Orchid Network Protocol Audit

Date	November 2019
Lead Auditor	Gonçalo Sá
Co-auditors	Steve Marx

- 1 Summary
- 2 Audit Scope
- 3 System Overview
- 4 Key Observations/Recommendations
- 5 Security Specification
  - 5.1 Actors
  - 5.2 Trust Model
  - 5.3 Important Security Properties
- 6 Issues
  - 6.1 Staking node can be inappropriately removed from the tree Critical ✓ Fixed
  - 6.2 Verifiers need to be pure, but it’s very difficult to validate pureness Medium ✓ Fixed
  - 6.3 Simplify the logic in `OrchidDirectory.pull()` Medium ✓ Fixed
  - 6.4 Remove unnecessary `address payable` Minor Won't Fix
  - 6.5 Use consistent `staker` , `stakee` ordering in `OrchidDirectory` Minor ✓ Fixed
  - 6.6 Use more descriptive function and variable names Minor Won't Fix
  - 6.7 In `OrchidDirectory.step()` and `OrchidDirectory.lift()` , use a signed `amount` Minor Won't Fix
  - 6.8 Document that math in `OrchidDirectory` assumes a maximum number of tokens Minor ✓ Fixed
  - 6.9 Unneeded named return parameter Minor ✓ Fixed
  - 6.10 Improve function visibility Minor ✓ Fixed
- 7 Tool-Based Analysis
  - 7.1 MythX
  - 7.2 Ethlint
  - 7.3 Surya

- 8 Sūrya's Description Report
  - 8.1 Files Description Table
  - 8.2 Contracts Description Table
  - 8.3 Legend
- Appendix 1 - Disclosure

1 Summary

ConsenSys Diligence conducted a security audit on Orchid Labs' network protocol, more specifically, the Ethereum smart contracts that are part of the bigger network protocol codebase.

Orchid Labs was founded with a mission to create a more open and more accessible internet. As part of this quest they created the Orchid network protocol, a truly decentralized VPN service.

The way the Orchid network achieves true decentralization is by allowing anyone to join the network as a bandwidth provider and anyone to use the service as an end user. This permissionless environment is achievable with the usage of programmatic, probabilistic micropayments on top of Ethereum, the focus of this audit.

2 Audit Scope

This audit covered the following files:

File	SHA-1 hash
contracts/curator.sol	5ea6c8374cec289bcf4dfe1adb3bb157ca9bab74
contracts/directory.sol	811ab3b049d570c4236ee965d76ed1a9f5cb929e
contracts/location.sol	1ea56960f41ca3a299c4fd35fab9ef1fdd494d5b
contracts/lottery.sol	e63f3c86b3abba57d0a7e3ca36436bfee4d9ac1b
contracts/token.sol	faf15f117ac160641adfe56c2a01ad14bff931f3

The audit activities can be grouped into the following three broad categories:

1. **Security:** Identifying security related issues within the contract.
2. **Architecture:** Evaluating the system architecture through the lens of established smart contract best practices.

3. **Code quality:** A full review of the contract source code. The primary areas of focus include:
    - Correctness
    - Readability
    - Scalability
    - Code complexity
    - Quality of test coverage

3 System Overview

Orchid network is comprised of two main on-chain components, `OrchidDirectory` and `OrchidLottery`. They both play important roles in setting up the proper incentives for the correct functioning of the bandwidth market of the VPN product.

`OrchidDirectory` keeps track of staked OXT tokens (Orchid's native token). The staked amounts are held in a binary tree that allows for easy weighted random selection by clients. Clients then purchase bandwidth from the selected stakee.

`OrchidLottery` is an implementation of probabilistic micropayments. These are used to stream payments to bandwidth providers.

These two components are completely independent of each other and meant to be used for the VPN client in different stages of the product usage.

A rough outline of the usage flow pertaining to these on-chain components is as follows:

1. The VPN client starts by engaging with an Ethereum client that runs the selection routine for the nodes by running code present in `OrchidDirectory`. This step can remain fully trustless by asking the Ethereum client (which may be untrusted) for merkle proofs of the selected storage blocks (and the blocks themselves). This effectively makes `OrchidDirectory.scan()`, which performs the random weighted selection, more of a reference implementation than a fundamental part of the staking infrastructure.
2. After the traffic route is set up, clients need to be able to pay for packets of data as they're being forwarded. This requires a payment system that enables low-cost micropayments. This is made possible (i.e., trustlessly enforced) by the `OrchidLottery` smart contract, a probabilistic micropayments implementation. `OrchidLottery` holds the end user's funds that are meant to be disbursed to

bandwidth providers, as well as an escrow amount to be burned as a penalty in the case of insufficient available payment funds.

