin: https://gitlab.com/praxxisio/xx-coin/tree/batch

Specifically, we examined the Git revisions for our initial review:

```
718dad5456e1156494d06a86342be595fba439b8
```

For the verification, we examined the Git revision:

```
18f93c716ac153b920b4f703de4b91932e653420
```

All file references in this document use Unix-style paths relative to the project's root directory.

## Supporting Documentation

The following documentation was available to the review team:

- Patches.txt (Received via email on 14 January 2020)
  - Code changes made by Praxxis to ERC-1404 prior to initial commit to xx coin repository
- xx coin ERC1404 Technical Summary: https://docs.google.com/document/d/1vZG-XzcnUD6URasi3PXhkwKHO3bml-bxwYEbBgzokZE/edit?ts=5e2217fb#heading=h.5ot291g76s3n
- Test Scripts: https://drive.google.com/drive/folders/1k94RXut9aHlPqY6O1NYixQEBXhuqYuDr
- Sale Script: https://drive.google.com/drive/folders/1ekdLZyfeOis-DuM8yZ0UmfJ8hw4NbvYk

## Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network related to the token sale;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Economic incentives: ensure token economics (monetary incentives to punish bad behavior and reward good behavior) are included and functional;
- DoS/security exploits that would impact the contracts intended use or disrupt the execution of the contract;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

# Findings

## General Comments

Overall, the contracts appear to function as intended. We did identify a few areas in the sections below where improvements to comprehensibility can be made.

However, while the additions which were made to the preexisting open source code were minimal, we found that the manner in which these components were assembled yielded a problematic level of complexity with regard to the inheritance chain - although this has been resolved since the delivery of our initial report. This made review difficult and, in some cases, somewhat confusing. We also found there to be a lack of supporting documentation and absence of a test suite, for which we have outlined details in the specific issues below.

In addition, upon delivery of our initial report, the Praxxis.io team acknowledged several of the reported issues, but state that they will not be resolved prior to deployment. Their response is outlined in the status section of the detailed issues and suggestions below. Although many of these issues are generally low impact or edge cases, we strongly recommend that they be addressed as early as possible.

## Specific Issues

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

| ISSUE / SUGGESTION | STATUS |
|---|---|
| Issue A: Test Suite is Completely Absent | Unresolved |
| Issue B: Inheritance Chain is Unnecessarily Complex | Resolved |
| Issue C: `transferOwnership` Function Does Not Handle Incorrect Address | Unresolved |

| | |
|---|---|
| Issue D: Gnosis MultiSig Wallet Does Not Check `ReplaceOwner` `NotNull` | Unresolved |
| Suggestion 1 Use Positive Truth Checks to Avoid Confusion | Unresolved |
| Suggestion 2: Batch Functions Always Return `False`; Return Type is Unnecessary | Resolved |
| Suggestion 3: Solidity Version | Resolved |
| Suggestion 4: Unnecessary Require Statement | Resolved |
| Suggestion 5: Batch Send Cost Expectations | Unresolved |
| Suggestion 6: Remove *hotWallet* From Whitelist After Deployment | Invalid |

## Issue A: Test Suite is Completely Absent

**Synopsis**

While a truffle configuration file does exist implying that a test suite was considered, there is no testing code to confirm that the contract functions as expected.

**Impact**

High. Given that the inheritance chain is very complex (see Issue B) and that the deployed contract could ultimately hold very high monetary value, it is of great importance that the contract's methods are thoroughly tested to ensure that the contract behaves as designed.

**Remediation**

Author a test suite using the tooling already declared in the repository.

**Status**

The Praxxis.io team acknowledged the need for an automated test suite, but decided that authoring one before taking the contracts to production was not feasible for the current release timeline. The team stated that they intend to mitigate this issue with a manual test suite they have run on the current code base, and will implement an automated test suite in future versions of the contract using the manual test suite as a guide.

**Verification**

Unresolved.

## Issue B: Inheritance Chain is Unnecessarily Complex

**Synopsis**

The xx coin contract depends on a complex, multipath inheritance chain where `ERC20Detailed` is inherited at multiple layers of the chain. Additionally, it appears that some parts the inheritance chain, such as `SimpleRestrictedToken#detectTransferRestriction`, seem to only exist for the purpose of satisfying an interface and is overwritten higher up the inheritance path.

**Impact**

Unknown. While the contract compiles with no issues and appears to function correctly, this complexity makes it difficult to follow the code and reason about which contracts and methods take precedence and how Solidity's C3 linearization-style inheritance model handles the inheritance graph.

