xDao Security Analysis

by Pessimistic

This report is public.

Published: May 7, 2021

Abstract

In this report, we consider the security of smart contracts of xDao project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audit is not an investment advice.

Summary

In this report, we considered the security of xDao smart contracts. We performed our audit according to the procedure described below.

The audit showed two critical issues: Reentrancy and Double-spending, many issues of medium and low severities.

The project has documentation and tests.

After the initial audit, the code base was updated. Some issues were fixed, some other issues were commented.

After the recheck #1, update #2 was made. **xDao** and **Minter** contracts were removed from the code base, one issue in **Dac** contract was fixed.

The developers do not follow best practices, and the resulting code quality is below average. Multiple issues of medium severity have not been fixed, as developers plan to address them in future versions.

General recommendations

We recommend improving the design of the system. Many complex solutions of the project have well-tested implementations in other projects. Some examples are mentioned further in the report. Consider adopting these solutions.

We also recommend fixing the rest of issues, following best practices, and adding CI to improve the quality and security of the code in the future.

Project overview

Project description

For the audit, we were provided with xDao project on a public GitHub repository, commit 53cdef20782162bd91c6a2173562a8ea3d2033b1.

The documentation for the project is stored on GitBook platform.

The project compiles successfully and has tests, the coverage is 62.42%.

The total LOC of audited sources is 1033.

Update #1

After the initial audit, the code base was updated. For the recheck, we were provided with commit 15024a691c3a11569c8765838969b12fa5750d13.

The documentation to the project was moved to docs.xdao.app website.

Update #2

After the recheck #1, the code base was updated once again. For the recheck #2, we were provided with commit b0bc6c48c6ee7743646c53f639c6be8efbd1a0d0.

Procedure

In our audit, we consider the following crucial features of the code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).
3. Whether the code meets best practices.

We perform our audit according to the following procedure:

- Automated analysis
  - We scan project's code base with automated tools: Slither and SmartCheck.
  - We manually verify (reject or confirm) all the issues found by tools.
- Manual audit
  - We manually analyze code base for security vulnerabilities.
  - We assess overall project structure and quality.
- Report
  - We reflect all the gathered information in the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger smart contracts security. We highly recommend fixing them.

Reentrancy (fixed)

`burnGovernanceTokens()` function of **Dac** contract does not follow CEI pattern and is therefore susceptible to reentrancy attack. As a result, an attacker can retrieve all the assets stored on a contract without paying governance tokens.

An attacker can provide `_tokens` list that can include an arbitrary address. If such an address is a contract, it can re-enter **Dac** contract when called form `burnGovernanceTokens()` function. As the balance of the caller is updated only after the call is made, a malicious contract can recursively initiate assets withdrawal multiple times, and each of these calls will send assets to an attacker.

The malicious contract can also call `Dac.transfer()`, so its Dac balance becomes `0`. However, at lines 473–475 of `burnGovernanceTokens()` function it has already become `0`, therefore no tokens are burned.

We recommend following CEI pattern and allowing only whitelisted assets.

The issue has been fixed and is not present in the latest version of the code.

Double-spending (fixed)

A user can call `burnGovernanceTokens()` function of **Dac** contract with `_tokens` argument where the same token occurs multiple times. As a result, the user will receive token multiple times.

Consider checking for duplicates in `_tokens`.

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium issues can influence project operation in current implementation. We highly recommend addressing them.

Bad voting design

In `signVoting()` function of **Daf** contract, a user who already voted can easily bypass the checks at lines 244–246 and 305–307 by transferring tokens to a new address.

When the number of signers of `votingsAddToWhitelist[_index]` vote is close to two thousand, `signVotingAddToWhitelist()` and `activateVotingAddToWhitelist()` functions will fail with to `Out of Gas` error. Thus, any user with a certain amount of Daf tokens and enough ether can block any proposal.

Since proper governance and voting designs are hard to implement, we recommend using existing solutions.

> *Comment from developers: That is okay for us.*

ERC20 standard violation

- The returned values of `ERC20.transfer()` and `ERC20.transferFrom()` functions are not checked. This violates EIP-20 MUST requirement:

```
Callers MUST handle false from returns (bool success). Callers MUST
NOT assume that false is never returned!
```

- According to EIP-20 specification, `decimals` must be of `uint8` type.
- According to EIP-20 specification, token transfers MUST trigger `Transfer` events. Consider calling `transfer()` function internally.

BEP20 standard violation

BEP20 standard requires tokens to implement `getOwner()` function. However, it is not implemented in the code.

> *Comment from developers: We will skip this for **DAC/DAF** contracts, we do not plan to flow across bc-bsc for our DAOs.*

Discrepancy with the documentation

According to the documentation, **ServiceDao** and **Dac** are almost identical, the only difference is that **ServiceDao** contract does not have tokens. However, in **ServiceDao** contract, a privileged vote (i.e., `goldenShare`) is required to approve any decision.

Low severity issues

Low severity issues can influence project operation in future versions of code. We recommend taking them into account.

