Report

Smart Contract Audit

dao.care

23rd of April 2020

1. Preface

The team of **dao.care** contracted ditCraft to conduct a software audit of their developed smart contracts written in Solidity. dao.care is a project creating a platform to democratically support community projects with joint interest earned through staking a stable coin called DAI[1] on the Ethereum blockchain.

The following services to be provided by ditCraft were defined:

- Manual code review
- Protocol/Logic review and analysis (including a search for vulnerabilities)
- Written summary of all of the findings and suggestions on how to remedy them (including a Zoom call to discuss the findings and suggestions)
- Final review of the code once the findings have been resolved

ditCraft gained access to the code via the public GitHub repository via https://github.com/DAOcare/app/tree/master/contracts. The state of the code that has been reviewed was last changed on the 14th of April 2020 at 09:38 AM CEST (commit hash 99b3f553158d4fe3b85facf6461d313695efa188).

[1] An ERC20 token where 1 DAI is pegged to be valued at 1 USD at all times. Developed by the Maker foundation.

2. Manual Code Review

ditCraft conducted a manual code review, where we focussed on the two main smart contracts as instructed by the dao.care team: "NoLossDao_v0.sol" and "PoolDeposits.sol". For a description of the functionalities of these contracts refer to section 3.

The code of these contracts has been written according to the latest standards used within the Ethereum community and best practice of the Solidity community. The naming of variables is logical and comprehensible, which results in the contract being easy to understand. As the dao.care project is a decentralized and open-source project, these are important factors.

The comments in the code help to understand the idea behind the functions and are generally well done, but in some areas there is still room for improvement.

On the code level, we did not find any alarming bugs or flaws. The ones we describe below (2.1 and 2.1) are of low severity.

2.1. NoLossDao_v0.sol

Line 209

```
  /// @dev Changes the amount required to stake for new proposal
  function setInterestReceivers(
    address[] memory _interestReceivers,
    uint256[] memory _percentages
  ) public onlyAdmin {
```

The comment is wrong, it has been copied from the function above (line 205).

Line 384

```
  /// @dev Anyone can call this every 2 weeks (more specifically every
*iteration interval*) to receive a reward, and increment onto the next
iteration of voting
  function distributeFunds() external iterationElapsed {
```

Some form of frontrunning is definitely possible here, but is expected and desired. Just noting for completeness, more details in the protocol/logic analysis.

2.1. PoolDeposits.sol

Line 263 (+ Line 268)

```
uint256 amountToSend = amountToRedeem.mul(percentages[i]).div(1000)

uint256 amountToSendToWinner =
amountToRedeem.mul(percentageWinner).div(1000);
```

Technically these lines do the right thing, but the division allows for a slight remainder in the case of an odd number which is not considered. See the following example:

> Let amountToRedeem be 1001 and the percentages be 13,5% and 1,5% as proposed by you. The divisions compute as follows:
>
> First iteration of the loop (line 261): amountToSend = 135,135, but since uint256 is an integer, the place after the decimal point will be dropped without any rounding leading to a value of 135.
>
> Same for the second iteration where we receive a value of 15 instead of 15,015.
>
> Line 268 will now compute the amountToSendToWinner as 850 instead of 850,85.
>
> In the process, the amount of “1” will be left and won’t be distributed..

The severity of this is very low. Since the DAI token is denominated with 18 decimal places as a 10^{18} value in uint256, this would mean that a value of 10^{-18} DAI is being kept without being distributed. The remainder would be distributed as soon as the amountToRedeem is even at the end of an iteration. There is no risk associated with this. If this is desired to be fixed, a possible solution would be to store and add the two amounts that are being sent within the for-loop (lines 261 to 266) and subtract this value from the original value of amountToRedeem. The result would be the rest that is being paid to the winning project, without leaving a remainder.

As the difference that this fix makes doesn’t have any real economic impact and probably introduces a slightly higher gas cost of the execution, there is **no** real benefit of fixing it.

3. Protocol/Logic Review

There are basically two main groups of participants when it comes to the user-side of the protocol. **Regular users** and **project owners**.

Regular users can stake DAI tokens to receive voting rights in the same height as their stake and influence the vote, where one project is regularly chosen to receive the accrued interest of the users. The DAI that is being staked is automatically lended via the Aave protocol in order to gain interest.

Project owners can stake DAI tokens to propose a project to be voted upon.

The whole protocol works in so called iterations, where one iteration will take two weeks (not fixed in code, can be changed). At the end of each iteration the winner of the week before will receive the accrued interest. A project that has won will be put on cooldown, which will be removed after the next iteration. A project on cooldown can't win another prize during that time.

