PUBLIC

Code Assessment

of the Serpentor Smart Contracts

September 27, 2022

Produced for

yearn

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Yearn team,

Thank you for trusting us to help Yearn with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Serpentor according to Scope to support you in forming an opinion on their security risks.

Yearn provides an alternative implementation of Compound's GovernorBravo in the Vyper programming language.

The most critical subjects covered in our audit are Ether Not Transferred in Proposals, External Call From Timelock With Wrong Payload, Incorrect Calldata ABI Encoding When Signature Is Not Empty, Version Not Included in Domain Separator, Timelock.executeTransaction() Returns Incorrect Value and Timelock.executeTransaction() reverts if trx.signature is not empty . The number of critical issues found is high. Many of the issues would have been caught with proper testing. The issues were resolved, and the testing improved. Nonetheless, we recommend extensive integration test before deployment with additional safety measures.

Given all raised issues have been fixed and testing improved, we rate the current security of the project sufficient. But as mentioned before, it is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	6
• Code Corrected	6
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	2
• Code Partially Corrected	1
• No Response	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Serpentor repository based on written communications. The following files were part of the assessment:

1. `src/SerpentorBravo.vy`
2. `src/Timelock.vy`

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	01 September 2022	452f9dd1409bd529cf834926dafbd1d2746a656f	Initial Version
2	13 September 2022	ed614c11cdc9b5ba2ca4a0b7a404a8ce7a7ce3f3	Version 2

For the Vyper smart contracts, the compiler version `0.3.6` was chosen.

2.1.1 *Excluded from scope*

Any contract inside the repository that are not mentioned in `Scope` are not part of this assessment. In particular, deployment scripts and tests are not part of the scope.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Yearn developed `SerpentorBravo`, a system for DAO governance written in `Vyper`, replicating the functionality of `GovernorBravo`, the time proven governance solution developed by Compound.

`SerpentorBravo` handles DAO governance through three contracts:

1. a governance token, which implements the `getPriorVotes()` interface. Out of scope for this audit.
2. contract `SerpentorBravo`, which handles the submission and voting on governance proposals.
3. contract `Timelock`, which handles the execution of succesful proposals in a safe way.

Owners of the governance token can participate to the governance of the DAO. Proposals of action to be taken by the DAO can be submitted by any user whose voting power exceeds a minimal anti-spam threshold. Governance token holders then can vote on submitted proposals for a fixed amount of time, after which the proposal is either successful or defeated. Successful proposals are queued in the timelock, and after a fixed amount of time they can be executed. The execution of a proposal consists of a series of arbitrary external contract calls from the timelock. Governance is implemented by setting administrative rights of contracts under the authority of the DAO to the address of the timelock, so that successful proposals can execute changes in the system.

2.2.1 *Contract* `SerpentorBravo`

Contract `SerpentorBravo` handles the interaction of voters with the governance process. Governance is implemented as the execution of proposals, conditional on the success of votes held on the proposals by holders of the governance token.

Proposals are submitted through calls to the `propose` function. Callers holding more than `proposalThreshold` votes or who are whitelisted are allowed to submit new proposals. A proposal consists in a list of arbitrary external calls to be performed from the `Timelock`.

When a proposal is submitted, its state is `PENDING` for a `votingDelay` amount of blocks. This `votingDelay` allows voters to reorganize voting power before voting starts.

After `votingDelay` blocks have elapsed, the proposal becomes `ACTIVE`. Users can vote on an active proposal by calling functions `vote` and `voteWithReason`. Votes on behalf of other users can be cast by using `voteBySig()`, and providing a valid signature for the voter. Voting happens during `votingPeriod` blocks, during which voters cast votes for a proposal in favor, against, or explicitly abstaining. Voting power of users is evaluated at the block prior to the beginning of voting.

After `votingPeriod`, the proposal is either `SUCCEEDED`, if it has more votes in favor than against, and the favorable votes exceed `quorumVotes`, or `DEFEATED` otherwise. No more actions can be performed on `DEFEATED` proposals. `SUCCEEDED` proposals can be queued for execution in `Timelock` through a unpermissioned call to function `queue`. The list of actions to be performed by the proposal is queued for execution by external calls to `Timelock.queueTransaction()`. `Proposal.eta`, the timestamp after which the proposal can be executed, is set to the current time plus the `Timelock.delay()`.

Once `queue()` has been called on a successful proposal its state becomes `QUEUED`. Once `block.timestamp` is more than the `eta` of the proposal, `execute()` can be called to perform the list of actions contained in the proposal. The proposal therefore becomes `EXECUTED`. If after `eta` no successful call to `execute()` happens for `Timelock.GRACE_PERIOD`, the proposal is not executable anymore and becomes `EXPIRED`.

A proposal can be cancelled by the proposer up to its execution. If the proposer's voting power falls below the `proposalThreshold`, anybody can cancel it, unless the proposer is whitelisted.

