Dedaub

Security Technology for Smart Contracts

Armor Governance

Smart Contract Security Assessment

Date: May 08, 2021

Abstract

Dedaub was commissioned to perform a security audit of the Armor Governance contracts. The audit is explicitly about the code. Two auditors worked over this codebase over a week.

Setting and Caveats

The code base is modest in size and is loosely based on governance concepts borrowed from other protocols (notably Compound), with significant adaptation. Governance is meant to interact with the entire Armor protocol, which is not trivial and Dedaub has already audited significant chunks of. The interactions between these two projects may therefore lead to complex issues. In addition, the audit was conducted on a codebase that has not undergone significant testing. Although some functional correctness issues can be uncovered by an audit, this audit is meant to evaluate the security of the project. It therefore is recommended that the developers perform more testing before deploying this contract to the Ethereum network.

Vulnerabilities and Functional Issues

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
Critical	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
High	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
Medium	Examples: 1) User or system funds can be lost when third party systems misbehave. 2) DoS, under specific conditions. 3) Part of the functionality becomes unusable due to programming error.

Low	Examples: 1) Breaking important system invariants, but without apparent consequences. 2) Buggy functionality for trusted users where a workaround exists. 3) Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed”, by the client, or “resolved”, per the auditors.

Critical Severity

Id	Description	Status
C1	**Logical error in `armorToVArmor()` calculation**	**Resolved**

In contract vArmor.sol function `armorToVArmor()`:

```
function armorToVArmor(uint256 _armor) public view returns(uint256) {
      if(totalSupply() == 0){
            return _armor;
      }
      return _armor * totalSupply() / _armor;
}
```

always returns totalSupply in the regular case, while could return something completely different if `_armor` is 0 or perhaps `totalSupply` doesn't fit in 128 bits

High Severity

Id	Description	Status
H1	**Multisig may pass proposals without any restriction**	**Open**

The protocol defines that the multisig can immediately queue a proposal surpassing the voting procedure of the DAO. However, the DAO should be able to cancel the proposal at any time during the timelock period, since neither the DAO nor the multisig should have any way to execute a transaction without the other having the chance to stop it from executing. It is not clear under which circumstances canceling of a multisig’s proposal should be possible though.

Function `ArmorGovernance::cancel()` proceeds only if the proposer is below threshold for votes.

```
function cancel(uint proposalId) public {
      ProposalState state = state(proposalId);
      require(state != ProposalState.Executed, "GovernorAlpha::cancel:
cannot cancel executed proposal");

      Proposal storage proposal = proposals[proposalId];
      proposal since they can directly call timelock.cancelTransaction()
      require(varmor.getPriorVotes(proposal.proposer,
sub256(block.number, 1)) < proposalThreshold(block.number - 1),
"GovernorAlpha::cancel: proposer above threshold");

      proposal.canceled = true;
      for (uint i = 0; i < proposal.targets.length; i++) {
            timelock.cancelTransaction(proposal.targets[i],
proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
proposal.eta);
      }

      emit ProposalCanceled(proposalId);
}
```

This is questionable for the multisig, since they most probably hold - by principle - more votes than the defined threshold.

Medium Severity

Id	Description	Status
M1	**Insecure math operations**	**Resolved**

In contract vArmor.sol functions `vArmorToArmor()` and `armorToVArmor()` perform numerical operations without checking for overflow.
In `vArmorToArmor()` overflow of multiplication is not checked:

```
function vArmorToArmor(uint256 _varmor) public view returns(uint256) {
      if(totalSupply() == 0){
            return 0;
      }
      return _varmor * armor.balanceOf(address(this)) / totalSupply();
}
```

Similar for `armorToVArmor()`.
These functions are called during `deposit` and `withdraw` for calculating token amounts to be transferred, so erroneous results will have a significant impact on the correctness of the protocol.

M2	**DoS by proposing proposals that need to be voted out quickly**	**Open**

Any governance token holder can DoS their peers by proposing many unfavorable proposals, which need to be voted out. Voting proposals out will incur more gas fees as these are subject to a deadline (and may be voted down by multiple participants) whereas a proposer can also wait for the optimal time to spend gas.