On top, there is an emergency functionality, where everyone who staked can vote for an emergency. As soon as >50% of the users voted on this, the state of emergency can be activated, which allows everyone to withdraw their stakes.

3.1. Functionality Descriptions

Regular Users

- Has three main functionalities: deposit, withdraw, vote
    - Additionally they can delegate their vote and vote on someones' behalf
- **Deposit**: Enters the pool with amount X (where amount X has to be >= the required amount)
    - Condition: User doesn't have a deposit yet
    - Condition: User doesn't have a proposal
- **Withdraw**: Leaves to pool and withdraws amount X
    - Condition: User hasn't voted during the current iteration
    - Condition: User doesn't have a proposal
- **Vote**: Votes on a proposal with a voting power of staked amount X
    - Condition: User has staked amount X
    - Condition: Proposal is active
    - Condition: User hasn't voted during the current iteration
    - Condition: User doesn't have a proposal
    - Condition: User joined before the current iteration started
- **Delegate Voting Rights**: Delegates the votings rights (amount X) to an address A

    - Condition: User has staked amount X
    - Condition: User doesn't have a proposal
    - Condition: Address A doesn't have a proposal
- **Vote via Proxy**: Votes on behalf of someone (address B) who delegated their voting rights of staked amount X
    - Condition: Address B has staked amount X
    - Condition: Proposal is active
    - Condition: Address B hasn't voted during the current iteration
    - Condition: Address B doesn't have a proposal active
    - Condition: Address B joined before the current iteration started
    - Condition: User has the right to vote on behalf of Address B

Project Owner

- Has two main functionalities: create and withdraw
- **Create**: Enters the pool with amount X (where amount X is the required amount) and adds the project as a proposal
    - Condition: User doesn't have a deposit yet
    - Condition: User doesn't have a proposal yet
- **Withdraw**: Leaves to pool and withdraws amount X, sets the proposal to "withdrawn" status
    - Condition: User has an active proposal
    - Condition: User was part of the pool for more than 2 iterations with this proposal

Emergency Functions

- Everyone can do two things: vote on emergency and withdraw if emergency has been declared
    - Additionally anyone can declare the state of emergency once more than 50% of the staked tokens have voted in favor
- **Vote**: Votes for an emergency state with staked amount X
    - Condition: User hasn't voted on emergency yet
    - Condition: User is part of the pool for more than 100 days
- **Withdraw**: Withdraws staked amount X during an emergency
    - Condition: Case of emergency has been activated
- There is a public function that can be called by anyone, activating the state of emergency
    - Condition: More than 50% of the users have voted in favor of an emergency
    - Condition: More than 200.000 DAI are staked in total

Public Mining Function

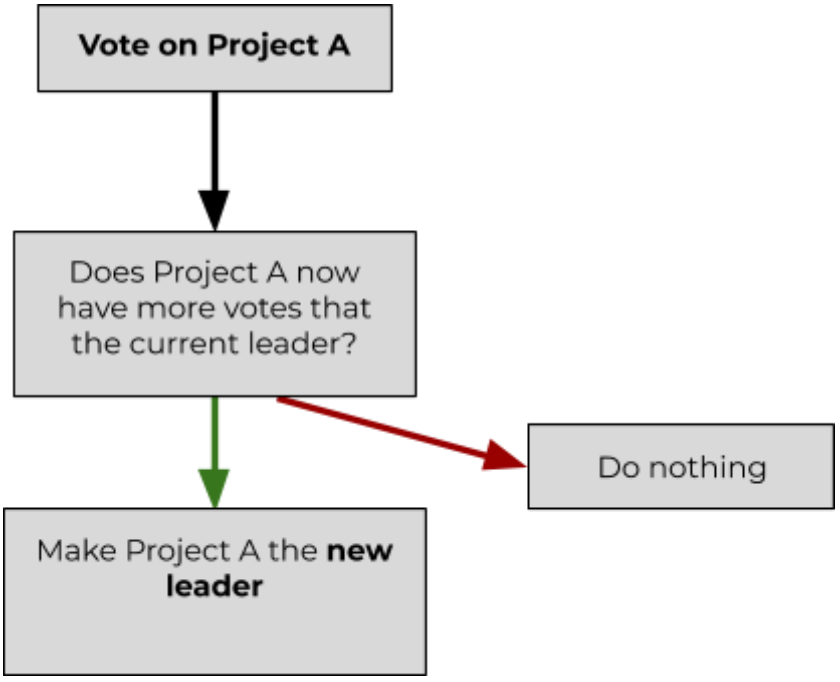
- In order to proceed to the next iteration there is a public mining function that can be called by anyone
    - Condition: The current iteration has elapsed

3.2. Protocol Logic

While the functionality description covers most of the logic, there are two things that need to be explained separately: the voting mechanism and the fund distribution during the iteration change.

