dxDAO Wallet Scheme Audit No. 2

June, 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. The first review of these new contracts was in April of 2021.

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

Whitelist Bypasses

This section discusses techniques an attacker could employ to bypass the `PermissionRegistry` whitelist. The severity/exploitability of these issues will depend on the run-time configuration of the contracts, but all of the issues described below involve configurations that are likely in practice and are at least superficially benign. This audit does *not* include descriptions of attacks that require passing more than one attacker-controlled proposal or attacks that rely on social engineering.

2.1 Bypass via Allowance

Contracts that accept ERC20 tokens for exchange or payment (like Uniswap pair and router contracts, Gnosis trading pairs, etc.) require that tokens are deposited via `ERC20.transferFrom()`, so the caller must have previously provided the correct allowance to the target contract via `ERC20.approve()`. In fact, it was this particular "approve-and-call" sequence that was highlighted as a primary use-case for the "Multicall Scheme" contracts reviewed in October of 2020, so presumably the same design rationale applies here as well.

In order to allow `ERC20.approve()` to be called on an ERC20 token, the whitelist must have `permissions[0][avatar][token][sig]` or `permissions[0][avatar][token][ANY_SIGNATURE]` configured. If any of those permissions are configured, then **any quantity of tokens can be approved to any address**. That means an attacker can simply create a proposal that calls `approve(attacker, uint256(-1))` and then subsequently call `token.transferFrom()` directly to transfer the entire balance of the token to any address.

More broadly, you are not guaranteed that `ERC20.transfer()` is the only way to produce a balance transfer, so in general it is unsafe to configure the whitelist to allow any calls to ERC20 tokens that are not `ERC20.transfer()`. However, forbidding calls to common functions like `ERC20.approve()` will severely limit the usefulness of this new code.

2.2 Bypass via Wildcard Permission

If `permissions[0][avatar][ANY_ADDRESS][ANY_SIGNATURE]` is set, then the attack described above can be carried out with **any** token, not just a specific ERC20 token that the contract ordinarily deals with.

(The combination of `ANY_ADDRESS` and `ANY_SIGNATURE` essentially allows the whitelist to be bypassed completely, so that configuration is almost definitionally unsafe.)

2.3 Bypass via Ownership

Typcially, contracts that "belong" to dxDAO are owned by the DAO's `avatar` contract. So, in practice it will likely be the case that the `PermissionRegistry` contract will be owned by the `avatar` contract, as that will allow the DAO to manage the whitelist using the same voting infrastructure that they use for managing other contracts.

However, the `PermissionRegistry.owner` address is granted special permissions when the `PermissionRegistry` contract is constructed or `transferOwnership()` is called:

```
permissions[address(0)][_owner][address(this)][ANY_SIGNATURE]
    .fromTime = now;
```

However, setting the permissions as above creates an exploitable configuration when `WalletScheme.controllerAddress` is non-zero. Permissions for ordinary calls are determined as follows in the `WalletScheme` code:

```
(_valueAllowed, _fromTime) = permissionRegistry
    .getPermission(
        address(0),
        controllerAddress != address(0) ? address(avatar) : address(this),
        proposal.to[i],
        callSignature
     );
```

So, when `controllerAddress` is non-zero, the right-hand-side expression looks like this:

```
getPermission(0, avatar, proposal.to[i], callSignature)
```

That means a proposal that makes **any** call to the `PermissionRegistry` contract itself will satisfy the check performed by `getPermission()` due to `avatar` being a privileged address in the whitelist contract.

Therefore, an attacker can bypass the whitelist entirely by beginning a proposal with two calls to the `PermissionRegistry` contract via the avatar. The first call of `PermissionRegistry.setDelay(0)` eliminates the mandatory delay for updates to

whitelist permissions, and the second call to `setAdminPermission()` can immediately update whatever permissions are necessary to successfully execute the rest of the attack. (Since `executeProposal()` evaluates permissions sequentially just before each external call is made, the new permissions will take effect immediately after the first set of calls to `PermissionRegistry`, so the subsequent calls in the proposal will be able to use the new permissions.)

This attack is also possible when the `WalletScheme` contract owns the `PermissionRegistry` contract, although that configuration seems less likely in practice.

2.4 Value Tracking Bypass

The `permissionHash` used to track the cumulative ERC20 token value transferred in each proposal by hashing the token address with the destination address and using that as an index to look up the total value transferred.

There are two ways to bypass this check.

First, an attacker can simply have the proposal make transfers to multiple different attacker-controlled addresses. The only additional cost imposed on an attacker is the additional gas cost of making multiple `ERC20.transfer()` calls rather than just one.

Second, an attacker can use re-entrancy via an external call in a proposal to cause a second proposal to be executed. If this second proposal executing concurrently with the first proposal uses the same `permissionHash` at any point, then the value in `totalValueTransferedInCall[permissionHash]` will be zeroed at the end of the proposal execution, which would allow the first proposal to continue as if no tokens had been transferred. Note that this second proposal does not need to be attacker-controlled; the attacker just needs to abuse another proposal that makes a payment with an equivalent `permissionHash`.

Chapter 3

Minor Issues

3.1 Default Proposal State is "Submitted"

The enum `ProposalState` begins with `Submitted`, which means that the "zero value" of `ProposalState` is `Submitted`. Consequently, the following check on lines 119-120 of `WalletScheme.sol` will always pass for invalid proposal IDs:

```
Proposal storage proposal = proposals[_proposalId];
require(proposal.state == ProposalState.Submitted, ...);
```

This issue probably isn't exploitable without an additional bug in the `VotingMachine` contract(s), but it is inadvisable to allow an uninitialized `Proposal` to pass this check either way.

3.2 Obsolescent Solidity Version

The code covered in this review still uses solidity version `0.5.17`, which was released more than one year ago. (Other dxDAO code appears to have migrated to more recent releases.)

For information on bugs associated with specific solidity versions, see `https://github.com/ethereum/solidity/blob/develop/docs/bugs.json`. There appear to be a handful of low- and medium-severity issues that were introduced before and fixed after release `0.5.17`.