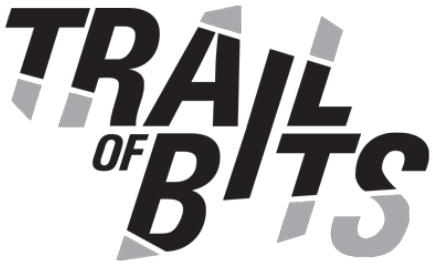

The Computable Protocol

Security Assessment

August 30, 2019

Prepared For:
Bharath Ramsundar | Computable
bharath@computable.io

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Rajeev Gopalakrishna | *Trail of Bits*
rajeev@trailofbits.com

Executive Summary

From July 8 through August 2, 2019, Computable engaged Trail of Bits to review the security of their smart contracts. Trail of Bits conducted this assessment over the course of eight person-weeks, with three engineers working from commit hash `1502a7993d914047401fb3a7a3ee5d4dcfae2c7a` from the `computable` repository. Computable developed the contracts using Vyper, a relatively new contract-oriented, pythonic programming language, which targets the Ethereum Virtual Machine. While Vyper aims to reduce the number of security issues developers could inadvertently create, the code produced with it should be carefully reviewed. Vyper is still an early and rapidly evolving language.

During the first week, we began to familiarize ourselves with the smart contracts used by the Computable system and performed a full review of ERC20 token contracts. We dedicated the second week to investigating the voting procedure and the interactions of the Reserve with the rest of the contracts. We focused the third week's on the delivery system and the more complex interaction with the voting contracts. For the final week of the assessment, Trail of Bits concluded the manual review of the Computable contracts. We also evaluated potential approaches to overcome trust issues in the system (as detailed in Appendix E) and developed a small Python library to test some of the key findings using Manticore (as detailed in Appendix F).

Trail of Bits identified 22 issues, ranging from low to high severity:

- A large number of the issues relate to voting process subversion, including disrupting new candidate addition, stake blocking, and vote manipulation.
- Another source of issues is the code related to deliveries. One issue concerns incorrect price computation, and the other results from the amount of bytes to deliver, which can impede users from obtaining their requested data or trap their tokens in the contracts.
- Other issues arise from the volatility of market parameters, which can change at any time, producing unexpected prices.
- The remaining issues relate to the ERC20 standard, missing checks in privileged functions, and excessive permissions in the token owner accounts.

The overall quality of the code base is good. The architecture is suitable and avoids unnecessary complexity. Component interactions are well defined. The functions are small, clean, and easy to understand.

However, this codebase frequently suffers from front-running, denial-of-service, and other high-severity issues. The contracts also depend heavily on the system parameters. These are not properly validated and can be changed at any time, which can lead to different

attacks if the market votes to use incorrect values. The voting procedure contained the majority of the issues would benefit from a refactoring of its hash schema.

Trail of Bits highly recommends that Computable fixes these issues before deploying this code in production. The proposed fixes should be carefully considered on a one-by-one basis, since some of them require changes in the design architecture and could affect other system components.

Project Dashboard

Application Summary

Name	Computable
Version	`1502a7993d914047401fb3a7a3ee5d4dcfae2c7a`
Type	Vyper Smart Contracts
Platforms	Ethereum

Engagement Summary

Dates	July 8 - August 2, 2019
Method	Whitebox
Consultants Engaged	3
Level of Effort	8 person-weeks

Vulnerability Summary

Total High-Severity Issues	11	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Total Medium-Severity Issues	5	■ ■ ■ ■ ■
Total Low-Severity Issues	8	■ ■ ■ ■ ■ ■ ■ ■
Total	22	

Category Breakdown

Data Validation	11	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Input Validation	1	■
Undefined Behavior	1	■
Denial of Service	3	■ ■ ■
Access Control	1	■
Timing	5	■ ■ ■ ■ ■
Total	22	

Engagement Goals

Trail of Bits and Computable scoped the engagement to provide a security assessment of the Computable protocol smart contracts in the `contracts` repository.

Specifically, we sought to answer the following questions:

- Can participants abuse the market trust model?
- Can participants delay, block, or unfairly influence voting?
- Is it possible to manipulate the market by using specially crafted parameters or front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Can participants perform denial-of-service or phishing attacks against any of the contracts?
- Are there potential or concrete Vyper-specific issues?

Coverage

This review included the seven contracts comprising the Computable protocol. The specific version of the codebase used for the assessment came from commit hash `1502a7993d914047401fb3a7a3ee5d4dcfae2c7a` of the `computable` Github repository.

Trail of Bits reviewed contracts for common design flaws, such as market and voting manipulations, front-running, denial-of-service, race conditions, and trust breaches. They were also reviewed for Vyper-specific flaws, such as dangerous type-conversions and unexpected integer overflow reverts.

Market manipulation. Participants may manipulate market parameters, time their interactions, or collude with other participants to profit from crypto-economic incentives underlying the data markets. In reviewing the design and implementation, we assumed any participant may be untrustworthy and malicious.

Voting manipulation. The design leverages voting mechanism to let participants approve or challenge data listings and providers. Because blockchain-based online voting is susceptible to front-running, race-conditions, and denial-of-service, we analyzed all of these vulnerabilities in the contracts.

Access controls. Many parts of the system expose privileged functionality, such as minting new tokens or managing candidates. We reviewed these functions to ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges.

Arithmetic. We reviewed these calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the contract.

Some specific types of issues outside the scope of this assessment are:

EVM-generated code: since the Vyper compiler is still under development, the generated EVM bytecode should be carefully reviewed to make sure it matches the high-level Vyper code. However, we did not have enough time to perform this analysis.

Voting effectiveness: Computable protocol adopts a voting mechanism to allow Makers and Backends to propose their services, while allowing other market participants to challenge these proposals with a stake. This voting is managed on the blockchain for transparency. Voting on blockchain has several challenges, such as front-running and voting manipulation, via delays and timing attacks. This is not specific to the Computable protocol, but is inherent to voting mechanisms on any blockchain, and is therefore outside the scope of this assessment.

Malicious Backend: We considered the Backend user as a trustee entity that will always deliver the correct data. Appendix E describes potential protocol improvements to reduce the trust required in the Backend.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Properly document the candidate-proposal APIs and the possibility of denial-of-service attacks from front-running vulnerabilities**. This will make sure users are aware of this kind of attack.

❑ **Add sanity/threshold/limit checks on all system parameters**. This will guarantee that the contract will work correctly, regardless of the market's decision about its parameters.

❑ **Rename `increaseApproval` and `decreaseApproval` functions to `increaseAllowance` and `decreaseAllowance`.** Properly implement the boolean value return in these functions, in both `EtherToken` and `MarketToken` contracts to allow external contracts to use them.

❑ **Add 0x0 address checks for all the address parameters in `setPrivilege`.** This will prevent the contract from getting into an incorrect and irrecoverable state.

❑ **Add the previously staked tokens in `transferStake`.** This will prevent losing stakes when this function is called twice.

❑ **Add the previously staked tokens in `addCandidate`.** This will prevent losing stakes when this function is called twice.

❑ **Modify `removeListing` to transfer balance tokens for that listing to the listing owner, instead of burning them.** This will prevent losing stakes when this function is called.

❑ **Save the Parameterizer fee percentage and byte cost for each delivery when it is requested.** This will prevent unexpected price changes and trapped tokens in the Datatrust.

❑ **Remove the initial balance parameters in MarketToken and EtherToken.** This will avoid the possibility of the owner stealing all the tokens if its keys are compromised.

❑ **Do not store the exceeding value of bytes purchased and delivered.** `listingAccessed` can either revert in case of excess, or ignore it.

❑ **Implement a minimum amount for delivery or include `msg.sender` in the hash computation of `requestDelivery`**. Properly document this behavior, and make sure users are aware of this attack.

❑ **Properly document the candidate creation of challenges, listings or Backend, parameter changes, and stake retrievals.** This will help make sure users are aware of potential attacks to block this procedure.

❑ **Do not update `backend_url` in `register(..)`**. Update it in `resolveRegistration(..)` only if voting passes.

❑ **Implement a weighted stake, with weight decreasing over time, to incentivize users to vote earlier**. While this will not prevent users with unlimited funds from manipulating the vote in the last minute, it will make the attack more expensive.

❑ **Monitor the blockchain events for the Transfer to the Reserve account.** This will help catch the price manipulation.

❑ **Modify the `requestDelivery` and the `delivered` functions to**:

1. Calculate the total fees
2. Calculate the subtraction of the total fees and the amount paid, and
3. Add the subtraction from (2) into the total fees.

❑ **Document the edge cases of the price computation, including sending EtherTokens directly to the Reserve contract, buying/selling ShareTokens before large transactions, and claiming listing fees.** Several price manipulation attacks can be partially mitigated if the users are aware of them.

❑ **Ensure that users are aware they should approve a suitable amount of tokens to the DataTrust**. If the system parameters change, they can suddenly increase price of a delivery. A careful use of the ERC20 token approvals can help users avoid paying more than expected.

❑ **Add a voting timeout to specify how much time is allowed for a candidate to be resolved, before the voting is discarded**. While this voting resolution window will not completely solve TOB-Computable-020, complementing it with a good off-chain UI to show all the non-resolved votes to users will help further mitigate it.

❑ **Require the user to input an upper bound of the price that he is willing to pay for delivery**. Revert if price is higher than expected.

Long Term

❑ **When a user is going to perform a large buy, consider:**
1. **Warning off-chain about this behavior, and**
2. **Recommend an increase to the gas price or split the transaction into several smaller ones.** This will help avoid detection by potential attackers.

❑ **Recommend changing the hash generation to be non-repeatable.** This will avoid potential denial-of-service attacks in the voting procedure.

❑ **Take into account that every information sent to Ethereum is public prior being accepted by the network**. Design the contracts to be robust to the unpredictable nature of transaction ordering.