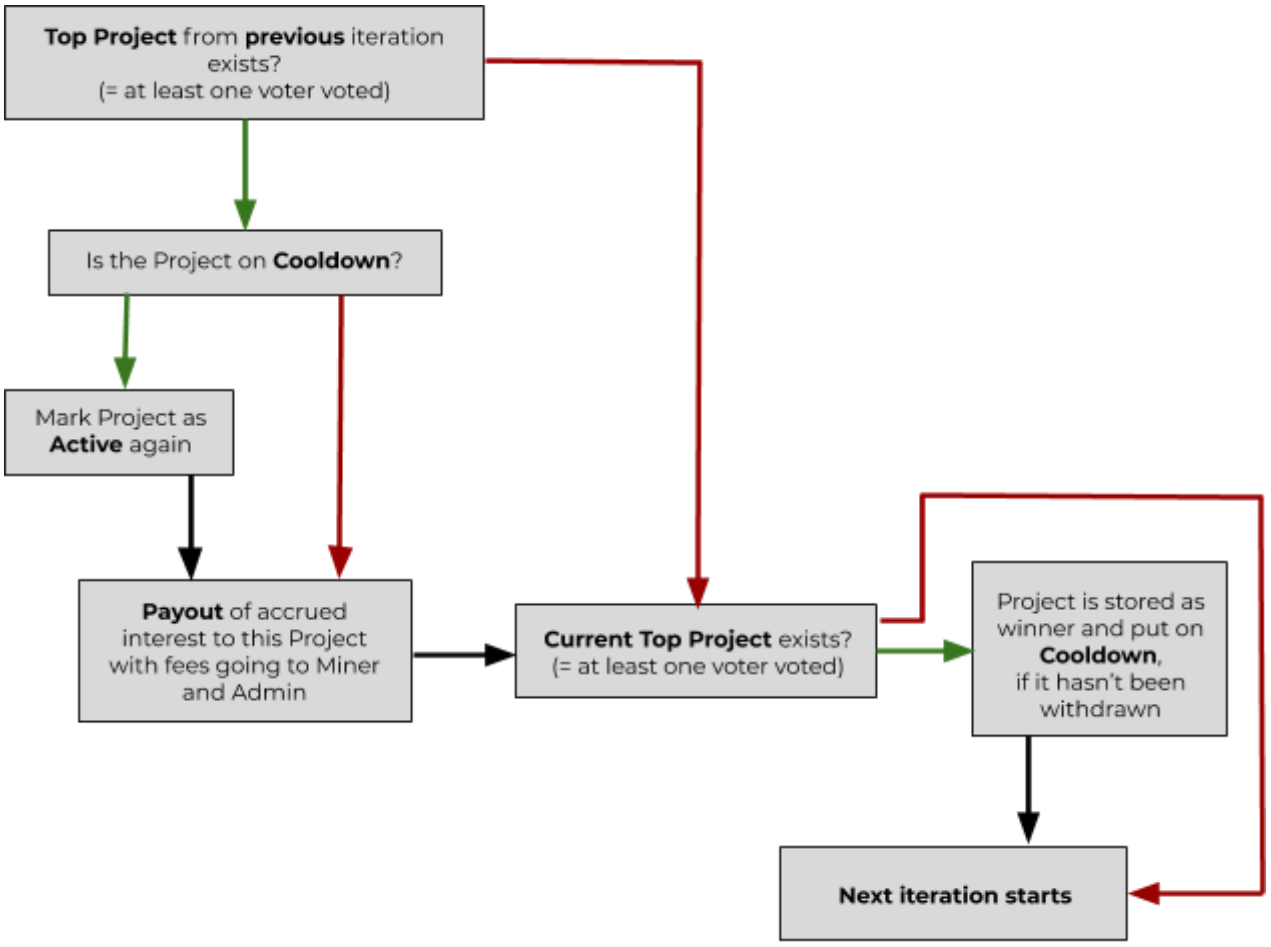
Voting Mechanism

If a user votes on a project, their token amount is taken as votes and added to the counter of the corresponding project for the current iteration. In the event of a tie, the project that reaches the number of votes first wins.



Fund Distribution and Iteration Change

At the end of each iteration a function will start the next iteration and pay out funds to the last week's winning project. A winning project is called the ”top project”.