The whitelist exists to allow privileged accounts to submit proposals for voting even without the necessary voting power. `SerpentorBravo` has a privileged role called `queen`, equivalent to that of an administrator. Its capabilities are of changing voting parameters such as `votingDelay`, `votingPeriod`, `proposalThreshold`, and adding whitelisted accounts and granting the `knight` role. The role called `knight` allows to cancel proposals of whitelisted users that fell below the proposal threshold and can whitelist other users.

2.2.2 *Contract* `Timelock`

Contract `Timelock` is the actual proposal executor, and as such it holds the administrative rights to other contracts under the authority of the governance. `Timelock` has a single privileged user, `queen`, which can queue and execute transactions in the `Timelock`. Typically, `queen` is set to the governance contract `SerpentorBravo`.

The `queen` user can queue a transaction, by calling `queueTransaction()`. If the `eta`, the time at which the transaction can be executed, is bigger than the current time plus `delay`, the transaction's hash is stored as queued. Otherwise, the transaction is rejected. This forces a minimum `delay` amount of time to pass between the submittal of a transaction and its execution.

When `block.timestamp` becomes greater than `Transaction.eta`, `queen` can execute a transaction by calling `executeTransaction()`. `Transaction.target` is called with the specified payload data and `Transaction.amount` ether value.

If `GRACE_PERIOD` amount of time elapses between `Transaction.eta` and the calling of `executeTransaction()`, the transaction is stale and no longer executable.

2.2.3 Differences to Compound's implementation

Serpentor aims to replicate in the Vyper programming language Compound's governor functionality. There are a few differences of which to be aware:

1. Compound implements the Governor as a couple of contracts: the `GovernorBravoDelegator` proxy and the `GovernorBravoDelegate` logic contract. Serpentor implements the governor as a single contract `SerpentorBravo`. Deployment is different since it is call to constructor instead of constructor and then initialization function.
2. The Governors `state` function return different values for equivalent proposal states in GovernorBravo and Serpentor. This is due to Vyper internal enum representation. `ordinalState` is equivalent to GovernorBravo's `state`.
3. The domain separator of Serpentor includes `version`, which is not in GovernorBravo.
4. The bit size of some `uint` values is different. In Serpentor votes are `uint256`, while they are `uint96` in GovernorBravo.
5. The storage locations are not the same in GovernorBravo and Serpentor.
6. naming choices: the `data` argument of proposals and transaction in GovernorBravo has been renamed to `callData` in Serpentor. `castVote*` functions have been renamed to `vote*`. `proposalMaxOperations` -> `proposalMaxActions`. `queen` & `knight` instead of `admin` and `whitelistGuardian`
7. Some function's interfaces are not preserved identical.
8. Some values that are constants in GovernorBravo (`quorumVotes`, `proposalMaxOperations`) are storage variables in Serpentor
9. The value of constant `MAX_PROPOSAL_THRESHOLD` has been reduced.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

- Sanity Check in Timelock.cancelTransaction
- Gas Optimizations Code Partially Corrected

5.1 Sanity Check in `Timelock.cancelTransaction`

Design Low Version 1

When `Timelock.cancelTransaction` is called, `queuedTransactions[trxHash]` is set to `False` without doing a sanity check if it was `True` before.

5.2 Gas Optimizations

Design Low Version 1 Code Partially Corrected

The following gas optimization are possible, some of them quite impactful:

- Full `Proposal` struct is copied from storage to memory, even if only one field is used. Reading a `Proposal` from storage takes at least 17 `SLOAD` operations, and more depending on how many actions are present. In `queue()`, `execute()`, `cancel()`, `getActions()`, the full `Proposal` is loaded but only `.actions` is used. In `_vote()`, the full proposal is loaded only to access `.startBlock`. in `_state` the full proposal is loaded, however it is only incrementally accessed and could be short circuited.
- Full `Receipt` struct copied to memory (3 `SLOAD``s), only to access only ``.hasVoted` in `_vote`.
- `Receipts` take 3 storage slots, if `.votes` could be restricted to `uint240` then the struct could only take 1 storage slot with some manual byte packing.
- `quorumVotes and proposalMaxActions` are defined as mutable storage variables instead of constants. Since they are only set in the constructor, could be immutable variables with appropriate getter.

Code partially corrected

The following changes have been made:

- The storage loads for Proposal``s has been optimized except for in the ``_state function. There self.proposals[proposalId].forVotes is accessed twice.
- The _vote function now calls _getHasVoted which only accesses .hasVoted from the Receipts struct.
- The Receipts struct is not changed.
- quorumVotes and proposalMaxActions are now public getter functions, backed by immutable variables set in the constructor, instead of public storage variables.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	6

- Ether Not Transferred in Proposals Code Corrected
- External Call From Timelock With Wrong Payload Code Corrected
- Incorrect Calldata ABI Encoding When Signature Is Not Empty Code Corrected
- Version Not Included in Domain Separator Code Corrected
- Timelock.executeTransaction() Returns Incorrect Value Code Corrected
- Timelock.executeTransaction() Reverts if trx.signature Is Not Empty Code Corrected

Medium-Severity Findings	1

- Can't Set Timelock as SerpentorBravo Queen at Deployment Code Corrected

Low-Severity Findings	0

6.1 Ether Not Transferred in Proposals

Correctness High Version 1 Code Corrected

Actions in `SerpentorBravo` and Transactions in `Timelock` have the `value` field, however `Timelock.executeTransaction()` is not `@payable`, and `Timelock` lacks a payable `__default__()` function (fallback), so there is no way to transfer Ether to proposal execution.