Low Severity

Id	Description	Status
L1	**Admin and gov privileged users not checked for address zero**	**Open**

In Timelock.sol the addresses of gov and admin are set during the construction of the contract. Requirements for checking non-zero addresses is suggested.

L2	**tokenHelpers introduce opportunities for reentrancy during swaps**	**Open**

In vArmor.sol, governance through a simple proposal can add tokenHelpers that are executed whenever a token transfer takes place. Token transfers also take place during swaps or other activities like deposits or withdrawals. The opportunity for reentrancy may not be immediately visible but if this were to be possible, consequences may include the draining of LP pool funds.		
L3	**Proposer can propose multiple proposals (Sybil attack)**	**Open**
A proposal can propose multiple proposals at the same time, defeating checks to disallow this: 1) Deposit enough $armor in the vArmor pool 2) Propose a proposal 3) Withdraw $armor from vArmor pool 4) Transfer $armor to a different address 5) Repeat The protocol offers the `function cancel(uint proposalId) public` to mitigate this attack, which proceeds in canceling a proposal if the proposer's votes have fallen below the required threshold. However, this requires some users or the mutlisig to constantly be in a state of readiness.		

Other/Advisory Issues

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing.

Id	Description	Status
A1	**Inconsistent type declarations**	**Open**
In contract `ArmorGovernor.sol` the parameters of several functions are declared as uint256, whereas most numerical variables are declared as uint. We suggest that a single style of declaration is used for clarity and consistency.		
A2	**Inconsistent code style regarding subtractions**	**Resolved**
In contract `ArmorGovernor.sol` functions cancel() and propose() include same subtraction operation (block.number - 1) twice but with slightly different implementation. One is executed immediately, while the other uses a safety checking function `sub256()`. In `propose()`:		

```
require(varmor.getPriorVotes(msg.sender, sub256(block.number, 1)) >
proposalThreshold(block.number - 1),
```

Similar in `cancel()`.
Underflow seems unlikely in this case, however we suggest that all subtractions are performed in the same way for consistency.

A3	Typo errors in error messages	Partially resolved (error in AcceptGov() remains)

In contract `Timelock.sol` functions `acceptGov()` and `setPendingGov()` contain a typo in the error messages of a requirement. In `acceptGov()`:

```
require(msg.sender == address(this), "Timelock::setPendingAdmin: Call
must come from Timelock.");
```

Should become:

```
require(msg.sender == address(this), "Timelock::setPendingGov: Call must
come from Timelock.");
```

Similar for `setPendingGov()`.

A4	Wrong event emitted	Resolved

In contract Timelock.sol the function setPendingGov() emits a wrong event.

```
emit NewPendingAdmin(pendingGov);
```

Should become

```
emit NewPendingGov(pendingGov);
```

A5	Incomplete error messages	Resolved

In contract `Timelock.sol` the functions which are `admin-` or gov-only refer only to admin when it comes to authorization-related error messages.
For example, in function `queueTransaction()`

```
require(msg.sender == admin || msg.sender == gov,
"Timelock::queueTransaction: Call must come from admin.");
```

Similar for functions `cancelTransaction()`, `executeTransaction()`.
We suggest that the error messages are extended to include gov as well.

A6	Unintuitive code reuse	Info

In contract `vArmor.sol` the `Checkpoint` struct is used to record both account votes (storage variable `checkpoints`) and the total token supply (storage variable `checkpointsTotal`) while the struct field is named `votes`, making the code slightly harder to follow. For example, in function `_writeCheckpointTotal` we inspect the following

```
checkpointsTotal[nCheckpoints - 1].votes = newTotal;
```

A7	Floating pragma	Info

Use of a floating pragma: The floating pragma `pragma solidity ^0.6.6;` is used in the `Timelock` contract allowing it to be compiled with any version of the Solidity compiler that is greater or equal to v0.6.6 and lower than v.0.7.0. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.
ArmorGovernance contract uses `pragma solidity ^0.6.12;` which can be altered to the identical and simpler `pragma solidity 0.6.12;`.

Disclaimer

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness status of the contract. While we have

conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

About Dedaub

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.