3.3. Vulnerabilities and Flaws

While we did not find any bugs during the manual code review, we do have some potential flaws/vulnerabilities that are potentially exploitable and might pose some risks on the protocol level.

A) No project check for delegate voter (NoLossDao_v0.sol)

When a regular user wants to vote on a proposal, it is ensured that this account does not have a project/proposal themselves (line 330). We think that this is a good measure to separate the two types of users and clearly distinguish between the ones who are giving and the ones who receive something. This check is also in place when someone wants to transfer their voting rights to another address, where both are not allowed to be a project owner (lines 314-315). However, you are not checking whether the person who receives the voting rights still doesn't have a proposal once they are voting. In theory, the following can happen:

- **Alice** is a regular user and stakes 1000 DAI
- **Bob** is not yet a participant (not a project owner nor a regular user that staked)
- **Alice** delegates her voting rights to **Bob**
- **Bob** then creates a proposal himself
- **Bob** votes on his own proposal via proxy (with the voting power of the 1000 DAI of **Alice**)

This can potentially lead to the abuse of the vote via proxy system, where users like Bob may promise to vote on behalf of other users in a good way, only to later create a proposal himself and then vote on it with their votes.

Possible mitigation

The easiest way to fix this would be to include another check in the voteProxy function (line 340). Therefore we propose the following change. Currently, the list of modifiers for the "voteProxy" function looks like this:

```
proposalActive(proposalIdToVoteFor)
proxyRight(delegatedFrom)
noVoteYet(delegatedFrom)
userStaked(delegatedFrom)
userHasNoActiveProposal(delegatedFrom)
joinedInTime(delegatedFrom)
```

The following addition would mitigate the risk (changes in bold):

```
proposalActive(proposalIdToVoteFor)
proxyRight(delegatedFrom)
noVoteYet(delegatedFrom)
userStaked(delegatedFrom)
userHasNoActiveProposal(delegatedFrom)
userHasNoActiveProposal(msg.sender)
joinedInTime(delegatedFrom)
```

B) Distribute Funds allows to frontrunning (NoLossDao_v0.sol)

The function “distributeFunds” in line 385 to 431 is prone to frontrunning. Since anyone can execute it after an iteration has finished, the first one who submits their transaction will succeed. If the project gains enough traction, there might be people who apply scripts to do this automatically and outbid other people trying to do so by upping the gas price, leading to a mining-like process. We are aware that this is probably a desired property, since it definitely keeps the protocol running in a timely manner and incentivizes people to do so on a continuous basis. It is probably not the fairest mining mechanism, but should be sufficient to produce the desired outcome without posing any risks to the users. We are not flagging this as a vulnerability or flaw, just noting it for completeness.

C) Possible flaw in the Cooldown flagging mechanism (NoLossDao_v0.sol)

The function “distributeFunds” in line 385 to 431 includes a segment where the last week's project receives their accrued funds. In lines 416 to 419 the project is being set on cooldown if it hasn’t been withdrawn already.

```
if (state[iterationTopProject] != ProposalState.Withdrawn) {
   state[iterationTopProject] = ProposalState.Cooldown;
   emit ProposalCooldown(iterationTopProject, proposalIteration);
}
```

Potentially, a project could withdraw their project immediately before the iteration ends and they are sure to be winning. They would still win and could then go ahead to submit another project for the next iteration and potentially win again, not being blocked by the cooldown.

Possible mitigation

If this behaviour is not desired, its abuse could be blocked in two possible ways. First, you could introduce a check whether someone who aims to withdraw their proposal is currently leading the vote and thus block them from withdrawing. Another possible solution would be to check whether someone creating a new proposal is actually the benefactor of last week's proposal.

D) Current weeks prize money goes to last weeks winner

In your protocol, the winner of an iteration will receive the accrued interest of the following week. Vice versa, at the end of an iteration the accrued interest will be sent to the winner of the week before (e.g. winner of iteration 6 receives the interest of iteration 7 and so on). This could potentially lead to the following case:

- Project A wins with a very slim majority of 51% during the current iteration
- They are now eligible to receive the next iterations accrued interest
- The 49% that voted for another project are not happy with the outcome and leave the pool immediately
- Project A receives 49% less money at the end of the next iteration because people rage-quitted after the vote

Although we can see that this might be intentional, we also see a DAO as a democratic and solidary construct. The vote loses some of its credibility if the participants don't need to act according to what they agreed on. If I enter the social construct of a DAO voting scheme, I am probably okay with the democratic nature of the fund distribution. Of course, it is okay to allow people to leave the pool at any time, but they shouldn't be able to influence the result afterwards through leaving it. In the current implementation they are essentially voting on next week's money, where they might not be part of the pool anymore if they decide to leave.