❑ **Review all capabilities and responsibilities of trusted users in your contracts**. The codebase relies on several privileges users, so their access boundary must be ensured to prevent privilege escalation.

❑ **Use Echidna and Manticore to ensure the correctness of the codebase.** Numerous issues can be expressed as a property to be checked by Echidna or Manticore. Appendix F shows how to use Manticore. The properties to be checked include:

- The fee computation is correct, and no token can be trapped in `Datatrust`
- No candidate can be resolved after a specified amount of time
- Votes have no effect if they are not won
- The Market token price can only be updated through the expected procedure
- The delivery cost cannot be modified after the delivery request
- No stake can be destroyed, other than through the owner withdraw
- The stake is preserved if a challenge is re-created
- The stake is preserved if `transferStake` is re-created
- All the parameters of initialization functions are validated

❑ **Make sure to comply with ERC20 implementation interfaces, even for mitigations**. You can use Open Zeppelin's ERC20 as an example.

❑ **There are two possibilities for mitigating denial-of-service issues from front-running candidate proposals.** Consider either
1. Enforcing a stake when submitting candidate applications and registrations, or
2. Implementing a pre-commit function that lets a user submit the hash of a candidate.

These two changes will increase the cost and difficulty of exploiting denial-of-service vulnerabilities using front-running attack.

Findings Summary

#	Title	Type	Severity
1	Candidate proposal denial-of-service by front-running transactions	Denial of Service	Medium
2	Re-parameterization may be abused to exploit markets	Data Validation	High
3	increaseApproval and decreaseApproval do not follow ERC20 standard	Undefined Behavior	Low
4	Missing check for zero address in setPrivileged	Input Validation	Low
5	Staked tokens can be destroyed through a failed challenge	Data Validation	High
6	Staked tokens can be destroyed through a challenge re-creation	Data Validation	High
7	A successful challenge may force the Maker to lose all tokens	Data Validation	Medium
8	Bookkeeping inconsistency in Datatrust in case of price change	Data Validation	Low
9	EtherToken/MarketToken owners can drain ether from users	Access Control	High
10	Reporting excess bytes delivered will prevent ongoing purchases	Data Validation	Medium
11	Delivering more bytes than purchased can trigger unexpected behavior for third parties	Data Validation	Low
12	Request delivery denial of service by front-running transactions	Denial of Service	Low

13	Attackers can prevent new challenges/listings/backends, parameter changes, and stake retrievals	Denial of Service	Medium
14	Malicious backend candidate can exploit submitted `url` for phishing or denial-of-service.	Data Validation	High
15	Quick buy and sell allows vote manipulation	Timing	High
16	EtherTokens can be used to increase the price arbitrarily	Data Validation	Low
17	Arithmetic rounding might lead to trapped tokens	Data Validation	Low
18	Race condition on Reserve buy and sell allows one to steal ethers	Data Validation	Medium
19	requestDelivery is prone to a race condition when computing the price	Timing	Low
20	Lack of timeout to resolve candidates	Timing	High
21	No quorum in voting allows attack to spam the election with candidates	Timing	High
22	Lack of timeout to claim listing fees allows price manipulation	Timing	High

1. Candidate proposal denial of service by front-running transactions

Severity: Medium
Type: Denial of Service
Target: `Listing`, `Datatrust`

Difficulty: Medium
Finding ID: TOB-Computable-001

Description
The voting process has two functions that can be front-run to produce a denial-of-service attack during the voting process.

The voting process requires Makers to propose a listing candidate. This could be done using the `list` function in the `Listing` contract, as shown in Figure 1.

```
@public
def list(hash: bytes32):
 """
 @notice Allows a maker to propose a new listing to a Market, creating a candidate for
voting
 @dev Listing cannot already exist, in any active form. Owner not set here as it's an
application
 @param An Ethereum hash (keccack256) serving as a unique identifier for this Listing when
hashed
 """
 assert not self.voting.isCandidate(hash) # not an applicant
 assert self.listings[hash].owner == ZERO_ADDRESS # not already listed
 self.voting.addCandidate(hash, APPLICATION, msg.sender, self.parameterizer.getStake(),
self.parameterizer.getVoteBy())
 log.Applied(hash, msg.sender)
```

Figure 1: list function in Listing.vy

Additionally, Backend candidates can be proposed using the `register` function in the `Datatrust` contract, as shown in Figure 2.

```
@public
def register(url: string[128]):
 """
 @notice Allow a backend to register as a candidate
 @param url The location of this backend
 """
 assert msg.sender != self.backend_address # don't register 2x
 self.backend_url = url # we'll clear this if the registration fails
```

```
 hash: bytes32 = keccak256(url)
 assert not self.voting.isCandidate(hash)
 self.voting.addCandidate(hash, REGISTRATION, msg.sender, self.parameterizer.getStake(),
self.parameterizer.getVoteBy())
 log.Registered(hash, msg.sender)
```

Figure 2: register function in Datatrust.vy

However, both functions could be front-run to steal the ownership of the candidate.

Exploit Scenario
Alice tries to propose her candidate. Bob sees the unconfirmed transaction and blocks Alice's transaction by front-running it. Alice cannot recover her candidate ownership unless Bob allows it.

Recommendation
Short term, add a fee to the candidate proposal.

Long term, carefully consider the unpredictable nature of Ethereum transactions and design your contracts to not depend on the transactions ordering.

2. Re-parameterization may be abused to exploit markets

Severity: High
Type: Data Validation
Target: Parametrizer

Difficulty: High
Finding ID: TOB-Computable-002

Description
Certain parameters of the contracts can be configured to invalid values, causing a variety of issues and breaking expected interactions between contracts.

Parameterizer allows market participants to re-parameterize critical market parameters via voting. However, re-parameterization lacks sanity/threshold/limit checks on all parameters. Once a parameter change is voted, the resolveReparam function will set up the new values, as shown in Figure 1.

```
@public
def resolveReparam(hash: bytes32):
 """
 @notice Determine if a Reparam Candidate collected enough votes to pass, setting it if so
 @dev This method enforces that the candidate is of the correct type and its poll is closed
 @param hash The Reparam identifier
 """
 assert self.voting.candidateIs(hash, REPARAM)
 assert self.voting.pollClosed(hash)
 # ascertain which param,value we are looking at
 param: uint256 = self.reparams[hash].param
 value: uint256 = self.reparams[hash].value
 if self.voting.didPass(hash, self.plurality):
   #TODO in time we can likely tell an optimal order for these...
   if param == PRICE_FLOOR:
     self.price_floor = value
   elif param == SPREAD:
     self.spread = value
   elif param == LIST_REWARD:
     self.list_reward = value
   elif param == STAKE:
     self.stake = value
   elif param == VOTE_BY:
     self.vote_by = value
   elif param == PLURALITY:
     self.plurality = value
```

```
    elif param == MAKER_PAYMENT:
      self.maker_payment = value
    elif param == BACKEND_PAYMENT:
      self.backend_payment = value
    elif param == COST_PER_BYTE:
      self.cost_per_byte = value
    log.ReparamSucceeded(hash, param, value)
  else: # did not get enough votes...
    log.ReparamFailed(hash, param, value)
  # regardless, cleanup the reparam and candidate
  self.voting.removeCandidate(hash)
  # TODO make sure this works as expected
  clear(self.reparams[hash])
```

Figure 1: resolveReparam (Parameterizer.vy#L194-L232)

Critical market parameters such as SPREAD, VOTE_BY, STAKE and PLURALITY are re-parameterized without any sanity/threshold/limit checks. This feature could be abused by market participants who have the required voting power.

In a "Madman"-like attack, where a well-resourced adversary can establish a challenge-proof majority, or in a collusion attack scenario of independent market participants, these parameters may be elected with values that temporarily undermine the market economics, allowing adversaries to gain unfairly and then exit.

Exploit Scenario
This issue has several exploit scenarios:

- **Scenario 1**. Alice proposes some parameters to vote. However, she includes by mistake `plurality` and `vote_by` values that are invalid. The proposal wins the voting. As a result, any contract that requires the voting procedure will always fail, breaking several important interactions between contracts.
- **Scenario 2**. Bob proposes some `price_floor` and `spread` parameters to manipulate the buy-curve and enter support at a very high CMT/CET ratio. Similarly `list_reward` may be manipulated to be very high for the Maker's future listings.
- **Scenario 3**. Alice wants to invest some ether in the Computable contract, either buying or selling some data. She makes estimations based on the current parameters to determine if it is economically viable to invest. At the same time, Bob proposes some parameters to vote. Alice decides to interact with the contracts at the same time that the parameters are changed. As a result, Alice's decision could lead to an economic loss for her.

Recommendation

Short term add parameters' invariant, including:
- PLURALITY < 100
- MAKER_PAYMENT + BACKEND_PAYEMENT < 100

Review if some of the parameters should not be equal to 0, including:
- COST_PER_BYTE
- STAKE
- VOTE_BY

Consider adding hardcoded increase and decrease limits for some of the parameters, including:
- COST_PER_BYTE
- VOTE_BY
- PRICE_FLOOR
- SPREAD
- LIST_REWARD
- STAKE

Additionally, carefully review each parameter to ensure it will not take a value, trapping the system.

Long term, use Echidna and Manticore to locate missing parameter checks.

References
- Madman attacks

3. increaseApproval and decreaseApproval do not follow ERC20 standard

Severity: Low
Type: Undefined Behavior
Target: `EtherToken`, `MarketToken`

Difficulty: Medium
Finding ID: TOB-Computable-003

Description
The `increaseApproval` and `decreaseApproval` functions in `EtherToken` and `MarketToken` contracts use non-standard names and do not return a boolean value as is expected in ERC20.

These functions were introduced to mitigate the well-known race condition in the ERC20 specification. They provide a safer way to increase or decrease the allowance. According to the mitigation proposal, they should return a boolean value to signal to the caller whether the change in allowance was successful.