A probabilistic micropayment is a message, signed by the payer, that behaves very much like a lottery ticket. It entitles the recipient of the message to a possible payment of a certain amount with a given probability. This payment is enforced on-chain. Although each payment is probabilistic, over a large number of such payments, the actual amount paid trends towards the expected value.

Creating these probabilistic micropayments is done offline and is computationally inexpensive, so it can be used to pay for a continuous stream of packets without disruption or the need to wait for on-chain confirmations.

4 Key Observations/Recommendations

- The *usage of the Ethereum blockchain* within the Orchid network protocol is *an exceptional example* of decentralized coordination with on-chain enforcement. 👏
- The *formal modeling of Orchid's network market in the presented whitepaper is impressively thorough* and shows the amount of consideration and thought that went into the product planning.
- The *code lacks detailed documentation*. The whitepaper explains the algorithms at a high level, but there is no documentation that explains the code itself. A good start would be clear variable and function names as well as explanatory code comments.
- A good test suite enables the early suppression of bugs and protection against future bugs introduced by composability, i.e., adding new features to the code that breaks some security assumption of a previously written section. The *codebase is lacking in tests*.
- Good code coverage helps immensely to catch control flow errors. While high code coverage gives no guarantees of algorithmic correctness, a lack of coverage usually allows for control flow bugs to exist. Two issues found during the audit would have been discovered during development by measuring code coverage. The Diligence team recommends *integrating a code coverage tool and achieving 100% code coverage*.

5 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to

identify specific security properties that were validated by the audit team.

5.1 Actors

The relevant actors are as follows:

- **VPN users** – Users find providers via the directory and purchase bandwidth from those providers via micropayments.
- **Bandwidth providers** – Providers are advertised in the directory and sell bandwidth via micropayments.
- **Stakers** – To mitigate certain kinds of attacks, the VPN software prefers bandwidth providers with more Orchid tokens (OXT) at stake. Stakers stake their tokens to increase the likelihood that a given bandwidth provider will be selected.
- **Micropayment funders** – Anyone can establish a fund for micropayments.
- **Micropayment signers** – Signers authorize micropayments. There are potentially many signers for a given fund.
- **Orchid** – Orchid itself deploys the smart contracts but retains no special privileges.

5.2 Trust Model

In any smart contract system, it's important to identify what trust is expected/required between various actors.

- None of the participants need to trust Orchid because Orchid retains no special privileges after contract deployment.
- A micropayment funder authorizes certain signers to act on their behalf, and those signers must be trusted by the funder to spend up to the total amount in the fund.
- Users and bandwidth providers need to trust each other very little. The risk for the bandwidth provider is that they provide forwarding service but aren't properly paid, and the risk for the user is that they pay but do not receive service. Because the micropayments are very small, this risk is negligible.

Selecting a Bandwidth Provider

Selecting a bandwidth provider is an adversarial environment. The Orchid network protocol is built in such a way that the client should not need to trust the nodes that traffic is routed through.

The bandwidth consumer should be confident that when choosing a node, that node is incentivized to behave well. The likelihood that a given bandwidth provider is chosen is directly proportional to the amount of OXT staked on that provider's behalf. The built-in withdrawal delay creates an opportunity cost for stakers. This disincentivizes malicious staking behavior.

Since the method for node selection currently requires offloading the `scan()` routine to an external Ethereum client, the client can verify correctness of the response by requiring merkle proofs of storage lookups.

There is also a secondary filtering mechanism that the client can use after the initial pool selection. This filtering is based on available metadata for each node (e.g., exit geolocation, latency/ping, node whitelists, etc.). While this does introduce some variability in the intended choosing probabilities, that variability is small enough to be ignored.

Probabilistic Micropayments

The probabilistic micropayments environment (a.k.a. the lottery) is adversarial as well. There is no trust assumed between the two actors for the correct functioning of the system.

