Gluwacoin ERC-20 Wrapper

ConsenSys Diligence

Date	October 2021
Auditors	Heiko Fisch

1 Executive Summary

This report presents the results of our engagement with **Gluwa** to review their **Gluwacoin ERC-20 wrapper** contract.

The review was conducted from **October 11–19, 2021**, and a total of 5 person-days were spent.

In July 2021, we had audited the same codebase at commit hash `3180db39d8491b972d55b8081a572867b4619ca0`, and in the meantime, the Gluwa team has made several changes to their contracts, many of them based on the findings and recommendations from that engagement.

Initially, the current review focused on the commit hash `72db3ee06e8d5889f27b6cbee7109ba8a10ea843`. The list of files in scope can be found in the Appendix. During the first week, we communicated our findings and recommendations informally to the client, who has implemented many of our suggestions over the course of that week. The commit hash of the **final version** is `bf2c7a26b1e24f29cf21bbc3445f14dcbcf8c1e7`. Our review is based on the assumption that **no existing production deployment has to be upgraded** to this version and that there will be a **new deployment** instead.

The contract we reviewed implements a Gluwacoin (T-G) backed by a specific ERC-20 token (T). T-G can be minted by locking the same amount of T in the contract; when T-G is burned, the locked T is released again. There are also "ETHless" versions of minting and burning, i.e., the user provides a signature, and the action is executed by a third party – the Wrapper – for a fee. Gluwacoins add additional functionality to the standard ERC-20 interface: There are ETHless transfers that work similarly to ETHless burning and minting, and there's a "reserve" mechanism that puts funds in escrow which can only be released to the pre-designated receiver or unlocked back to the sender.

Users of the system should be aware that it uses **upgradeable contracts**. The Gluwa team – or anyone gaining access to the key(s) with upgrade privileges (e.g., via hacking) – can, without prior notice, change the implementation contract and **take all the funds locked in the system**. We advise the Gluwa team to **follow established best practices to secure these keys**; in particular, upgrade privileges should be held by an appropriately configured multisig.

It should also be noted that there are some **requirements on the wrapped ERC-20 token**. First of all, the Gluwacoin ERC-20 wrapper contract has **no reentrancy protection**; ERC-20 tokens that would allow reentering the Gluwacoin contract – via callbacks (as in ERC-777) or any other means – should not be wrapped; doing so might result in a **loss of funds**. Secondly, the wrapped token's contract must adhere strictly to the ERC-20 specification; in particular, tokens that exhibit the missing-return-value bug can't be used with the Gluwacoin ERC-20 wrapper. *[This has been fixed in* `bf2c7a2`*; this version can handle tokens with the missing-return-value bug.]* Thirdly, more "exotic" types like **rebasing or fee-on-transfer tokens are not supported** either. The Gluwa team is not only aware of these limitations but has made a conscious choice not to support tokens with these characteristics (with the current codebase, at least). **Token contracts have to be carefully vetted** before a Gluwacoin backed by this token is created. This is, first and foremost, the Gluwa team's task, but careful users might want to verify the compatibility of the wrapped token's contract themselves.

Users should be **careful what they sign**. While this is trivially true for any signature-based system, Gluwacoin doesn't employ EIP-712; instead, users sign an "Ethereum Signed Message" with the hash of the payload. The careful user should verify that the hash matches their expectations. Moreover, they should be mindful of phishing attempts and replay attacks across different systems. If in doubt, keys and addresses used for Gluwacoin should not be used with other (signature-based) systems.

Finally, for the "Non-custodial Exchange Functions" (described in the whitepaper or the Gluwacoin Specification), it should be noted that the mentioned **3rd party – the Executor – needs to be trusted** because (before expiry) they can choose to either release the reserve amount to the pre-designated receiver (`execute`) or to unlock the funds back to the sender (`reclaim`) (or do nothing at all). If Alice and Bob want to exchange tokens, the 3rd party could `execute` for Alice and `reclaim` for Bob, so Alice would lose her tokens to Bob while Bob would get his tokens back. (Alternatively and with same result, the Executor could just `execute` for Alice and do nothing for Bob, allowing Bob to `reclaim` his tokens after a timeout.) Bob and the 3rd party could even collude and agree to share what Bob gets from Alice, effectively incentivizing the Executor to screw Alice over. So the 3rd party has some control over reserved amounts and must be trusted to "do the right thing".

2 Security Specification

This section outlines the system's primary actors and summarizes its most important trust assumptions.