The `increaseApproval` and `decreaseApproval` functions are implemented in `EtherToken` and `MarketToken` contracts, as shown in Figure 1.

```
def decreaseApproval(spender: address, amount: wei_value):
 """
 @notice Decrement the amount allowed to a spender by the given amount
 @dev If the given amount is > the actual allowance we set it to 0
 @param spender The spender of the funds
 @param amount The amount to decrease a previous allowance by
 """
 self.allowances[msg.sender][spender] -= amount #vyper will throw if overrun here
 log.Approval(msg.sender, spender, self.allowances[msg.sender][spender])

def increaseApproval(spender: address, amount: wei_value):
 """
 @notice Increase the amount a spender has allotted to them, by the owner, by the given
amount
 @param spender The address whose allowance to increase
 @param amount The amount to increase by
 """
 self.allowances[msg.sender][spender] += amount
 log.Approval(msg.sender, spender, self.allowances[msg.sender][spender])
```

Figure 1: increaseApproval and decreaseApproval functions in EtherToken.vy and MarketToken.vy

However, the functions have non-standard names: they should be called `increaseAllowance` and `decreaseAllowance`. They also do not return a boolean value, as indicated by the mitigation. This can cause difficulty or preclude the use of these tokens by external contracts, when they need to implement a safer approach to change the allowance.

Exploit Scenario
An external contract calls `increaseApproval` or `decreaseApproval`, expecting it to return a boolean value for success/failure. Given the absence of a return value, the external contract fails to infer the correct outcome of the method call, leading to unexpected behavior.

Recommendation
Short term, to comply with the mitigation, rename the functions to `increaseAllowance` or `decreaseAllowance`, and properly implement the boolean value return in these functions, in both `EtherToken` and `MarketToken` contracts.

Long term, make sure to comply with ERC20 implementation interfaces, even for mitigations. You can use Open Zeppelin's ERC20 as an example.

References
- Open Zeppelin's ERC20

4. Missing check for zero address in setPrivileged

Severity: Low
Type: Input Validation
Target: `MarketToken, Voting`

Difficulty: Medium
Finding ID: TOB-Computable-004

Description
The `setPrivileged` function allows the data market owner to specify certain addresses as privileged contracts, which will then have exclusive access to certain functions. This is meant to be called once during initialization.

In general, it is useful to add zero address checks for all such addresses, because uninitialized values in the EVM are zero values, and this check helps catch potential issues.

Although the `setPrivileged` function checks for 0x0 address on the state variables (to ascertain that this is the first time they're being initialized), it misses checking the parameters for 0x0 address, as shown in Figures 1 and 2.

```
@public
def setPrivileged(reserve: address, listing: address):
 """
 @notice We restrict some activities to only privileged contracts. Can only be called once.
 @dev We only allow the owner to set the privileged address(es)
 @param reserve The deployed address of the reserve Contract
 @param listing The deployed address of the Listing Contract
 """
 assert msg.sender == self.owner_address
 assert self.listing_address == ZERO_ADDRESS
 assert self.reserve_address == ZERO_ADDRESS
 self.reserve_address = reserve
 self.listing_address = listing
```

Figure 1: setPrivileged function in MarketToken.vy

```
@public
def setPrivileged(parameterizer: address, reserve: address, datatrust: address, listing:
address):
 """
 @notice Allow the Market owner to set privileged contract addresses. Can only be called
once.
 """
 assert msg.sender == self.owner_address
```

```
assert self.parameterizer_address == ZERO_ADDRESS
assert self.reserve_address == ZERO_ADDRESS
assert self.datatrust_address == ZERO_ADDRESS
assert self.listing_address == ZERO_ADDRESS
self.parameterizer_address = parameterizer
self.reserve_address = reserve
self.datatrust_address = datatrust
self.listing_address = listing
```

Figure 2: setPrivileged function in Voting.vy

Exploit Scenario

The data market owner accidentally uses 0x0 for one of the two parameters in `setPrivileged` of `MarketToken` for some of the four parameters in `setPrivileged` of `Voting`. The contract will then be in an incorrect and irrecoverable state because `setPrivilege` cannot be called again successfully; one of the `asserts` will always fail on the correctly set non-0x0 address. The contract will have to be redeployed.

Recommendation

Short term, add 0x0 address checks for all the address parameters in `setPrivilege`. This will prevent the contract from getting into an incorrect and irrecoverable state.

Long term, use Echidna and Manticore to ensure that all the parameters are properly validated.

5. Staked tokens can be destroyed through a failed challenge

Severity: High
Type: Data Validation
Target: Voting

Difficulty: Medium
Finding ID: TOB-Computable-005

Description
Users stake tokens to vote or create a challenge. Tokens that are not unstaked can be destroyed if the tokens' owner is challenged and wins.

`Voting.transferStake` is called in case of a failed challenge:

```
  else: # Case: listing won

    # the voting contract will make the stake avail (via unstake) to the list owner

    self.voting.transferStake(hash, self.listings[hash].owner
```

Figure 1: transferStake call (Listing.vy#L215-L217):

`transferStake` will transfer the stake of the challenger to the challenged:

```
def transferStake(hash: bytes32, addr: address):

  """

 @notice The stakes belonging to one address are being credited to another due to a
lost challenge.

 @param hash The Candidate identifier

 @param addr The Address recieving the credit

 """

  assert msg.sender == self.listing_address # only the listing contract will call this

  staked: wei_value = self.stakes[self.candidates[hash].owner][hash]

  clear(self.stakes[self.candidates[hash].owner][hash])

  self.stakes[addr][hash] = staked
```

Figure 2: *transferStake function* (Voting.vy#L209-L218)

`transferStake` assigns the destination's stake, without adding the previously staked tokens. As a result, if the challenger has staked tokens previously, these tokens will be destroyed.

This situation can occur either by mistake or by a malicious challenger.

Exploit Scenario
Bob is a listing owner. Eve challenges Bob's listing. The challenge cost is 100$. Bob votes against the challenge, and stakes 10,000$. Bob wins the vote. Bob's stake is reduced to 100$. Bob loses 9,900$.

Recommendation
Short term, add the previously staked tokens in `transferStake`.

Long term, use Echidna and Manticore to ensure that the stake is always preserved if the `transferStake` is called.

6. Staked tokens can be destroyed through a challenge re-creation

Severity: High
Type: Data Validation
Target: Voting

Difficulty: Medium
Finding ID: TOB-Computable-006

Description
Users stake tokens to vote or create a challenge. Tokens that are not unstaked can be destroyed if the tokens' owner challenges a listing multiple times.

To challenge a listing, a stake must be sent:

```
if kind == CHALLENGE: # a challenger must successfully stake a challenge
  self.market_token.transferFrom(owner, self, stake)
  self.stakes[owner][hash] = stake
```

Figure 1: addCandidate (Voting.vy#L134-L136)

The new value is assigned to the stake and is not added to the previously staked tokens. As a result, if previously staked tokens are not withdrawn, they will be destroyed.

Exploit Scenario
Bob votes against Alice's listing. During the vote, Bob stakes 10,000$. Alice's listing is accepted. A few days later, Bob convinces other users that Alice's listing should be removed. Bob challenges Alice's listing. As a result, Bob's 10,000$ stake is destroyed.

Recommendation
Add the previously staked tokens in `addCandidate`.

Long term, use Echidna and Manticore to ensure that the stake is always preserved if a challenge is re-created.

7. A successful challenge may force the Maker to lose all tokens

Severity: Medium
Type: Data Validation
Target: Listing

Difficulty: Medium
Finding ID: TOB-Computable-007

Description

When a successful challenge is resolved, it forcibly removes the listing and burns the associated tokens, without giving the listing owner (i.e., Maker) a chance to withdraw reward tokens already accumulated.

The code in removeListing removes a listing and burns all the tokens associated with it, as shown in Figure 1.

```
def removeListing(hash: bytes32):
 """
 @notice Used internally to remove listings
 @dev We clear the members of a struct pointed to by the listing hash (enabling re-use)
 @param hash The listing to remove
 """
 supply: wei_value = self.listings[hash].supply
 if supply > 0:
   self.market_token.burn(supply)
 clear(self.listings[hash]) # TODO assure we don't need to do this by hand
 # datatrust now needs to clear the data hash
 self.datatrust.removeDataHash(hash)
 log.ListingRemoved(hash)
```

Figure 1: removeListing function in Voting.vy

This private function is called from `resolveChallenge` when a challenge wins, as shown in Figure 2.

```
@public
def resolveChallenge(hash: bytes32):
 """
 @notice Determines the winner for a given Challenge. Rewards winner, possibly removing a
listing (if appropriate)
 @dev Challenge stakes are unlocked, replinished and rewarded accordingly. Note that the
ability to fund the stake
 takes precedence over voting
```

```
  @param hash The identifier for a Given challenge
  """
  assert self.voting.candidateIs(hash, CHALLENGE)
  assert self.voting.pollClosed(hash)
  owner: address = self.voting.getCandidateOwner(hash) # TODO this could likely be removed
now
  # Case: challenge won
  if self.voting.didPass(hash, self.parameterizer.getPlurality()):
    self.removeListing(hash)
    log.ChallengeSucceeded(hash, owner)
  else: # Case: listing won
    # the voting contract will make the stake avail (via unstake) to the list owner
    self.voting.transferStake(hash, self.listings[hash].owner)
    log.ChallengeFailed(hash, owner)
  # regardless, clean up the candidate
  self.voting.removeCandidate(hash)
```

Figure 2: resolveChallenge (Listing.vy#L212-L213)

However, in certain situations, `removeListing` can burn tokens that a user fails to withdraw.

Exploit Scenario

Alice has a successful listing. It has been accessed many times and allowed Alice to accumulate a wealth of market tokens. Bob challenges Alice's listing. The market votes in Bob's favor. Alice is disconnected and unable to withdraw her tokens in time. The successful challenge removes her listing after burning all her accumulated tokens for that listing.

