dxDAO Wallet Scheme Audit No. 1

April, 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.

This review covers new "scheme" contracts that allow the DAO to transfer funds and make arbitrary external calls. The feature set of these contracts is similar to that of the `GenericSchemeMultiCall` contracts reviewed for dxDAO in October of 2020.

1.2 Source Files

This review covers code from the following public git repositories and commits:

`github.com/AugustoL/dxdao-contracts`
`512a856442fcf00d83d4b8edba56f694dd7cd246`

Within those commit ranges, only the following files were reviewed:

- contracts/schemes/WalletScheme.sol
- contracts/schemes/PermissionRegistry.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 de-

fects. This document is provided expressly for the benefit of dxDAO developers and only under the following terms:

THIS REVIEW IS PROVIDED BY SUNFISH TECHNOLOGY, LLC. “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL SUNFISH TECHNOLOGY, LLC. OR ITS OWNERS OR EMPLOYEES BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS REPORT OR REVIEWED SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Major Issues

Issues discussed in this section are defects that are directly exploitable without any additional bugs.

2.1 Destination Address Forgery

The `PermissionRegistry.getPermission()` function calls `to.implementation()` in order to determine if the destination of a transfer of funds is a proxy contract. However, the implementation of `getPermission()` does not have *a priori* knowledge that the destination implements the `implementation()` function faithfully. An attacker can simply publish a contract that returns an arbitrary (attacker-controlled) address from `implementation()` in order to forge the `to` address that is used for the rest of the body of `getPermission()`. In other words, the `to` parameter in `getPermission()` is always attacker-controlled, which makes it impossible for the rest of the function to distinguish between a transfer to a "known-good" address versus an attacker-controlled address.

As a general rule of thumb, you should not trust the return value of functions called on arbitrary contracts. If you have not audited the code, you do not know what the functions will return. You cannot assume that functions named according to standard interfaces will always behave as you expect; **you must confirm that they behave as you expect by manually inspecting the code**.

Also, it is not safe to use `bytes4(0xaaaaaaaa)` as a sigil for a function signature wildcard. It is trivial to generate a real function with that selector (for example, `expressionalolivinitic()`).

2.2 Function Signature Punning

It is not safe for `WalletScheme.isCallAllowed()` to examine the `calldata` of a transaction in order to determine its behavior. In this particular case, `isCallAllowed()` automatically promotes `asset` to `to` and sets `to` and `value` to attacker-

controlled arguments when it encounters a function signature that matches `transfer(address,uint256)`.

An attacker-controlled contract can implement a function like `transfer(address,uint256)` without it actually implementing anything that looks like an ERC20 token transfer. Moreover, Solidity function selectors are only four bytes, so you can't ensure that you've uniquely identified a particular function call when you match a selector. For example, the `transfer(address,uint256)` selector is the same as `multitonedocreaceous(address,uint256)`, and `approve(address,uint256)` is the same as `addressorqualified(address,uint256)`.

An attacker can provide `to = 0` and crafted `calldata` (prefixed with the appropriate `ERC20.transfer()` function signature) in order to override the value of `asset` and `value` passed to `getPermissions()`. So, the call to `getPermissions()` could look like this, where `to` is attacker-controlled:

```
address from = controllerAddress != address(0) ?
               address(avatar) :
               address(this);
getPermissions(0, from, to, callSignature);
```

This means an attacker can easily assume the permissions granted to any particular destination address as long as there is an association with that address and `ERC20.transfer()` or `ANY_SIGNATURE` in the permissions contract.

Chapter 3

Moderate Issues

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 are not Atomic

The `WalletScheme.executeProposal()` function executes each proposed call in sequence, but it does not insist that all of the calls succeed or fail together; it simply stops executing calls once it encounters a call that fails. In other words, if one call of the proposal succeeds, but another fails, the proposal will mark itself as `ProposalState.ExecutionFailed`, but the part of the proposal that succeeded will not be reverted.

If parts of a proposal involve transactions that are not idempotent (such as calls to `ERC20.approve()`), a partially-executed proposal could leave funds belonging to the DAO's `Avatar` in an unsafe state.

The `executeProposal()` function should not leave a proposal in a partially-executed state. If any components of the proposal fail to execute successfully, the `executeProposal()` function needs to revert the entire transaction in order to be sure that it left the target contracts in a reasonable state.

Chapter 4

Other Notes

Function signatures are easily forged. Do not use them in a security-sensitive context unless you are examining calls being made to a pre-audited contract. Both of the major issues uncovered in this review are due to the code implicitly trusting attacker-controlled inputs and return values.

Function signatures are only meaningful when they are evaluated in the context of a particular callee contract. For example, if you want to examine arguments to a call that looks like `ERC20.transfer(address,uint256)`, **you must already know that the callee faithfully implements the ERC20 standard**.