2.1 Actors

The relevant actors are listed below with their respective abilities:

- **Users**. Users can `mint` / `burn` Gluwacoin tokens by depositing/withdrawing the same amount of the wrapped ERC-20 token. The usual ERC-20 functionality is available in Gluwacoins. In addition to that, users can sign `mint`, `burn`, and `transfer` operations, which are then executed by a third party (Wrapper or Relayer) for a fee. Users can also `reserve` funds for pre-defined recipients and have Executors `execute` (or `reclaim`) these reservations. After a timeout, the user can `reclaim` a reservation that wasn't executed.
- **Wrapper**. The Wrapper role can mint and burn tokens on behalf of a user. They get a fee for their service.
- **Relayer**. The Relayer role can transfer tokens on behalf of a user. They get a fee for their service.
- **Executor**. The Executor of a reservation can execute it – which transfers the locked funds to the pre-determined recipient – or they can reclaim the reservation, unlocking the funds back to the sender. Executors also get a fee in exchange for their services.
- **Admin**. The Admin role manages the system's roles.
- **System Operator**. The System Operator can upgrade the contract.

2.2 Trust Model

We briefly recapitulate what has already been discussed in section 1:

- A malicious insider or a hacker with access to the private key(s) can extract funds from the system by performing a contract upgrade. We strongly recommend implementing a rigorous security process around upgrades. A multisig wallet should be used to sign off on upgrades only when there is consensus among the team.
- ERC-20 tokens that will be wrapped by the system should be reviewed and audited. Users and relayers trust tokens to behave according to the ERC-20 specification. The wrapped token must not allow reentering the Gluwacoin ERC-20 wrapper (e.g., via callbacks); rebasing and fee-on-transfer tokens aren't supported either.
- Before reservation expiry, Executors can choose between `execute` and `reclaim` and are generally trusted to "do the right thing"; in particular, they are trusted to behave "symmetrical" to both parties involved in an exchange of tokens.

3 Recommendations

1. Due to upgradeability, users interacting with the system have to place an enormous amount of trust in Gluwa, not only in their honest *intentions* but also in their *ability* to secure the keys with upgrade privileges. Funds locked in the system are always at risk of being stolen by a malicious insider or an external hacker. We recommend moving to a system that requires considerably less trust, either by abandoning upgradeability completely or at least by utilizing a time lock for upgrades, so users of the system get a chance to withdraw their funds before an update gets activated. In the short term, the Gluwa team should follow established best practices and utilize an appropriately configured multisig such that a single or even a few compromised keys are not sufficient to upgrade the contract and steal the user funds.
   Exploring alternative designs that don't require a trusted Executor could be a promising direction for future development.
2. The current signature scheme – an "Ethereum Signed Message" with the hash of the payload – makes it hard for users to know what exactly they're signing. Moreover, users have to be aware of phishing attempts and replay attacks across different systems. We recommend following EIP-712 for the hashing and signing of structured data.
3. The Gluwacoin ERC-20 wrapper has no reentrancy protection. The Gluwa team has made it clear that they have no intention of wrapping a token that would allow reentering the wrapper contract. Nevertheless, we recommend adding a reeentrancy guard such as OpenZeppelin's as a precaution.
4. There is a considerable amount of tokens that do not strictly adhere to the ERC-20 standard, in that their `approve`, `transfer`, and `transferFrom` functions do not return anything. They revert in case of failure and indicate success just by not reverting. Although it is not standard-compliant, the number of tokens that exhibit this behavior and the importance of some of them make this issue hard to ignore when dealing with "ERC-20" tokens. We recommend using OpenZeppelin's `SafeERC20` library in `ERC20Wrapper`, so you can also wrap such tokens that don't strictly comply with the specification. *[This has been implemented in `bf2c7a2`.]*
5. The contracts of tokens to be wrapped need to be carefully reviewed to make sure they're compatible with the Gluwacoin ERC-20 wrapper. As long as the two previous recommendations haven't been implemented, special attention has to be given to reentrancy and the missing-return-value bug.
6. The findings discussed in the next section should be fixed.
7. While reviewing the correctness and comprehensiveness of the tests was not in scope for this audit, we happened to notice that central functionality was not sufficiently tested (see issue 4.2). We recommend reviewing test quality and coverage to make sure the system is well-tested before it is deployed.
8. The implementation contract should be initialized immediately after its deployment. This can either happen manually or with a constructor, see here. (Theoretically, manual initialization in a separate transaction is front-runnable, but this is unlikely to become a problem in practice.)
9. Working with upgradeable contracts has its own intricacies, dangers, and pitfalls, for example, how to handle storage and the `__gap` variable. It is crucial to educate oneself and understand these; a good resource is OpenZeppelin's documentation.
10. Some external functions don't have NatSpec annotations. To quote the Solidity documentation: "It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI)." For example, all external functions in `ERC20Reservable` are undocumented. We recommend adding NatSpec annotations to at least every contract and every public or external function. *[NatSpec annotations have been added to `ERC20Reservable` in `bf2c7a2`.]*

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Initialization flaws Minor ✓ Fixed

Resolution

This has been fixed in `bf2c7a2`.