Recommendation

Short term, modify `removeListing` to transfer balance tokens for that listing to the listing owner instead of burning them.

Long term, use Echidna and Manticore to prove that no stake can be destroyed if the owner does not withdraw it.

8. Bookkeeping inconsistency in Datatrust in case of price change

Severity: Low
Type: Data Validation
Target: Datatrust

Difficulty: Medium
Finding ID: TOB-Computable-008

Description
When requesting a delivery, the user pays its cost. The actual delivery cost and the cost paid can differ if the fees parameter are changed during the delivery. As a result, the datatrust contract can have an unexpected deficit or surplus.

The cost of a delivery comprises the fees for the Maker, the Backend, and the Reserve. These costs rely on parameters returned by the Parameterizer contract.

The delivery cost is paid during the delivery request:

```
  total: wei_value = self.parameterizer.getCostPerByte() * amount

  res_fee: wei_value = (total * self.parameterizer.getReservePayment()) / 100

  self.ether_token.transferFrom(msg.sender, self, total) # take the total payment
```

Figure 1: requestDelivery (Datatrust.vy#L209-L211)

The fees for the Maker and the Backend are paid after the delivery:

```
  # now pay the datatrust from the banked delivery request

  back_fee: wei_value = (self.parameterizer.getCostPerByte() * requested *
self.parameterizer.getBackendPayment()) / 100

  self.ether_token.transfer(self.backend_address, back_fee)
```

Figure 2: delivered (Datatrust.vy#L293-L295)

```
 # the algo for maker payment is (accessed*cost)/(100/maker_pct)
 accessed: uint256 = self.datatrust.getBytesAccessed(hash)
 maker_fee: wei_value = (self.parameterizer.getCostPerByte() * accessed *
self.parameterizer.getMakerPayment()) / 100
```

Figure 3: claimBytesAccessed (Listing.vy#L172-L174)

The fees rely on the values returned by the Parameterizer contract, which can change after the delivery request. As a result, the cost initially paid by the user will not be equal to the sum of the fees.

If the inconsistency leads the datatrust to a deficit of ethers, it will not be possible for all users to withdraw their currency. If the inconsistency leads to an extra number of ethers, these ethers will be trapped.

Exploit Scenario
Bob is the only delivery buyer of the system. Bob requests a delivery of 100 bytes for 1$ per byte and pays 100$. The Backend fee percentage is 20%. The delivery is made, leaving 20$ in the datatrust contract. The cost increases to 2$ per byte before the end of the delivery. As a result, the datatrust contract does not hold enough money, and the Backend is unable to receive its fee.

Recommendation
Short term, save the Parameterizer fee percentage and byte cost for each delivery when it is requested.

Long term, use Echidna and Manticore to show that the cost cannot be modified after a delivery is requested.

9. EtherToken/MarketToken owners can drain ether from users

Severity: High
Type: Access Control
Target: EtherToken, MarketToken

Difficulty: High
Finding ID: TOB-Computable-009

Description
The owner of the EtherToken and MarketToken contracts have the power to initialize their accounts with any number of tokens and then withdraw them.

The `EtherToken` contract is initialized with an address and some arbitrary number of tokens as the initial balance, as shown in Figure 1.

```
def __init__(initial_account: address, initial_balance: wei_value):
 self.balances[initial_account] = initial_balance
 self.decimals = 18
 self.symbol = "CET"
 self.supply = initial_balance
```

Figure 1. constructor of the EtherToken contract

Similar code is available in the MarketToken contract.

Any user can also recover their ether, using the withdraw function, as shown in Figure 2.

```
def withdraw(amount: wei_value):
 """
 @notice Allow msg.sender to withdraw an amount of ETH
 @dev The Vyper builtin `send` is used here
 @param amount An amount of ETH in wei to be withdrawn
 """
 self.balances[msg.sender] -= amount
 self.supply -= amount
 send(msg.sender, amount)
 log.Withdrawn(msg.sender, amount)
```

Figure 2. withdraw function in EtherToken.vy

The withdraw function is also present in the `Reserve` contract and allows users to exchange market tokens for ether tokens, as shown in Figure 3.

```
def withdraw():
```

```
    """
    @notice Allows a supporter to exit the market. Burning any market token owned and
    withdrawing their share of the reserve.
    @dev Supporter, if owning a challenge, may want to wait until that is over (in case they
win)
    """
    withdrawn: wei_value = self.getWithdrawalProceeds(msg.sender)
    assert withdrawn > 0
    # before any transfer, burn their market tokens...
    self.market_token.burnAll(msg.sender)
    self.ether_token.transfer(msg.sender, withdrawn)
    log.Withdrawn(msg.sender, withdrawn)
```

Figure 3. withdraw function in Reserve.vy

This gives token contract owners the power to withdraw any amount of tokens as ether.

Exploit Scenario
The private key of the owner of one the token contracts is compromised. The attacker is able to steal all the ether from legitimate users. As a result, the price of ether tokens suffers a big drop, as users lose confidence in the contracts.

Recommendation
In the short term, remove the initial balance parameters in MarketToken and EtherToken.

In the long term, review all capabilities and responsibilities of trusted users in your contracts.

10. Reporting excess bytes delivered will prevent ongoing purchases

Severity: Medium
Type: Data Validation
Target: Datatrust.vy

Difficulty: Medium
Finding ID: TOB-Computable-010

Description
When requesting a delivery, the user pays its cost. The actual delivery cost and the amount paid can differ if the Backend reports more bytes delivered than what was paid for. The excess cost will be taken from the pool of the user's ongoing purchases, preventing their completion.

The Backend can report the delivery of excess bytes:

```
 # this can be claimed later by the listing owner, and are subtractive to bytes_purchased
 self.bytes_accessed[listing] += amount
 self.bytes_purchased[self.deliveries[delivery].owner] -= amount
# bytes_delivered must eq (or exceed) bytes_requested in order for a datatrust to claim
delivery
 self.deliveries[delivery].bytes_delivered += amount
```

Figure 1 (Datatrust.vy#L252-L253)

`bytes_purchased` decreases by the amount of bytes reported. It holds the total amount of bytes available for the user's ongoing purchases. As a result, if `bytes_purchased` decreases more than the bytes paid for this purchase, it will not hold enough to cover the other purchases. The other purchases can be trapped and never completed.

Exploit Scenario
Scenario 1. The Datatrust holds $10,000. Bob requests two deliveries and pays $100 for each. The Backend reports more bytes than are paid for the first delivery. The cost of the delivery with the excess is $150. The first delivery is completed. The second delivery requires $100, but only $50 is left. As a result, the second delivery cannot be completed.

Scenario 2. Bob requests two deliveries, one of $100 on Eve's listing, and one of $1,000 on Alice's listing. The Backend colludes with Eve and reports 10 times the bytes expected on Eve's listing. Eve shares the fees with the Backend. As a result, the Backend and Eve steal ethers meant for Alice.

Recommendation
Do not store the exceeding value. `listingAccessed` can either revert in case of excess, or be ignored.

Ensure that all the features are properly tested and documented. Consider using Manticore to explore the contract systematically.

11. Delivering more bytes than purchased can trigger unexpected behavior for third parties

Severity: Low
Type: Data Validation
Target: Datatrust.vy

Difficulty: Medium
Finding ID: TOB-Computable-011

Description
The Backend is allowed to deliver and report more bytes than purchased. If the user has no other ongoing purchase, the report will fail, leading to unexpected behavior for the Backend.

The Backend can report the delivery of an excess number of bytes:

```
 # this can be claimed later by the listing owner, and are subtractive to bytes_purchased
 self.bytes_accessed[listing] += amount
 self.bytes_purchased[self.deliveries[delivery].owner] -= amount
# bytes_delivered must eq (or exceed) bytes_requested in order for a datatrust to claim
delivery
 self.deliveries[delivery].bytes_delivered += amount
```

Figure 1 (Datatrust.vy#L252-L253)

If the user has no other ongoing purchases,

```
self.bytes_purchased[self.deliveries[delivery].owner] -= amount
```

will fail. As a result, it will not be possible for the Backend to report more bytes than purchased.

Exploit Scenario
Bob requests a delivery for 1,000 bytes. The Backend delivers 1,100 bytes. The Backend calls `listingAccessed` to report the delivery. The call fails, preventing the Backend from reporting the delivery on-chain.

Recommendation
Do not store the exceeding value. `listingAccessed` can either revert in case of excess, or be ignored.

Ensure that all the features are properly tested and documented. Consider using Manticore to explore the contract systematically.

12. Request delivery denial of service by front-running transactions

Severity: Low
Type: Denial of Service
Target: `Datatrust`

Difficulty: High
Finding ID: TOB-Computable-012

Description
The delivery process has a function that can be front-run to produce a denial-of-service attack during the delivery process.

Users can purchase data by calling `requestDelivery`, with a unique `hash` identifier:

```
def requestDelivery(hash: bytes32, amount: uint256):
 """
 @notice Allow a user to purchase an amount of data to be delivered to them
 @param hash This is a keccack hash recieved by a client that uniquely identifies a
request. NOTE care should be taken
 by the client to insure this is unique.
 @param hash A unique hash generated by the client used to identify the delivery
 @param amount The number of bytes the user is paying for.
 """
 assert self.deliveries[hash].owner == ZERO_ADDRESS # not already a request
 total: wei_value = self.parameterizer.getCostPerByte() * amount
 res_fee: wei_value = (total * self.parameterizer.getReservePayment()) / 100
 self.ether_token.transferFrom(msg.sender, self, total) # take the total payment
 self.ether_token.transfer(self.reserve_address, res_fee) # transfer res_pct to reserve
 self.bytes_purchased[msg.sender] += amount # all purchases by this user. deducted from via
listing access
 self.deliveries[hash].owner = msg.sender
 self.deliveries[hash].bytes_requested = amount
```

Figure 1: requestDelivery function in Datatrust.vy

Ethereum transactions are sent to the network before being accepted. Thus, the hash is public prior to being registered. As a result, an attacker can steal delivery ownership by front-running the call to `requestDelivery`.