Possible mitigation

If this behaviour is not desired, it would be possible to give the current iterations accrued interest to the winner of the same iteration, equivalent to how projects like PoolTogether are handling it as far as we are aware.

E) Vote is public and prone to potentially be manipulated by whales

Since your voting scheme is public and does not provide voting confidentiality, it is possible to see the current standing at all times. This could potentially allow people with large holdings inside the pool to wait till the iteration is nearing completion and then decide on the project that they are in favor of, especially if two projects are nearly tied.

Possible mitigation

A possible solution to this would be the implementation of a commit-reveal-based voting scheme, where the votes will be public after the iteration finishes but are confidentiale before this happens. We do see that this might now be desired and you might be okay with the public property and the risks that it poses. We also see possible implications for usability that might lead you to deciding not to use a commit-reveal-based voting scheme, since it requires the user to facilitate an additional transaction during the reveal-phase, which is not nearly as nice from a UX

perspective. We merely want to draw attention to the implications of a public voting scheme and list it for completeness.

F) Fixed 200k limit for emergency function (PoolDeposits.sol)

In lines 65 to 68 there is a fixed limit of 200.000 DAI that acts as a limit for the emergency state functionality.

```
require(
    totalDepositedDai > 200000000000000000000000,
    //200 000 DAI needed in contract
    '200 000DAI required in pool before emergency state can be
     declared'
);
```

Below this threshold, it is not possible to activate the emergency state. While there might be a rational decision behind this implementation, we would at least like to see an explaining comment for anyone interested in this. We personally do see a problem when just 10.000 DAI are locked in the smart contracts and an emergency occurs. In this case, none of the participants would be able to recover their funds. A very simple solution would be lowering this limit, which does indeed introduce more risks of abuse of the emergency function. A better solution would probably be a bit more complex, but we still suggest to spend some thoughts on an alternative solution, even if it is “just” for the next version of the smart contracts.

4. Summary

Overall the smart contracts are very well written and thought out. We like the idea of not recycling any foreign code in your core logic and writing it all by yourselves, since it enables you to understand every little detail and doesn't introduce any unnecessary risks.
During the manual code review we did not find any alarming bugs or flaws. Merely the comments could be improved.
Our protocol and logic analysis did show five possible flaws. While none of them are posing a threat to the users funds, we see especially **3.3.A** ("No project check for delegate voter") as something that needs to be fixed before launching the dao.care product to mainnet. Still, the rest of these findings need to be taken into account and decided upon by the dao.care team.

5. Update on the 26th of May 2020

Since we sent our report to the dao.care team, the findings have been discussed in a bi-lateral meeting.The team of dao.care addressed our findings and the implementation of these mitigations in this pull request on GitHub. Overall, the following of our found flaws have been addressed:

2.1. Manual Code Review - Line 209
The comment has been corrected.

2.2. Manual Code Review - Line 263
The error allowing for a slight remainder in the division has been fixed.

A) No project check for delegate voter (NoLossDao_v0.sol)
The necessary check (in the form of a "require") has been added.

B) Distribute Funds allows to frontrunning (NoLossDao_v0.sol)
As stated in the report and pointed out by the team, they are aware of this, as it is part of their design and is intended to be this way.

C) Possible flaw in the Cooldown flagging mechanism (NoLossDao_v0.sol)
The dao.care team is not worried about this, as they don't think, that making use of this will be socially possible. We agree with this.

D) Current weeks prize money goes to last weeks' winner
The dao.care team understands our concerns. However, they intended this feature the way it is implemented, in order to allow for "rage quits", which is understandable. We agree with this.

E) Vote is public and prone to potentially be manipulated by whales
The dao.care team stated that adding a concealment mechanism for the votes would add a level of complexity to the system that they are not looking to introduce during this iteration. We agree with this.

F) Fixed 200k limit for emergency function (PoolDeposits.sol)
The dao.care team states, that they like the large limit, but they introduced a new function that allows an additional set of admin keys to immediately declare the state of emergency. As this ensures, that in the case of an "early failure" of any of the systems everyone will get their money back as soon as the team declares the state of emergency, we think that this is a good way to handle it.

Additional features that have been added
Two major additional features or modifications have been implemented since we audited the code in the context of this pull request: 1) Upgrade to Solidity version 0.6.10 and 2) the ability to partially deposit and withdraw funds. The main reason for (1) is to account for failures in the aDai contract, which is now being handled in

a try catch manner. We checked the pull request that contained these changes and were not able to find any issues that have been introduced this way. We further used an automated code checking software (MythX) to verify that we haven't missed anything.