Description

For non-upgradeable contracts, the Solidity compiler takes care of chaining the constructor calls of an inheritance hierarchy in the right order; for upgradeable contracts, taking care of initialization is a manual task – and with extensive use of inheritance, it is tedious and error-prone. The convention in OpenZeppelin Contracts Upgradeable is to have a `__C_init_unchained` function that contains the actual initialization logic for contract `C` and a `__C_init` function that calls the `*_init_unchained` function for every super-contract – direct and indirect – in the inheritance hierarchy (including `C`) in the C3-linearized order from most basic to most derived. This pattern imitates what the compiler does for constructors.

All `*_init` functions in the contracts (`__ERC20WrapperGluwacoin_init`, `__ERC20Reservable_init`, `__ERC20ETHless_init`, and `__ERC20Wrapper_init`) are missing some `_init_unchained` calls, and sometimes the existing calls are not in the correct order.

Examples

The `__ERC20WrapperGluwacoin_init` function is implemented as follows:

code/contracts/ERC20WrapperGluwacoin.sol:L36-L48

```
function __ERC20WrapperGluwacoin_init(
    string memory name,
    string memory symbol,
    IERC20 token
) internal initializer {
    __Context_init_unchained();
    __ERC20_init_unchained(name, symbol);
    __ERC20ETHless_init_unchained();
    __ERC20Reservable_init_unchained();
    __AccessControlEnumerable_init_unchained();
    __ERC20Wrapper_init_unchained(token);
    __ERC20WrapperGluwacoin_init_unchained();
}
```

And the C3 linearization is:

```
ERC20WrapperGluwacoin
  ↖ ERC20Reservable
  ↖ ERC20ETHless
  ↖ ERC20Wrapper
  ↖ ERC20Upgradeable
  ↖ IERC20MetadataUpgradeable
  ↖ IERC20Upgradeable
  ↖ AccessControlEnumerableUpgradeable
  ↖ AccessControlUpgradeable
  ↖ ERC165Upgradeable
  ↖ IERC165Upgradeable
  ↖ IAccessControlEnumerableUpgradeable
  ↖ IAccessControlUpgradeable
  ↖ ContextUpgradeable
  ↖ Initializable
```

The calls `__ERC165_init_unchained();` and `__AccessControl_init_unchained();` are missing, and `__ERC20Wrapper_init_unchained(token);` should move between `__ERC20_init_unchained(name, symbol);` and `__ERC20ETHless_init_unchained();`.

Recommendation

Review all `*_init` functions, add the missing `*_init_unchained` calls, and fix the order of these calls.

4.2 Flaw in `_beforeTokenTransfer` call chain and missing tests Minor ✓ Fixed

Resolution

This has been fixed in `bf2c7a2`. (The added tests have not been reviewed, as the tests in general are out of scope for this engagement.)

Description

In OpenZeppelin's ERC-20 implementation, the virtual `_beforeTokenTransfer` function provides a hook that is called before tokens are transferred, minted, or burned. In the Gluwacoin codebase, it is used to check whether the *unreserved* balance (as opposed to the regular balance, which is checked by the ERC-20 implementation) of the sender is sufficient to allow this transfer or burning.

In `ERC20WrapperGluwacoin`, `ERC20Reservable`, and `ERC20Wrapper`, the `_beforeTokenTransfer` function is implemented in the following way:

code/contracts/ERC20WrapperGluwacoin.sol:L54-L61

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal override(ERC20Upgradeable, ERC20Wrapper, ERC20Reservable) {
    ERC20Wrapper._beforeTokenTransfer(from, to, amount);
    ERC20Reservable._beforeTokenTransfer(from, to, amount);
}
```

code/contracts/abstracts/ERC20Reservable.sol:L156-L162

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual override (ERC20Upgradeable) {
    if (from != address(0)) {
        require(_unreservedBalance(from) >= amount, "ERC20Reservable: transfer amount exceeds unreserved balance");
    }

    super._beforeTokenTransfer(from, to, amount);
}
```

code/contracts/abstracts/ERC20Wrapper.sol:L176-L178

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual override (ERC20Upgradeable) {
    super._beforeTokenTransfer(from, to, amount);
}
```

Finally, the C3-linearization of the contracts is:

```
ERC20WrapperGluwacoin
  ↖ ERC20Reservable
  ↖ ERC20ETHless
  ↖ ERC20Wrapper
  ↖ ERC20Upgradeable
  ↖ IERC20MetadataUpgradeable
  ↖ IERC20Upgradeable
  ↖ AccessControlEnumerableUpgradeable
  ↖ AccessControlUpgradeable
  ↖ ERC165Upgradeable
  ↖ IERC165Upgradeable
  ↖ IAccessControlEnumerableUpgradeable
  ↖ IAccessControlUpgradeable
  ↖ ContextUpgradeable
  ↖ Initializable
```

This means `ERC20Wrapper._beforeTokenTransfer` is ultimately called twice – once directly in `ERC20WrapperGluwacoin._beforeTokenTransfer` and then a second time because the `super._beforeTokenTransfer` call in `ERC20Reservable._beforeTokenTransfer` resolves to `ERC20Wrapper._beforeTokenTransfer`. (`ERC20ETHless` doesn't override `_beforeTokenTransfer`.)

Moreover, while reviewing the correctness and coverage of the tests is not in scope for this engagement, we happened to notice that there are no tests that check whether the unreserved balance is sufficient for transferring or burning tokens.

Recommendation

`ERC20WrapperGluwacoin._beforeTokenTransfer` should just call `super._beforeTokenTransfer`. Moreover, the `_beforeTokenTransfer` implementation can be removed from `ERC20Wrapper`.

We would like to stress the importance of careful and comprehensive testing in general and of this functionality in particular, as it is crucial for the system's integrity. We also encourage investigating whether there are more such omissions and an evaluation of the test quality and coverage in general.

4.3 Hard-coded decimals Minor ✓ Fixed

Resolution

In `bf2c7a2`, the decimals are provided as an initialization parameter and stored in a state variable, as we recommended. However, the Gluwa team chose to leave the decimals logic in the `ERC20WrapperGluwacoin` contract.

Description

The Gluwacoin wrapper token should have the same number of decimals as the wrapped ERC-20. Currently, the number of decimals is hard-coded to 6. This limits flexibility or requires source code changes and recompilation if a token with a different number of decimals is to be wrapped.

code/contracts/ERC20WrapperGluwacoin.sol:L32-L34

```
function decimals() public pure override returns (uint8) {
    return 6;
}
```

Recommendation

We recommend supplying the number of decimals as an initialization parameter and storing it in a state variable. That increases gas consumption of the `decimals` function, but we doubt this view function will be frequently called from a contract, and even if it was, we think the benefits far outweigh the costs.
Moreover, we believe the decimals logic (i.e., function `decimals` and the new state variable) should be implemented in the `ERC20Wrapper` contract – which holds the basic ERC-20 functionality of the wrapper token – and not in `ERC20WrapperGluwacoin`, which is the base contract of the entire system.

Appendix 1 - Files in Scope

This audit covered the following files:

A.1.1 Initial version

Commit hash: `72db3ee06e8d5889f27b6cbee7109ba8a10ea843`

File Name	SHA-1 Hash
contracts/ERC20WrapperGluwacoin.sol	fb66dc6b671cf27137e342bdeed596e39d433fa5
contracts/abstracts/ERC20ETHlessTransfer.sol	049886593bf87559b1a1aac654a051f3f4958413
contracts/abstracts/ERC20Reservable.sol	66d0f77355a42b5a5bf826aad0131c23606d0e90
contracts/abstracts/ERC20Wrapper.sol	0f33b03a65bfbe39148a64e41be9f5717d0ae90e
contracts/abstracts/Validate.sol	d4f293cb9870f28ca05cc0c35300963f83b5452b
contracts/libs/GluwacoinModel.sol	44921f971f529e5ae2419c153120def8d32fa746

A.1.2 Revised version

Commit hash: `bf2c7a26b1e24f29cf21bbc3445f14dcbcf8c1e71`

File Name	SHA-1 Hash
contracts/ERC20WrapperGluwacoin.sol	22fc2b92fb0793ed7fd5bb2ed482f5e3d80a27c8
contracts/abstracts/ERC20ETHlessTransfer.sol	1d825daa64beb58c6695d8b1328aca9069914534
contracts/abstracts/ERC20Reservable.sol	d5215896492290681dad614126094de400bc8d40
contracts/abstracts/ERC20Wrapper.sol	845d185d45ad66c9a837c4b59656eb0906b7bb8a
contracts/abstracts/Validate.sol	9ae992a0d990cf94f5d740dd2749d9b6adf5d837

File Name	SHA-1 Hash
contracts/libs/GluwacoinModel.sol	3d363edaf6ccbf9719947f0428ac1bb17ec8d26f

Appendix 2 - Disclosure