Exploit Scenario
Alice requests a delivery. Eve sees the transaction before it has been accepted and calls `requestDelivery` with the same hash. Eve's transaction is accepted before Alice's. As a result, Alice cannot receive her delivery.

Recommendation

Short term, consider one of the following solutions:

- Implement a minimum amount for a delivery.
- Including msg.sender in the hash computation. For example, `requestDelivery(hash: bytes32, amount: uint256)` could be `requestDelivery(seed: bytes32, amount: uint256)`, where `hash` is later computed, using the `seed` and `msg.sender`. If msg.sender is not meant to be known by the actor generating the seed, use `amount`.
- Properly document this behavior, and make sure users are aware of this attack.

Long term, take into account that every piece of information sent to Ethereum is public prior being accepted by the network. Design the contracts to be resilient against the unpredictable nature of transaction ordering.

13. Attackers can prevent new challenges/listings/backends, parameter changes, and stake retrievals

Severity: Medium
Type: Denial of Service
Target: `Voting`

Difficulty: High
Finding ID: TOB-Computable-013

Description
Candidate creation allows attackers to block essential operations performed by other users.

The computable contracts allow candidates to propose new challenges, listings, or Backend and parameter changes. The `addCandidate` function implements this functionality:

```
def addCandidate(hash: bytes32, kind: uint256, owner: address, stake: wei_value, vote_by:
timedelta):
 """
 @notice Given a listing or parameter hash, create a new voting candidate
 @dev Only priveliged contracts may call this method
 @param hash The identifier for the listing or reparameterization candidate
 @param kind The type of candidate we are adding
 @param owner The adress which owns this created candidate
 @param stake How much, in wei, must be staked to vote or challenge
 @param vote_by How long into the future until polls for this candidate close
 """
 assert self.hasPrivilege(msg.sender)
 assert self.candidates[hash].owner == ZERO_ADDRESS
 if kind == CHALLENGE: # a challenger must successfully stake a challenge
   self.market_token.transferFrom(owner, self, stake)
   self.stakes[owner][hash] = stake
 end: timestamp = block.timestamp + vote_by
 self.candidates[hash].kind = kind
 self.candidates[hash].owner = owner
 self.candidates[hash].stake = stake
 self.candidates[hash].vote_by = end
 log.CandidateAdded(hash, kind, owner, end)
```

Figure 1: addCandidate function in Voting.vy

However, this approach can be used to prevent proposing new challenges, listings, or Backend, as well as parameter changes and stake retrievals. Anyone can add arbitrary hashes and block other users from candidate creation.

According to the documentation, the developers might be aware of this issue.

Exploit Scenario

Eve can disrupt Alice’s efforts to create fake candidates:

- **Scenario 1:** Alice wants to challenge a listing. To block Alice's action, Eve can create a fake challenge for that listing. As a result, Alice's transaction to create a challenge will revert.
- **Scenario 2:** Alice wants to propose a parameter change. To block Alice's action, Eve can create a fake parameter change. As a result, Alice's transaction to create a challenge will revert.
- **Scenario 3:** Alice wants to propose a new listing or Backend. To block Alice's action, Eve can create a fake candidate, using the same name. As a result, Alice's transaction to create a new candidate will revert.
- **Scenario 4**: If Alice finished voting and has $10,000 at stake, Eve can repeatedly create a new candidate with the same hash. As a result, Alice will be unable to withdraw her funds.

Recommendation
Short term, properly document this behavior, and make sure users are aware of this attack.

Long term, change the hash generation to be non-repeatable (for example, each vote would have a unique hash, using `msg.sender`, which cannot be reused).

14. Malicious Backend candidate can exploit submitted `url` for phishing or denial of service.

Severity: High
Type: Data Validation
Target: Datatrust

Difficulty: Low
Finding ID: TOB-Computable-014

Description
An incorrect update schema allows attackers to temporarily steal the Backend url destination, leading to denial of service for phishing campaigns.

`getBackendUrl` returns the URL of Backend's website, as shown in Figure 1.

```
@public
@constant
def getBackendUrl() -> string[128]:
 """
 @notice Return the URL of the currently registered backend
 """
 return self.backend_url
```

Figure 1. `getBackendUrl()` *in Datatrust.vy*

During a new Backend registration, the `backend_url` is updated, even when the registration's voting has not been completed:

```
@public
def register(url: string[128]):
 """
 @notice Allow a backend to register as a candidate
 @param url The location of this backend
 """
 assert msg.sender != self.backend_address # don't register 2x
 self.backend_url = url # we'll clear this if the registration fails
 hash: bytes32 = keccak256(url)
 assert not self.voting.isCandidate(hash)
 self.voting.addCandidate(hash, REGISTRATION, msg.sender, self.parameterizer.getStake(),
self.parameterizer.getVoteBy())
 log.Registered(hash, msg.sender)
```

Figure 2. `register` *function in Datatrust.vy*

As a result, anyone can update the Backend url.

Exploit Scenario
Eve registers as a Backend candidate and provides a malicious url. Eve's url is stored in Datatrust. While voting is in progress, buyers who want to access data interact with Eve's url. As a result, Eve could mislead users to steal their funds.

Recommendation
Short term, do not update `backend_url` in `register(..)`. Update it in `resolveRegistration(..)` only if voting passes.

Long term, use Manticore and Echidna to ensure that vote-based modifications cannot occur without a vote being won.

15. Quick buy and sell allows vote manipulation

Severity: High
Type: Timing
Target: `Voting`

Difficulty: High
Finding ID: TOB-Computable-015

Description
Computable relies on a voting system that allows anyone to vote with any weight at the last minute. As a result, anyone with a large fund can manipulate the vote.

Computable's voting mechanism relies on staking. There is no incentive for users to stake tokens well before the voting ends. Users can buy a large amount of Market tokens just before voting ends and sell them right after it. As a result, anyone with a large fund can decide the outcome of the vote, without being a market participant.

As all the votes are public, users voting earlier will be penalized, because their votes will be known by the other participants. An attacker can know exactly how much currency will be necessary to change the outcome of the voting, just before it ends.

Exploit Scenario
Alice and Bob vote for their candidates with $10,000 worth of Market tokens. Eve buys $20,001 worth of Market token one block before the end of the vote. Eve votes for her candidate. Her candidate is elected. Eve sells all her Market tokens one block after the vote. As a result, Eve decided the outcome of the vote without being an active user of the system.

Recommendation
Blockchain-based online voting is a known challenge. No perfect solution has been found so far.

Incentivize users to vote earlier by implementing a weighted stake, with a weight decreasing over time. While it will not prevent users with unlimited resources to manipulate the vote at the last minute, it will make the attack more expensive.

16. EtherTokens can be used to increase the price arbitrarily

Severity: Low
Type: Data Validation
Target: Reserve

Difficulty: Low
Finding ID: TOB-Computable-016

Description
The support price can be arbitrarily increased by transferring Ether tokens to the Reserve contract account.

Users can buy Market tokens using the `support` function. Several parameters determine the price, including the price floor, the spread, the amount of Ether tokens in the Reserve, and the total supply of Market tokens, as shown in Figure 1.

```
def getSupportPrice() -> wei_value:
 """
 @notice Return the amount of Ether token (in wei) needed to purchase one billionth of a
Market token
 """
 price_floor: wei_value = self.parameterizer.getPriceFloor()
 spread: uint256 = self.parameterizer.getSpread()
 reserve: wei_value = self.ether_token.balanceOf(self)
 total: wei_value = self.market_token.totalSupply()
 if total < 1000000000000000000: # that is, is total supply less than one token in wei
   return price_floor + ((spread * reserve * 1000000000) / (100 * 1000000000000000000))
 else:
   return price_floor + ((spread * reserve * 1000000000) / (100 * total)) # NOTE the
multiplier ONE_GWEI
```

Figure 1. `getSupportPrice` function in Reserve.vy

Any user can manipulate this price by transferring tokens directly, using the EtherToken contract, to the address of the Reserve contract. This will increase the `reserve` value, without incrementing the Market Token total supply.

While a `Transfer` event will log the increase in the Reserve balance, it is unlikely that token holders will notice someone increasing the Market token price through Ether tokens transfers.

Exploit Scenario
Alice wants to buy Market tokens. Bob wants to keep control of the majority of market, so he transfers Ether tokens to the Reserve contract to artificially increase the price.
As a result, Alice desists in buying Market tokens, and Bob keeps control of the market.

Recommendation
Short term, monitor the blockchain events for any Transfer to the Reserve account to catch this price manipulation.

Use Manticore and Echidna to ensure the Market token price can only be updated through expected procedures.

17. Arithmetic rounding might lead to trapped tokens

Severity: Low
Type: Data Validation
Target: Datatrust.vy

Difficulty: Medium
Finding ID: TOB-Computable-017

Description
An arithmetic rounding issue in the delivery cost can lead to trapped tokens. The number of trapped tokens is low and does not present a significant loss.

Every time a user requests a delivery, part of its cost is transferred as fees for the reserve and the Backend. The cost of a delivery is computed in the `requestDelivery` and the `delivered` functions by the following formula:

```
total = getCostPerByte() * requested
total_fees = total * getMakerPayment / 100 + total * getBackendPayment /
100 + total * getReservePayment / 10
```

`total` is the cost paid for the delivery, and `total_fees` is the actual cost of the delivery. Due to arithmetic imprecision, `total` might be slightly greater than `total_fees`. As a result, the difference between the two will be trapped in the Datatrust contract.

Exploit Scenario
The parameters of the system are:

- Cost of the delivery: 101
- Market percentage: 33%
- Backend percentage: 33%
- Backend percentage: 33%
- Reserve percentage: 34%
- Price floor: 10**9

Bob buys for 72425248 bytes of data. The cost of the delivery is 7314950048 wei. The total amount of fees paid is 7314950047 wei, trapping 1 wei in the Datatrust contract.

