dxDAO ERC20 Guilds Audit No. 1

Jan, 2021

Chapter 1

Introduction

1.1 Scope of Work

This code review was prepared by Sunfish Technology, LLC at the request of members of dxDAO, an organization governed by a smart contract on the Ethereum blockchain. The code covered by this review (see section 1.2) implements a contract that makes external calls based on votes weighted by ownership of a particular ERC20 token.

1.2 Source Files

This review covers code from the following public git repository and commit SHA:

`github.com/AugustoL/dxdao-contracts`
`ccdf8d907bd5f121ae34c6d07aae9548f3cfe773`

Within that commit, only the following files were reviewed:

- ERC20Guild.sol
- ERC20GuildLockable.sol
- ERC20GuildPayable.sol
- ERC20GuildPermissioned.sol
- ERC20GuildSigned.sol
- ERC20GuildSnapshot.sol

This review was conducted under the optimistic assumption that all of the supporting software infrastructure necessary for the deployment and operation of the reviewed code works as intended. There may be critical defects in code outside of the scope of this review that could render deployed smart contracts inoperable or exploitable.

1.3 License and Disclaimer of Warranty

This source code review is not an endorsement of the code or its suitability for any legal/regulatory regime, and it is not intended as a definitive or exhaustive list of defects. This document is provided expressly for the benefit of dxDAO developers and only under the following terms:

Chapter 2

Critical Defects

2.1 Reentrancy Vulnerability in Proposal Execution

The `executeProposal()` function does not guard against being re-entered from the contract that it calls as part of proposal execution. The following line of code (on line 141 of `ERC20Guild.sol`) occurs **after** all the calls for the proposal have been made:

```
proposals[proposalId].executed = true;
```

This bug can be exploited directly by an attacker as follows:

- An attacker implements a contract with a fallback function that calls `ERC20Guild.executeProposal()` unless the contract reaches some predefined balance. The attacker takes note of the address at which this contract would be deployed given the deployment account and nonce, but **does not deploy it yet**.
- The attacker convinces the `Guild` contract to make a small payment to the address noted in step 1. At this point in time, the address appears to be an ordinary (non-contract) address.
- When the proposal passes, the attacker deploys the exploitation contract to the target address and calls `executeProposal()` as part of a single transaction. The target contract repeatedly calls `executeProposal()` again to drain the `Guild` contract balance in increments of the original payment amount.

An attack similar to the one above can be performed with ERC20 tokens that implement a token "fallback" function.

Chapter 3

Moderate Defects

Issues discussed in this section are code defects that may lead to unintended deviations in behavior. It may be possible to chain multiple moderate defects into a working exploit.

3.1 Proposals Can Grief Minority Voters

The `ERC20GuildLockable` contract requires ERC20 tokens to be locked in the contract in order for votes associated with those tokens to be counted. However, proposals can spend tokens deposited into the contract, and the contract does not take steps to avoid spending tokens that have been locked in the contract for voting purposes rather than spending purposes. Consequently, a small majority of `Guild` voters can grief a large minority of voters by successfully passing proposals that spend the tokens associated with their votes.

The `ERC20GuildLockable` contract should not carry the balance of locked tokens itself. Instead, it should examine the balance locked in an external contract that it trusts but does not directly control.

3.2 Permissions Mis-handle Empty Calldata

The `getFuncSignature()` function in `ERC20GuildPermissioned.sol` reads the first four bytes of the transaction `calldata` via inline assembly, even when the length of `calldata` is zero. Consequently, the function fails to distinguish between *no* function selector and a zero-valued function selector. However, the bytecode emitted by the solidity compiler *does* make this distinction; zero is a valid ABI selector.

The code in `ERC20GuildPermissioned.sol` ought to distinguish between ordinary function selectors and invalid or non-existent function selectors, as there may be a meaningful semantic distinction between the two.

Chapter 4

Minor Defects

Issues discussed in this sections are subjective code defects that affect readability, reliability, or performance.

4.1 Consider Merging Contracts

The `ERC20Guild.sol` contract is not a safe voting contract on its own, as flash loans make it possible to borrow a significant quantity of the voting token without undertaking any capital risk. Consequently, the deployed code will *at least* need `ERC20GuildLockable` in order to be a reasonably safe design (but, see chapter 5). Since it may be ergonomic to have token "lock" periods less than the duration of a vote, the `ERC20GuildSnapshot` code will likely also have to be present in the production contract.

The fact that these contracts all inherit from one another makes it possible that a programmer could unintentionally use the wrong security-critical inherited function. For example, the version of `votesOf()` in `ERC20Guild` is trivially exploitable for the reasons discussed in the previous paragraph, but its presence in the code means that it could still be used unintentionally.

In order to make the code more "obviously correct," consider implementing all of the *required* security functionality as a single base contract.

Chapter 5

Economic Considerations

In the `ERC20GuildLockable` contract, the cost of a vote is equivalent to the cost of borrowing that quantity of the guild's ERC20 token for the lock period duration. Since a voter can eliminate the counter-party risk associated with that loan by voting through a contract that deterministically pays back the principal after the lock period, the only risk a lender would be exposed to is duration risk. Consequently, we would expect the interest rate for these loans to be very low, particularly when interest rates themselves are stable. Consider that the overnight dollar swap rate in the U.S. is 0.086% as of this writing. This means the market-clearing rate for (approximately) "risk-free" dollars in the future is so low that borrowing an extraordinary quantity of dollar-denominated assets for just a single voting period is *almost* free. Given how cheaply an attacker could accumulate extraordinary voting power, it isn't clear how voting with fungible tokens leads would lead to desirable outcomes, as the "weight" of each voter would effectively be determined by their respective access to large quantities of very-short-term debt.