Routing nodes (a.k.a. "bandwidth providers") should have guarantees that the agreed upon probability of the ticket is enforced by the system. A random number is chosen by combining seeds from both the payer and the recipient. If that random number falls in a certain range (`ratio` in the code), the payment is transferred.

The protection against double spending and misbehavior from end users is enforced in the way of a an escrow fund on the payer's side. The higher the escrow amount the bigger the incentive against misbehavior. The slashed amount is irretrievably left in the smart contract (i.e., "burned").

5.3 Important Security Properties

`OrchidDirectory`

In the directory implementation we identified the following properties that should not be violated:

- Node selection is properly randomized:

    - A node's likelihood of being chosen is directly proportional to the size of its associated stake.
    - It is impossible to otherwise influence the outcome of node selection.
- Node stakes can be withdrawn only after the built-in delay.
- There are no lock-up conditions:
    - The tree cannot get so big that operations requiring tree traversable exceed block gas limits.
    - The algorithm never creates orphaned nodes or otherwise violates proper tree structure.
- External actors cannot interfere with the system:
    - No one aside from node stakers can remove a node from the tree.
    - No external actor can affect the tree shape.

`OrchidLottery`

In the lottery implementation we identified the following properties that should not be violated:

- The probabilities for ticket payout are correctly enforced by the smart contract.
- Only a winning ticket recipient can redeem funds.
- A winning ticket can only be redeemed once.
- Penalty escrows are only slashed in the case of legitimate payouts exceeding the available funds.
- Payouts decay linearly according to their `start` and `range` parameters.
- Micropayment signers can only spend from funds they were authorized for.
- There are no lock-up conditions:
    - The payout routine is not affected by how many lotteries exist.
    - The payout routine is not affected by how many pots exist.
- External actors cannot interfere with the system:
    - No one but the respective funder or signer can take funds out of a pot.
    - No one but the respective funder can further fund a pot.

6 Issues

Each issue has an assigned severity:

- **Minor** issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- **Medium** issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- **Major** issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 Staking node can be inappropriately removed from the tree

✓ Fixed

Resolution

This is fixed in https://github.com/OrchidProtocol/orchid/commit/8c586f22c54c20d24a066caf4c0efa9ce16

Description

The following code in `OrchidDirectory.pull()` is responsible for reattaching a child from a removed tree node:

code/dir-ethereum/directory.sol:L275-L281

```
if (name(stake.left_) == key) {
    current.right_ = stake.right_;
    current.after_ = stake.after_;
} else {
    current.left_ = stake.left_;
    current.before_ = stake.before_;
}
```

The condition `name(stake.left_) == key` can never hold because `key` is the key for `stake` itself.

The result of this bug is somewhat catastrophic. The child is not reattached, but it still has a link to the rest of the tree via its 'parent_' pointer. This means reducing the stake of that node can underflow the ancestors' before/after amounts, leading to improper random selection or failing altogether.

The node replacing the removed node also ends up with itself as a child, which violates the basic tree structure and is again likely to produce integer underflows and other failures.

Recommendation

As a simple fix, use `if(name(stake.left_) == name(last))` as already suggested by the development team when this bug was first shared.

Two suggestions for better long-term fixes:

1. Use a strict interface for tree operations. It should be impossible to update a node's parent without simultaneously updating that parent's child pointer.
2. As suggested in (issue 6.3), simplify the logic in `pull()` to avoid this logic altogether.

6.2 Verifiers need to be pure, but it's very difficult to validate pureness

Medium **✓ Fixed**

Resolution

This is addressed in https://github.com/OrchidProtocol/orchid/commit/1b405fb1096f6fd18792269453f7dd8ae17
With this change, the contract checks that the verifier's code doesn't change (via `extcodehash`). If the code *does* change, the contract "fails open" by skipping the verifier and allowing all payments.

Because the code can no longer change, the server can use the (relatively) simple method of executing the contract locally and only allowing a whitelist of opcodes that don't depend on or modify state.

The server already has mitigations for denial of service attacks, including limiting the amount of computing resources that can be used for validating code.