Recommendation
Short term, modify the `requestDelivery` and the `delivered` functions to:

1. Calculate the total fees
2. Calculate the difference between the total fees and the amount paid, and
3. Add the difference into the total fees.

Long term, use Manticore to ensure the fee computation is correct, and no token can be trapped in the Datatrust contract. Appendix F provides a script to implement this check.

18. Race condition on Reserve buy and sell allows one to steal ethers

Severity: Medium
Type: Data Validation
Target: Reserve.vy

Difficulty: Medium
Finding ID: TOB-Computable-018

Description
A race condition on the Reserve contract allows an attacker to make a profit by knowing a significant change in the Market token supply will occur.

In Ethereum, all the transactions appear on the network before being accepted. Users can see upcoming Market token buys and sells. As a result, an attacker can make a profit by buying and selling tokens quickly when she detects a profitable situation.

Our initial investigation showed that the exploitation can only be profitable if the new buy is made with more ether than the current Reserve's balance. Due to time constraint, we did not verify this claim.

Exploit Scenario
The parameters of the system are:

- Computable market tokens balance: 10**21
- Price floor: 10**6
- Spread 110

Bob calls support with 1 ether and receives 199 * 10**18 market tokens. Bob wants to buy more and calls support with 100 ether. Eve sees Bob's transaction before it has been mined and calls support with 1 ether. Eve receives 99 * 10**18 market tokens. Eve's transaction is mined before Bob's. Bob's transaction is accepted. Eve calls withdraw and receives 1.14 ethers. As a result, Eve made a profit of 0.14 ether.

Appendix F contains several exploit scenarios, built on top of Manticore.

Recommendation
Short term, document this behavior to make sure users are aware of price distortions that can be exploited by others. On large buys, consider:

1. Warning off-chain about this behavior
2. Recommending an increase to the gas price or splitting the transaction into several smaller ones to avoid detection by potential attackers

Long term, carefully consider the unpredictable nature of Ethereum transactions and design your contracts to not depend on transaction ordering.

19. requestDelivery is prone to a race condition when computing the price

Severity: Low
Type: Timing
Target: Reserve

Difficulty: High
Finding ID: TOB-Computable-019

Description
The function to pay for delivery does not fix its price, so the user cannot be sure about its price until the transaction is confirmed.

All the parameters of the market are subject to modification using the voting procedure at any time. One of these parameters allows control over the price per byte in a delivery:

```
@public
def resolveReparam(hash: bytes32):
 """
 @notice Determine if a Reparam Candidate collected enough votes to pass, setting it if so
 @dev This method enforces that the candidate is of the correct type and its poll is closed
 @param hash The Reparam identifier
 """
 assert self.voting.candidateIs(hash, REPARAM)
 assert self.voting.pollClosed(hash)
 # ascertain which param,value we are looking at
 param: uint256 = self.reparams[hash].param
 value: uint256 = self.reparams[hash].value
 if self.voting.didPass(hash, self.plurality):
   #TODO in time we can likely tell an optimal order for these...
   ...
   elif param == COST_PER_BYTE:
     self.cost_per_byte = value
   log.ReparamSucceeded(hash, param, value)
...
```

Figure 1: `resolveReparam` in Parameterizer.vy

This parameter will be used to compute the delivery price, which users should pay upfront when they call requestDelivery:

```
def requestDelivery(hash: bytes32, amount: uint256):
 """
 @notice Allow a user to purchase an amount of data to be delivered to them
 @param hash This is a keccack hash recieved by a client that uniquely identifies a
```

```
request. NOTE care should be taken
 by the client to insure this is unique.
 @param hash A unique hash generated by the client used to identify the delivery
 @param amount The number of bytes the user is paying for.
 """
 assert self.deliveries[hash].owner == ZERO_ADDRESS # not already a request
 total: wei_value = self.parameterizer.getCostPerByte() * amount
 res_fee: wei_value = (total * self.parameterizer.getReservePayment()) / 100
 self.ether_token.transferFrom(msg.sender, self, total) # take the total payment
 self.ether_token.transfer(self.reserve_address, res_fee) # transfer res_pct to reserve
 self.bytes_purchased[msg.sender] += amount # all purchases by this user. deducted from via
listing access
 self.deliveries[hash].owner = msg.sender
 self.deliveries[hash].bytes_requested = amount
```

Figure 2. `requestDelivery` function in Datatrust.vy

However, it is not possible for a user to know in advance what price to pay. If the parameters change during the data deliver request, the price to pay could be increased, forcing the user to pay for it upfront.

Exploit Scenario

Alice wants to request a delivery. She calls `requestDelivery`, expecting to pay a certain amount for it. At the same time, a proposal to change the cost per byte is closing. Bob calls `resolveReparam`, and his transaction is confirmed before Alice's. As a result, Alice pays a higher price for her delivery.

Recommendation

Short term, request the user to input an upper bound of the price that he is willing to pay for delivery. Revert if price is higher than expected. As a simpler alternative, document this behavior, advising users to carefully approve a tight amount of tokens for the DataTrust.

Long term, do not make assumptions on the contract's state when computing a price. Always limit the tokens' transfer to a limit provided by the user.

20. Lack of timeout to resolve candidates

Severity: High
Type: Timing
Target: Listing.vy, Parametrizer.vy

Difficulty: High
Finding ID: TOB-Computable-020

Description
After voting ends, there is no timeout to approve a winning candidate to change the state of the contract. A user can wait an arbitrary amount of time to do so.

After the voting period ends, one of the users is expected to call a resolve* function (e.g., `resolveRegistration`) to determine if the proposal will be accepted and update the corresponding contract (e.g., Datatrust).

```
def resolveRegistration(hash: bytes32):
 """
 @notice Set internal backend in use if approved (remove if not)
 @param hash The hashed string url of the backend candidate
 """
 assert self.voting.candidateIs(hash, REGISTRATION)
 assert self.voting.pollClosed(hash)
 owner: address = self.voting.getCandidateOwner(hash)
 # case: listing accepted
 if self.voting.didPass(hash, self.parameterizer.getPlurality()):
   self.backend_address = owner
   log.RegistrationSucceeded(hash, owner)
 else: # application did not pass vote
   log.RegistrationFailed(hash, owner)
   clear(self.backend_url)
 # regardless, the candidate is pruned
 self.voting.removeCandidate(hash)
```

Figure 1. `resolveRegistration` function in Datatrust.vy

However, there is no timeout to call `resolveRegistration.` So the actual contract updates do not happen until someone calls `resolveRegistration.` If voting was successful, the voted proposals affect the contract state. Other users interacting with these contracts will then suddenly be exposed to the updated contract state, which could significantly change the expected values.

Exploit Scenario

Eve deploys the Computable contracts. She makes a successful vote to register her own address as the Backend address of the Datatrust contract, but does not finalize the vote. Eve sets sets the Backend address to a known and trusted user. Alice and Bob verify that the deployed contracts are not modified in any way and start using them without realizing the Backend address could be changed without warning. After some time, Eve calls `resolveRegistration` to change the Backend address and steals all the fees.

Recommendation
Short term, add timeout to voting to specify how much time is allowed for a candidate to be resolved before discarding the vote. This value should remain below a hardcoded time (e.g., 24 hours) to prevent the creation of malicious proposals with large timeout. Ensure the offchain UI shows clearly the pending proposals to the users.

Long term, use Manticore and Echidna to ensure no candidate can be resolved after an arbitrary amount of time.

21. No quorum in voting allows attack to spam the election with candidates

Severity: High
Type: Timing
Target: Voting.vy

Difficulty: Low
Finding ID: TOB-Computable-021

Description
A malicious user can abuse candidates creation and the vote criteria to pass a candidate with low stake cost.

Unless you propose a new challenge, the candidate creation has no cost, nor is it restricted in any way:

```
def addCandidate(hash: bytes32, kind: uint256, owner: address, stake: wei_value, vote_by:
timedelta):
 """
 @notice Given a listing or parameter hash, create a new voting candidate
 @dev Only priveliged contracts may call this method
 @param hash The identifier for the listing or reparameterization candidate
 @param kind The type of candidate we are adding
 @param owner The adress which owns this created candidate
 @param stake How much, in wei, must be staked to vote or challenge
 @param vote_by How long into the future until polls for this candidate close
 """
 assert self.hasPrivilege(msg.sender)
 assert self.candidates[hash].owner == ZERO_ADDRESS
 if kind == CHALLENGE: # a challenger must successfully stake a challenge
   self.market_token.transferFrom(owner, self, stake)
   self.stakes[owner][hash] = stake
 end: timestamp = block.timestamp + vote_by
 self.candidates[hash].kind = kind
 self.candidates[hash].owner = owner
 self.candidates[hash].stake = stake
 self.candidates[hash].vote_by = end
 log.CandidateAdded(hash, kind, owner, end)
```

Figure 2: addCandidate function in Voting.vy

Users are free to vote for any candidate, as many times as they want, using their market tokens as stake:

```
@public
def vote(hash: bytes32, option: uint256):
 """
 @notice Cast a vote for a given candidate
 @dev User mush have approved market token to spend on their behalf
 @param hash The candidate identifier
 @param option Yea (1) or Nay (!1)
 """
 assert self.candidates[hash].owner != ZERO_ADDRESS
 assert self.candidates[hash].vote_by > block.timestamp
 stake: wei_value = self.candidates[hash].stake
 self.market_token.transferFrom(msg.sender, self, stake)
 self.stakes[msg.sender][hash] += stake
 if option == 1:
   self.candidates[hash].yea += 1
 else:
   self.candidates[hash].nay += 1
 log.Voted(hash, msg.sender)
```

Figure 3: vote function in Voting.vy

After the voting period is over, the `didPass` function determines if the candidate is elected, using the fraction of the number of positive votes, over the total ones and the plurality.

```
def didPass(hash: bytes32, plurality: uint256) -> bool:
 """
 @notice Return a bool indicating whether a given candidate recieved enough votes to exceed
the plurality
 @dev The poll must be closed. Also we cover the corner case that no one voted.
 @return bool
 """
 assert self.candidates[hash].owner != ZERO_ADDRESS
 assert self.candidates[hash].vote_by < block.timestamp
 yea: uint256 = self.candidates[hash].yea
 total: uint256 = yea + self.candidates[hash].nay
 # edge case that no one voted
 if total == 0:
   # theoretically a market could have a 0 plurality
   return plurality == 0
 else:
```

```
    return ((yea * 100) / total) >= plurality
```

Figure 4: vote function in Voting.vy

However, there is no minimum number of votes (quorum), so a single, positive vote is the only requirement for passing a candidate.