Code quality

- We recommend separating token, sale, and governance functionality for **Dac** and **Daf** contracts, and thus splitting each of these contracts into three.

  > *Comment from developers: That is okay for us.*

- In **Dac**, **Daf**, and **ServiceDao** contracts, some arguments for `VotingCreated`, `VotingSigned`, and `VotingActivated` events should be indexed, e.g. `signer` address, vote `index`.

  The issues have been fixed and are not present in the latest version of the code.

- The checks for even numbers are redundant since the following check will work fine for any number:

```
require(votings[_index].signers.length > (teammates.length / 2));
```

  - In **Dac** contract at line 255

    The issue has been fixed and is not present in the latest version of the code.

  - In **ServiceDao** contract at line 109

- The usage of `governanceTokensPrice` variable is inconsistent in `buyGovernanceTokens()` functions of **Dac** and **Daf** contracts.
- Using hardcoded addresses without appropriate names is error-prone and hinders readability.
- Consider declaring functions as `external` instead of `public` where possible.

  The issue has been fixed and is not present in the latest version of the code.

Code logic

- In `transferOfRights()` function of **Dac** and **ServiceDao** contracts, if `_newTeammate` is already in the team, the function will add the existing address to `teammates` anyway. Consider checking if `_newTeammate` is present in `teammates`.

  The issues have been fixed and are not present in the latest version of the code.

- Users who buy significant amounts of tokens can buy more tokens for the same price due to inaccurate rounding in `burnGovernanceTokens()` function of **Dac** contract.

  The issue has been fixed and is not present in the latest version of the code.

- There is a logical inconsistency in **Dac** contract: when a user burns tokens with `burnGovernanceTokens()` function, this user is removed from `teammates`. However, he/she is not added to this list when buying tokens. Note that a user can only buy tokens if `purchasePublic` option is enabled or if this user is a member of `teammates`.

> *Comment from developers: That is okay. First of all, you should be a teammate to buy tokens, not in reverse order.*

Gas consumption

- Consider declaring the visibility of `threeYearsLockExprired` variable explicitly and making it `internal` and `immutable`.

  The issues have been fixed and are not present in the latest version of the code.

- Reading `.length` property of an array costs gas. Therefore, consider storing it to a local variable when using it to iterate through arrays. This makes more sense when iterating through arrays from storage.

> *Comment from developers: That is okay for us.*

- Iterating through `teammates` and `signers` arrays is impractical for larger teams.

  We recommend replacing these arrays with `mapping(address => bool)` and `uint` counter pairs. This will consume the same amount of gas or less, while uniqueness and `goldenShare` checks will be much cheaper. However, in this case, you will need to compute the arrays off-chain.

> *Comment from developers: That is okay for us.*

Notes

Voting design

External conditions, like quorum or duration, for voting may change on-the-fly du to other proposals in **Dac** and **Daf** contracts.

> *Comment from developers: That is okay for us.*

This analysis was performed by Pessimistic:

Vladimir Tarasov, Security Engineer
Daria Korepanova, Security Engineer
Evgeny Marchenko, Senior Security Engineer
Boris Nikashin, Analyst
Alexander Seleznev, Founder

May 7, 2021

Appendix

Issues from xDao and Minter contracts (removed)

In update #2, **xDao** and **Minter** contracts were removed from the code base. Therefore, all the issues found in these contracts are considered obsolete. All the comments are kept for historical reason.

Price manipulation

In **Minter** contract, users can manipulate the price of assets on Pancake exchange to buy tokens at a lower price. Here is a simple example of the attack: a user buys significant amount of ether on ETH-BUSD pool, then buy tokens using `Minter.buyToken()` function effectively at a lower cost, and then sell ether back to Pancake.

We recommend using solutions like Uniswap's TWAP or a reliable off-chain oracle.

Imprecise assets calculation

Consider using `PancakeRouter.getAmountIn()` function to get precise amount of required asset in `buyToken()` function of **Minter** contract.

> *Comment from developers: That is okay for us.*

Bug

Expression at line 69 of **Minter** contract does not consider `ERC20.decimals` and therefore can fail due to built-in underflow check.

> *Comment from developers: It is correct if you provide full amount including all decimals.*

Incorrect calculation (fixed)

In `reverseConversion()` function of **Minter** contract, `_share` value is calculated incorrectly: `_xDAO.totalSupply() - _xDAO.balanceOf(address(this))` is equal to the number of all xDao tokens that were sent to users, including `msg.sender` balance. Thus, the function uses the user's balance twice which can result in an error. E.g., if all the tokens belong to a single user, `_share` will be equal to `2` instead of `1`.

The issue has been fixed and is not present in the latest version of the code.

Code quality

- In **xDAO** contract, consider declaring variables at lines 5–11 as `constant`s.

  The issue has been fixed and is not present in the latest version of the code.
- **xDAO** contract should inherit from **IERC20**.
- `reverseConversion()` function of **Minter** contract violates CEI pattern.