**Technical Details**

The inheritance graph appears as such, with each level of indentation indicating another layer down the graph.

```
XXCoin
  ERC20Detailed
      IERC20
  ManagedWhitelistToken
      MessagedERC1404
            SimpleRestrictedToken
                  ERC1404
                  ERC20
                              IERC20
                  ERC20Detailed
                        IERC20
      ManagedWhitelist
            Managed
```

As shown, `ERC20Detailed` appears in the chain more than once, while `ERC20` also appears - which inherits the same `IERC20` interface as `ERC20Detailed`. While the compiler does not throw an error and this is legal, it makes it more difficult to understand and follow. In addition, a simple change in the order of inheritance declarations will yield entirely different results by changing the override order.

**Remediation**

Flatten the contract and reorder the dependency chain by consolidating code and removing unnecessary dependencies. ManagedWhitelistToken already conforms to the `IERC20` interface through both `ERC1404` and `ERC20Detailed`. Reworking `ERC1404` to inherit `ERC20Detailed`, removing the dependency on `ERC20` directly, removing the redundant dependency on `ERC20Detailed` from xx coin, then consolidating the dependency path from `ManagedWhitelistToken` down to `ERC1404` would create a single path linear inheritance graph that is far less complex and still provides the same functionality.

Such an inheritance graph should look like the one below.

```
XXCoin
      ManagedWhitelistToken
```

```
                  ManagedWhitelist

                        Managed

            MessagedERC1404 (+SimpleRestrictedToken and ERC1404)

            ERC20Detailed

                        IERC20
```

**Status**

The inheritance graph was significantly simplified as shown below:

```
      XXCoin
            ManagedWhitelist
                  Managed
            MessagedERC1404
                  MessagesAndCodes (Library)
                  SimpleRestrictedToken
                        ERC1404
                              ERC20
                                    IERC20
            ERC20Detailed
                  IERC20
```

**Verification**

Resolved.

## Issue C: `transferOwnership` Function Does Not Handle Incorrect Address

**Location**

[https://github.com/OpenZeppelin/openzeppelin-contracts/blob/6f8e672f3fcb93289fb559ecbef72b8fd1cd56e1/contracts/ownership/Ownable.sol#L61-L76](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/6f8e672f3fcb93289fb559ecbef72b8fd1cd56e1/contracts/ownership/Ownable.sol#L61-L76)

**Synopsis**

A safeguard against incorrect addresses supplied to the `transferOwnership` function does not exist in any ERC20 standard token.

**Impact**

If the admin is changed to an invalid or unknown address, the contract becomes essentially useless. The likelihood of this mistake is minimal but the impact is severe.

**Technical Details**

The `transferOwnership` function does not double check the validity of the new admin. If a typo or uncontrolled address is inserted, the contract is permanently lost. Extended features are gas-expensive and are prone to error as well (specifically checking the data length of provided address).

### Remediation

Consider deviating from the ERC20 standard and creating a two step update process that would propose an update to the ownership of the contract. After that transaction is confirmed, the token could require the proposed address to accept the new ownership. This would catch any updates to incorrect or malformed state.

### Status

The Praxxis.io team acknowledged the issue but believes it may not be valid because it deviates from the ERC20 standard. They intend to mitigate the issue by planning to transfer ownership once to their MultiSig contract, verify correctness, and then never call that function for the duration of the validity of the contract. Some Praxxis.io members believe that this is intentional in the ERC20 standard so that the ability to perform ownership functions to the contract are permanently disabled. While it is possible to "burn" the ownership functionality by publicly displaying transfer to an uncontrollable address, treating this as a feature, it is not to our knowledge a standard way of using the ownership extension to the ERC20 standard.

### Verification

Unresolved.

## Issue D: Gnosis MultiSig Wallet Does Not Check `ReplaceOwner` `NotNull`

### Location

https://github.com/gnosis/MultiSigWallet/blob/864911d31e55d899b55a6d7084432fa0a3efeb2b/dapp/src/services/Wallet.js#L887-L911

### Synopsis

Similar to Issue C, the multisig wallet intended to be used in this project does not check for a null address provided to updating the owners of the wallet contract.

### Impact

If an owner address is applied as a null address, the wallet will no longer be able to send transactions and anything it owns will be locked.

### Technical Details

The xx coin intends to use the Gnosis multisig wallet. While this wallet provides a `notNull` modifier which checks that an incoming new owner address is not `0x0`, it is not applied to the `replaceOwner()` function. This could lead to the case where a null address is updated into the contract.

### Remediation

Consider adding the `notNull()` modifier to the `replaceOwner()` function and testing this functionality.

### Status

The Praxxis.io team acknowledged the issue constituted an edge case that is valid, but has decided not to address it because this code was initially deployed and managed by a 3rd party (TokenSoft). The team states they do not intend to use this functionality and therefore will never trigger the stated scenario. If it becomes necessary to use it, the team stated that they intend to mitigate this issue procedurally, first by ensuring mistakes can always be reversed by verifying enough multisig participants are available to reverse any change and secondly by having multiple individuals verify the validity of the new address. The team indicated that it may be addressed at a future point in time.

#### Verification

Unresolved.