Exploit Scenario
Eve wants to pass a candidate. Eve creates 1,000 proposals for this candidate. Eve waits until the last second to vote on one of the proposals with no votes. Bob and Alice did not vote on this proposal. The proposal passed, and Eve's candidate is accepted.

Recommendation
Short term, add a fixed fee to create a candidate and a quorum on vote. Monitor the system to detect any suspicious activity on proposal creations.

Long term, use Manticore and Echidna to ensure no candidate can be passed with less than the minimum number of quorum votes.

22. Lack of timeout to claim listing fees allows price manipulation

Severity: High
Type: Timing
Target: Listing.vy

Difficulty: High
Finding ID: TOB-Computable-022

Description
After a delivery completes, there is no timeout to claim the listing fees. Claiming fees will mint new tokens, affecting the price. As a result, a successful listing owner can wait an arbitrary amount of time to claim its tokens and exploit a significant price change in the market.

After a delivery is at least partially complete, the listing owner should call the `claimBytesAccessed` function to collect the listing fees.

```
def claimBytesAccessed(hash: bytes32):
 """
 @notice Allows a listing owner to claim the rewards of listing access. These support
 the market and will be noted at the listing.supply (MarketToken)
 @param hash The listing identifier
 """
 assert msg.sender == self.listings[hash].owner
 # the algo for maker payment is (accessed*cost)/(100/maker_pct)
 accessed: uint256 = self.datatrust.getBytesAccessed(hash)
 maker_fee: wei_value = (self.parameterizer.getCostPerByte() * accessed *
self.parameterizer.getMakerPayment()) / 100
 price: wei_value = self.reserve.getSupportPrice()
 # if credits accumulated are too low for support, exit now
 assert maker_fee >= price
 # clear the credits before proceeding (also transfers fee to reserve)
 self.datatrust.bytesAccessedClaimed(hash, maker_fee)
 # support is now called, according to the buy-curve.
 minted: uint256 = (maker_fee * 1000000000) / price # 1Billionth token is the smallest
denomination...
 self.market_token.mint(minted)
 self.listings[hash].supply += minted
 log.BytesAccessedClaimed(hash, accessed, minted)
```

Figure 1. `claimBytesAccessed` function in Listing.vy

This function will mint new market tokens to pay for the fees. However, there is no timeout to call this function. Minting new market tokens will affect the price of it.

Exploit Scenario
Eve creates a legitimate listing, and other users request her data. After some time, she collects a large amount of fees. Instead of calling *`claimBytesAccessed`* for each, she waits until there is a convenient moment for her to buy or sell market tokens. As a result, Eve is able to manipulate the price and exploit the market.

Recommendation
Short term, consider implementing one of the following mitigations:
1. Allow any user to call *`claimBytesAccessed`*, while incentivizing the caller to get a small fee for it.
2. Allow the contract owner to call *`claimBytesAccessed`*, if a potential attack to manipulate prices is detected.
3. Allow the datatrust to call *`claimBytesAccessed`* after a delivery.
4. Automatically call *`claimBytesAccessed`* when `getBytesAccessed` is called.

Alternatively, off-chain monitoring can be used to detect listing owner with high reward uncollected.

Long term, consider that listing owners can be malicious and ensure they can't influence the market.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General

- **Check for `transfer` and `transferFrom` return value.** Currently, these functions are used without checking their return values. However, according to the ERC20 functions, the return value could be false if the tokens are not transferred.
- **Use 10**X instead of plain text for large numerical values.** Large numerical are more difficult to review and are error-prone. For example, Reserve.vy and Listing.vy rely on 1000000000 and 1000000000000000000.
- **Use constants instead of repeated numerical values.** Repeated numerical values are difficult to maintain and are error prone. For example, both Reserve.vy and Listing.vy rely on 1000000000. If it is updated in one file and not the other, the system will break.

Datatrust.vy

- **Implement a `hasPrivilege` method to be consistent with the `MarketToken` and `Voting` code patterns**. Instead of implementing and using a `hasPrivilege` method, Datatrust methods check directly if `msg.sender` is `listing_address` for privileged access to certain methods.
- **Generate suitable events for calls `setDataHash` and `removeDataHash`**. Allowing users to listen to these events mitigates the effects of a potentially malicious Backend.

MarketToken.vy

- **Consider emitting `Transfer(0, to, amount)` upon a minting operation**. This event could be emitted in addition to the Minted event, to follow the ERC20 recommendations.
- **Prevent the address 0x0 from appearing in the `approve`, `decreaseApproval` or `increaseApproval` functions**. ERC20 best practices disallows 0x0 to catch third-party issues earlier.
- **Consider disallowing deposits and withdraws with zero tokens**. If your off-chain code is listening to the `Deposited` and `Withdrawn` events, these corner cases can produce unexpected behavior (e.g., division by zero).

EtherToken.vy

- **Prevent the address 0x0 from appearing in the `approve`, `decreaseApproval` or `increaseApproval` functions**. ERC20 best practices disallows 0x0 to catch third-party issues earlier.

Voting.vy

- **Remove the privileged status from `Reserve` contract in `Voting`.** Voting contract gives privileged status to Reserve contract, which does not interact with the Voting contract at all. This makes the privilege given to Reserve in Voting unnecessary.

Listing.vy

- **Replace `the ApplicationFailed` event in Listing.vy#159 by a new event: `EmptyDataHash.`** `ApplicationFailed` is incorrectly emitted: the code path corresponds to an empty hash failure, not an application failure.

C. Vyper does not check for function ID collisions

During this engagement, we discovered a security issue in the Vyper compiler that can be used to create backdoors or lead to broken contracts. We have reported this issue to the Ethereum Foundation, and a fix has been proposed for the Vyper compiler. In the interim, we have created a Slither detector to identify instances of this bug in deployed code.

Description
To call a smart contract function, the first four bytes of the keccack of the function signature is used as an ID. As only four bytes are used, a collision can occur. Vyper does not check for collisions. As a result, calling a function via its signature might lead to calling an unintended function with the same signature.

For example, in the following contract:

```
@public
@constant
def gsf() -> uint256:
   return 1

@public
@constant
def tgeo()  -> uint256:
   return 2
```

Calling **tgeo()** will return 1 instead of 2.
Indeed, in the bytecode:

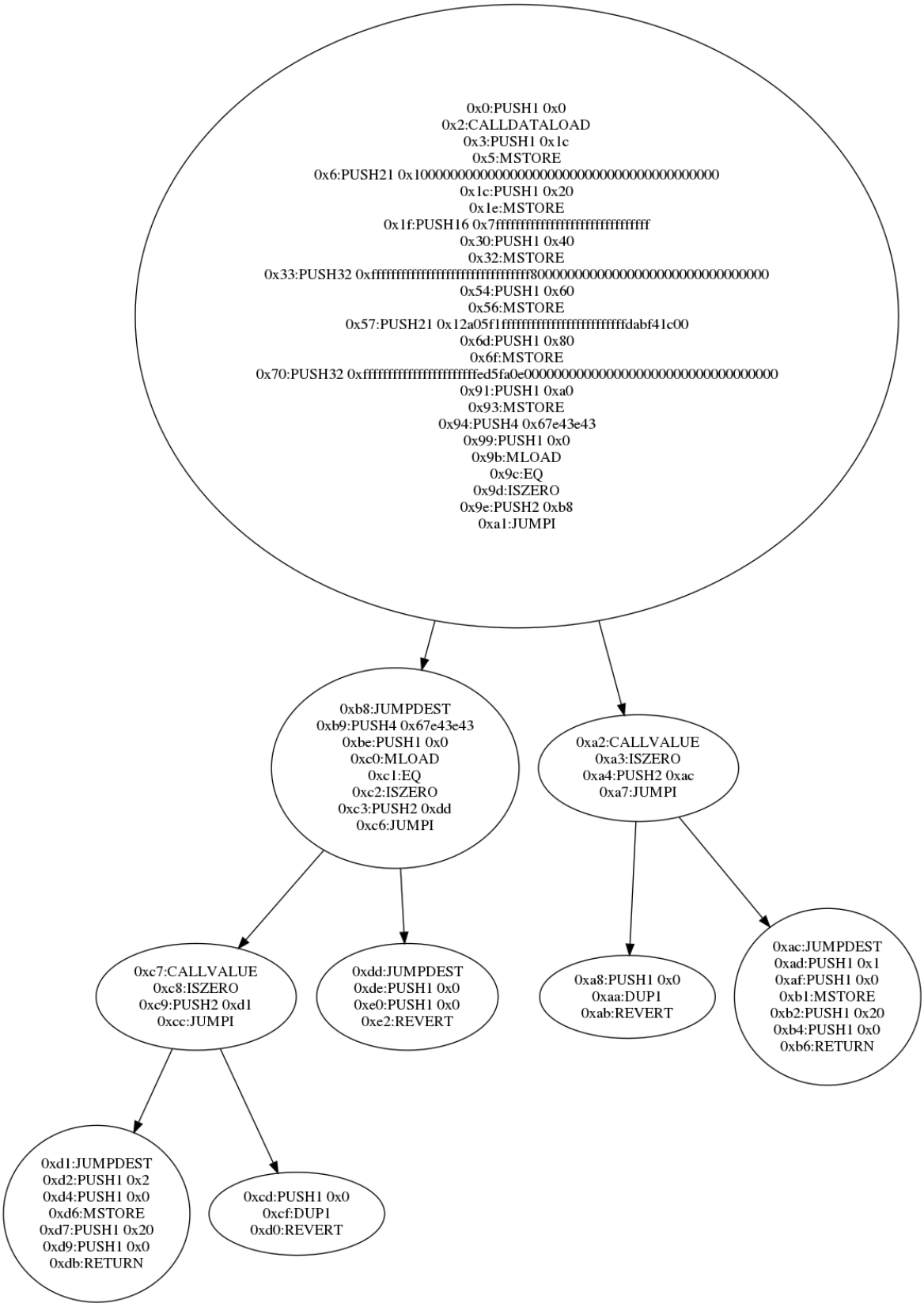

Figure 1: Bytecode with vyper 0.1.0b10 and [evm_cfg_builder]

0x67e43e43 is used two times in the dispatcher. It is the function ID of both `gsf()` and `tgeo()`.