The `raw_call` invocation in `Timelock` uses the `value` keyword, as a consequence, if any `ProposalAction.value` is set to something else than 0, the proposal execution will revert since no `value` can be transferred.

Code corrected

The Timelock fallback function `__default__()` has been defined `payable`, as well as the function `executeTransaction`.

6.2 External Call From `Timelock` With Wrong Payload

Correctness High Version 1 Code Corrected

In `Timelock.executeTransaction()`, `raw_call()` is passed `trx.callData` as the `data` argument, when the local variable `callData` should be passed instead.

Code corrected

`callData` is now passed into `raw_call` as call data argument instead of `trx.callData`.

6.3 Incorrect Calldata ABI Encoding When Signature Is Not Empty

Correctness High Version 1 Code Corrected

In `Timelock.executeTransaction()`, when `signature` is not empty, `callData` is defined as:

```
callData = _abi_encode(func_sig, trx.callData)
```

However, `_abi_encode()` is incorrect in this context, since `func_sig` will be right padded to 32 bytes, and `trx.callData` will be encoded as a dynamic array, by prepending to it the offset of the data and the length of the bytes array. What is actually needed is the concatenation of the 4 bytes `func_sig` and `trx.callData`.

The calldata passed to `raw_call` is the concatenation of the `func_sig` and `trx.callData` using `concat()`.

6.4 Version Not Included in Domain Separator

Correctness High Version 1 Code Corrected

In the computing of the domain separator, the contract's `EIP712Domain` structure is defined and hashed as

```
DOMAIN_TYPE_HASH: constant(bytes32) = keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)')
```

to include `name`, `version`, `chainId`, and `verifyingContract` address.

Domain separator however is computed as

```
def _domainSeparator() -> bytes32:
    return keccak256(
        concat(
            DOMAIN_TYPE_HASH,
            keccak256(convert(ChainSecurity, Bytes[20])),
            convert(chain.id, bytes32),
            convert(self, bytes32)
        )
    )
```

Either the hash of the version should be included in the domain separator for it to be valid, or version omitted from the `DOMAIN_TYPE_HASH`.

Code corrected

The `_domainSeparator` now includes the version as `keccak256("1")`.

6.5 Timelock.executeTransaction() Returns Incorrect Value

Correctness High Version 1 Code Corrected

executeTransaction() should return the returndata of the external call, but it returns callData instead. max_outsize in raw_call is set to 32 bytes, and response is capped to 32 bytes, which is insufficient for generic return data sizes.

Code corrected

max_outsize is now set to MAX_DATA_LEN which is 16608 and executeTransaction returns the response object from the raw_call.

6.6 Timelock.executeTransaction() Reverts if trx.signature Is Not Empty

Correctness High Version 1 Code Corrected

if trx.signature is not empty, the function selector is computed as:

```
sig_hash: bytes32 = keccak256(trx.signature)
func_sig: bytes4 = convert(sig_hash, bytes4)
```

However, convert(sig_hash, bytes4) reverts as no bytes32 -> bytes4 cast is defined. sig_hash must first be reduced to the appropriate length with the slice() built-in.

Code corrected

The bytes32 object sig_hash is first cut to 4 bytes by slice and then passed into convert.

6.7 Can't Set Timelock as SerpentorBravo Queen at Deployment

Design Medium Version 1 Code Corrected

After a proper governor deployment, the timelock should be set as the governor's admin and the governor as the timelock's admin, assuring that changes to either system can only happen through a governance proposal.

However, in SerpentorBravo, queen is always set to msg.sender at contract creation time. To set Timelock as queen, the contract deployer needs to go through the lengthy process of setting Timelock as pendingQueen, then create a proposal for Timelock to call acceptThrone(), hold a successful vote, and execute the proposal.

This procedure is more inconvenient than what happens in GovernorBravo, which allows Timelock to be set as the Governor's admin at deployment time.

Code corrected

The Vyper initiation function of the `SerpentorBravo` contract now allows to pass in the address of the queen as argument and sets it when the contract is deployed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 API Differences With GovernorBravo

Note Version 1

Serpentor implements a number of small changes from Compound's GovernorBravo that result in a different API. Different function names, different value for `ProposalState`, and different event names will result in incompatibility with some existing governance operation platforms, such as Tally