## Suggestion 1: Use Positive Truth Checks to Avoid Confusion

#### Location

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L11

#### Synopsis

The modifier `notRestricted` checks the address in the whitelist and returns true if the address is present (by calling `detectTransferRestriction`). However using a check for **not** restricted (as in **is present** in the whitelist) can be ambiguous and confusing.

#### Remediation

Change the modifier to `isRestricted`, change the logic to return if the address **is not** in the whitelist, then where used return early else proceed with the function instead of wrapping the function body within the truth check.

#### Status

The Praxxis.io team agreed with the suggested practice, but decided against the suggested modifications stating it does not change behavior and would have modified pre-existing vetted code.

#### Verification

Unresolved.

## Suggestion 2: Batch Functions Always Return `False`; Return Type is Unnecessary

#### Location

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L45

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L68

#### Synopsis

`transferFromBatch` and `transferBatch` are expected to return a boolean value but default to always be `false`.

#### Remediation

Remove the return type declaration so that functions completing successfully do not confusingly return `false`.

#### Status

The return types and return values were removed from the functions in question.

#### Verification

Resolved.

## Suggestion 3: Solidity Version

**Synopsis**

The lowest compiler version that can be used is 0.5.0.

**Remediation**

We suggest updating to a more recent compiler version with bug fixes and testing the behavior of the contract against this version. Update contracts to - pragma solidity ^0.5.15.

**Status**

The compiler version was upgraded to ^0.5.15.

**Verification**

Resolved.

## Suggestion 4: Unnecessary Require Statement

**Location**

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L54

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L77

**Synopsis**

In the batch transfer functions, `require()` is wrapped around each `transfer()` attempt. ERC20 already has `requires` that will revert if failure occurs so this should not be necessary and could cost an additional 23 gas per loop.

**Remediation**

Remove the `require()` calls and return true if no failures occur to revert.

**Status**

The unnecessary `require()`s were removed.

**Verification**

Resolved.

## Suggestion 5: Batch Send Cost Expectations

**Location**

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L43-56

https://gitlab.com/praxxisio/xx-coin/blob/batch/contracts/ERC1404/SimpleRestrictedToken.sol#L66-79

**Synopsis**

The deploy documentation does a cost analysis of batching based on an average gas price of 4 Gwei. While this price may be acceptable during low traffic times for a single transaction of a small size, it is likely that a higher price will be needed to include a large batch send. Given that large sends are taking more of the block gas, a priority needs to be placed on the miners so that this transaction is included first and pushes all single transactions on to the next block.

The implementation for batch transfer of tokens also uses the ERC20 standard transfer implemented by OpenZeppelin. This send does requirement checks and safe math features that make it a more expensive send to use in a loop.

**Remediation**

Consider the extra cost needed to prioritize the batch transfers. Also consider using the cheaper batch method provided by ChainSafe.

```
function sendBatchCS(address[] _recipients, uint[] _values) external
returns (bool) {

  require(_recipients.length == _values.length);

  uint senderBalance = balances[msg.sender];

  for (uint i = 0; i < _values.length; i++) {

    uint value = _values[i];

    address to = _recipients[i];

    require(senderBalance >= value);

    if(msg.sender != _recipients[i]){

      senderBalance = senderBalance - value;

      balances[to] += value;

    }

    emit Transfer(msg.sender, to, value);

  }

  balances[msg.sender] = senderBalance;

  return true;

}
```

**Status**

The Praxxis.io team acknowledged the suggestion, but are comfortable with the existing cost estimation and stated that they  do not want to bypass security controls provided by pre-existing vetted code. It should be noted that larger transactions yield higher gas cost, thus, we recommend prioritizing larger transactions.

**Verification**

Unresolved.

### Suggestion 6: Remove *hotWallet* From Whitelist After Deploy

**Location**

xx coin ERC1404 Technical Summary:
https://docs.google.com/document/d/1vZG-XzcnUD6URasi3PXhkwKHO3bml-bxwYEbBgzokZE/edit?ts=5e2217fb#

**Synopsis**

Deploy method D is utilized. The *hotWallet* must be the original wallet that deployed the contract for step 5 to work. At the end of deployment, the *hotWallet* will still be whitelisted.

**Impact**

It may not be intended for the *hotWallet* to remain in the whitelist of addresses that are allowed to send or receive tokens after deployment. If this wallet is used in the future, it could be breaking an assumption that only the owner or managers are able to send or receive.

**Remediation**

Ensure adding a step in the deployment process that removes the *hotWallet* from the whitelist when deployment is complete.

**Status**

The deployment strategy case provided to Least Authority was miscommunicated and the incorrect case was evaluated rendering this suggestion void.

**Verification**

Invalid.

# Recommendations

We recommend that the unresolved *Issues* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team. We also must reiterate the importance of authoring a full test suite for the contracts before deploying them to production.

## About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

## Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

### Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

### Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

### Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

## Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

## Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.