Exploit scenario
Bob creates a contract with a crafted function ID collision. Bob uses the collision to hide a backdoor. Bob deploys the contract and steals ethers from the contract's users.

Recommendation
Check for function ID collision at compile time.

Note that solc checks for this condition and is, therefore, not affected by the underlying issue.

D. Check for outdated dependencies with Slither

Vyper does not handle importing files. As a result, the interface of external contracts must be manually copied and pasted. This process is error prone and difficult to maintain.

Trail of Bits built a new tool on top of Slither, slither-dependencies, that checks the correct interface across contracts.

How to use it

```
$ slither-dependencies Datatrust.vy EtherToken.vy Listing.vy MarketToken.vy
Parameterizer.vy Reserve.vy Voting.vy
INFO:Slither-dependencies:No error found
```

`slither-dependencies` can be added to the CI to ensure that all the contracts stay up to date.

E. Trust Model

The data market relies fundamentally on trust. As in any system with untrusted actors, without guarantees on the quality and reliability of data listings and delivery, market participants will not want to engage with each other.

As you'd expect, many potential issues could arise if malicious users gain trust. For example, in the current design, a malicious Backend could collect payment without delivering any data or delivering the wrong data. In such a scenario, Buyers will likely avoid such a Backend in future.

There are multiple ways to address the trust model in data markets. In this appendix, we outline their advantages and disadvantages.

However, we must note that since this is a very hard problem, no solution is perfect. Computable should review the advantages and disadvantages of each solution and decide which mitigation to implement, according to the time and resources available before deploying the contracts.

Crypto-economics

In the current iteration of design and implementation, market participants—Makers, Buyers, Backends and Patrons—are mostly trusted to behave in a manner which is crypto-economically rational (i.e., if they misbehave, then other market participants are expected not to engage with them in future, thereby causing them financial loss).

Advantages

Lower on-chain technical complexity and no overhead.

Disadvantages

Market participants can choose not to interact with misbehaving entities only if they can identify them successfully. In case of a misbehaving Backend, even if Buyers communicate and coordinate off-chain to do so, Backends can simply re-register with different addresses and URLs, thereby changing their identities and defeating any informal blacklisting.

Challenge-Response

The current design allows participants to challenge Maker listings and Backend registrations. A similar challenge-response process could be added for Backend deliveries. Buyers could then challenge a Backend upon receiving invalid or incomplete data.

However, similar to other voting scenarios, it is not clear what information participants use to decide their votes. For example, how would voters decide to back the Buyer or the Backend when a Buyer complains of not receiving data from a Backend?

Advantages

Straightforward extension to current implementation and low overhead.

Disadvantages

Increases on-chain technical complexity and codebase by introducing another challenge voting round.

Atomic data-delivery vs. payment

Atomic data delivery relies on a commit-reveal schema. The Datatrust sends the delivery encrypted to the buyer, and it receives a fee only after revealing the encryption key on-chain.
A detailed explanation of this approach is available here: https://forum.computable.io/t/towards-atomic-delivery-of-data/62.

Advantages

Protocol-level guarantee.

Disadvantages

Significantly increases on-chain technical complexity. New protocol has to be specified, implemented, and audited.

Reputation system

A reputation system can be implemented using existing contracts (e.g., a Buyer gives both the Maker and the Backend a five-star-based rating after receiving the dataset). This rating can be tracked on-chain and made accessible to Buyers before they engage with a Maker listing or a Backend. This can also be implemented off-chain.

Advantages

Well-known system for improving quality of marketplaces.

Disadvantages

Increases on-chain technical complexity. Off-chain implementation introduces Computable as a trusted party for enforcing reputation. It opens the door to new issues such as dealing with fake reviews.

F. Formal verification using Manticore

Trail of Bits used Manticore, our open-source dynamic EVM analysis tool that takes advantage of symbolic execution, to find and verify certain properties of the Computable contracts. Symbolic execution allows us to explore program behavior in a broader way than classical testing methods, such as fuzzing.

Trail of Bits used Manticore to verify certain behaviors of the Computable contracts. For instance, we used our tool for the TOB-Computable-004 finding. The script checks that the Datatrust contract initialization allows invalid values in the `setPrivileged` call, as shown in Figure E.1.

```
from manticore.ethereum import ManticoreEVM
from manticore.utils import config
from manticore.core.smtlib.expression import BoolAnd, BoolNot

from initialize_market import initialize, DEFAULT_PARAMETERS

consts = config.get_group("evm")
consts.oog = "ignore"

################ Script ########################
m = ManticoreEVM()
market_parameters = DEFAULT_PARAMETERS
market = initialize(m, parameters=market_parameters, only_create=True)

arg_1 = m.make_symbolic_value(name="mt_arg_1")
arg_2 = m.make_symbolic_value(name="mt_arg_2")

# reserve: address, listing: address
market.market_token_contract.setPrivileged(arg_1, arg_2, caller=market.market_token_owner)

for state in m.ready_states:
   m.generate_testcase(state, name="BugFound", only_if= BoolAnd(arg_1 == 0, BoolNot (arg_2
== 0)))
   m.generate_testcase(state, name="BugFound", only_if= BoolAnd(BoolNot(arg_1 == 0), arg_2
== 0))

print(f"[+] Look for results in {m.workspace}"))
```

Figure E.1: Manticore script to symbolically execute `Datatrust.setPrivileged`.

This script will try to find calls to `setPrivileged` that do not revert where at least one of the arguments is the 0x0 address. In that case, it will produce the complete list of transactions to trigger the issue:

```
...

Transactions No. 7
Type: CALL (0)
From: normal0(0x5eb6de38ee506f72e2e6d26f6ed55267cf58998f)
To: contract0(0x7edbde3ee3953c6f59b0cac65f360eb24d1979a9)
Value: 0
Gas used: 281474976710655
Data:
0xb01e369f0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001 (*)
Return_data: 0x

Function call:
setPrivileged(0,1) -> STOP (*)
```

Figure E.2: Part of `BugFound_00000000.tx` file with the list of transactions to trigger TOB-Computable-004.

Our scripts use a small Python library we built with reusable components that make the key steps (market initialization, listing creation, etc.) modular and allow us to quickly prototype code verification on the Computable codebase.

We created scripts to verify issues TOB-Computable-001, TOB-Computable-002, TOB-Computable-004, TOB-Computable-005, TOB-Computable-007, TOB-Computable-008, TOB-Computable-012, TOB-Computable-013, TOB-Computable-017 and TOB-Computable-018.

However, not all the scripts use symbolic values, like the one in Figure E.1.
In some cases, we want to keep the script running in a reasonable time, instead of taking hours to obtain results. In other cases, there are more fundamental issues affecting the use symbolic execution. For instance, Trail of Bits used the following script to concretely verify that an attacker can block the creation of a listing (TOB-Computable-001):

```
from manticore.ethereum import ManticoreEVM
from manticore.utils import config
```

```
from initialize_market import initialize, DEFAULT_PARAMETERS
from constants import ONE_ETH

consts = config.get_group("evm")
consts.oog = "ignore"

################ Script ######################

m = ManticoreEVM()

market_parameters = DEFAULT_PARAMETERS
market = initialize(m, parameters=market_parameters)

user_balance = 100 * ONE_ETH
user_account = m.create_account(balance=user_balance)
print("[+] Creating a user account", hex(user_account.address))

attacker_balance = 100 * ONE_ETH
attacker_account = m.create_account(balance=attacker_balance)
print("[+] Creating a attacker account", hex(attacker_account.address))

user_list_hash = "my_list_hash"
market.listing_contract.list(user_list_hash, caller=attacker_account)
market.listing_contract.list(user_list_hash, caller=user_account)


m.finalize()
print(f"[+] Look for results in {m.workspace}")
```

Figure E.3: Manticore script to reproduce TOB-Computate-001.

While it is possible to use a symbolic value for the attacker input when he is calling `Listing.list`, Manticore will fail to detect that it requires the fixed value `"my_list_hash"` to succeed and produce a revert in the following transaction.

We determined the cause of this behavior is due to the use of a design pattern where Keccak hashes (e.g., candidates) are stored on-chain. The values of these should be stored off-chain, passed as function arguments where necessary, and validated against the on-chain hash. Unfortunately, since Keccak-256 is cryptographically secure, it is intractable for symbolic execution tools like Manticore to reason over non-concrete values in the encoded, packed, and hashed data structures.

The Manticore team designed a workaround for this issue, but in this case, the value needed by the attacker (e.g. `"my_lish_hash"`) is unknown when Manticore generates the path constraints: it is only revealed in the future transactions.

We recently identified this issue and the Manticore team has prepared an enhancement to overcome this issue. However, it was not ready to use during the timeframe of this audit.