Description

After the initial audit, a “verifier” was introduced to the `OrchidLottery` code. Each `Pot` can have an associated `OrchidVerifier`. This is a contract with a `good()` function that accepts three parameters:

code/lot-ethereum/lottery.sol:L28

```
function good(bytes calldata shared, address target, bytes calldata receipt) e
```

The verifier returns a boolean indicating whether a given micropayment should be allowed or not. An example use case is a verifier that only allows certain target addresses to be paid. In this case, `shared` (a single value for a given `Pot`) is a merkle root, `target` is (as always) the address being paid, and `receipt` (specified by the payment recipient) is a merkle proof that the target address is within the merkle tree with the given root.

A server providing bandwidth needs to know whether to accept a certain receipt. To do that, it needs to know that *at some time in the future*, a call to the verifier’s `good()` function with a particular set of parameters will return `true`. The proposed scheme for determining that is for the server to run the contract’s code locally and ensure that it returns `true` and that it doesn’t execute any EVM opcodes that would read state. This prevents, for example, a contract from returning `true` until a certain timestamp and then start returning `false`. If a contract could do that, the server would be tricked into providing bandwidth without then receiving payment.

Unfortunately, this simple scheme is insufficient. As a simple example, a verifier contract could be created with the `CREATE2` opcode. It could be demonstrated that it reads no state when `good()` is called. Then the contract could be destroyed by calling a function that performs a `SELFDESTRUCT`, and it could be replaced via another `CREATE2` call with different code.

This could be mitigated by rejecting any verifier contract that contains the `SELFDESTRUCT` opcode, but this would also catch harmless occurrences of that particular byte. https://gist.github.com/Arachnid/e8f0638dc9f5687ff8170a95c47eac1e attempts to find `SELFDESTRUCT` opcodes but fails to account for tricks where the `SELFDESTRUCT` appears to be data but can actually be executed. (See Recmo’s comment.) In general, this approach is difficult to get right and probably requires full data flow analysis to be correct.

Another possible mitigation is to use a factory contract to deploy the verifiers, guaranteeing that they're not created with `CREATE2`. This should render `SELFDESTRUCT` harmless, but there's no guarantee that future forks won't introduce new vectors here.

Finally, requiring servers to implement potentially complex contract validation opens up potential for denial-of-service attacks. A server will have to implement mitigations to prevent repeatedly checking the same verifier or spending inordinate resources checking a maliciously crafted contract (e.g. one with high branching factors).

Recommendation

The verifiers add quite a bit of complexity and risk. We recommend looking for an alternative approach, such as including a small number of vetted verifiers (e.g. a merkle proof verifier) or having servers use their own "allow list" for verifiers that they trust.

6.3 Simplify the logic in `OrchidDirectory.pull()` Medium ✓ Fixed

Resolution

This was addressed in the following commits:

- https://github.com/OrchidProtocol/orchid/commit/0ad24846e4320cc22c679d17c7239ff
- https://github.com/OrchidProtocol/orchid/commit/8b3e8217ea3a5a967f0965f291491d7
- https://github.com/OrchidProtocol/orchid/commit/affbf937c6b8414b63b63b1ad76cf38a
- https://github.com/OrchidProtocol/orchid/commit/e506c0f216c752e64475bc8eda58327
- https://github.com/OrchidProtocol/orchid/commit/f864e60f1aa58839133379ae46c777d

Description

`pull()` is the most complex function in `OrchidDirectory`, due to its need to handle removing a node altogether when its stake amount reaches 0.

The current logic for removing an interior node is roughly this:

- Given a node to be remove called `old`, walk down the tree, always stepping towards the "heavier" (in terms of total stake) subtree, until you reach a leaf node (called `target`).

- If `target` is a direct child of `old`:
  - Set `target` to be a child of `old.parent`.
  - Move the remaining child of `old` to be under `target`.
- If `target` is not a direct child of `old`:
  - Swap `target` and `old` in the tree.
  - Walk up the tree from `old` (now a leaf node) to `target` to subtract `target`'s staked amount from the nodes in between.
  - Detach `old` from the tree.

The code for this is fairly complex, and one serious bug (issue 6.1) was identified in this code.

This logic can be simplified by combining the two cases (direct child and not) and thinking of it as roughly a two-step operation of "detach leaf node" and "replace interior node with leaf node".

- Given a node to be removed called `old`, walk the tree to find `target` as before.
- Walk back up to `old`, subtracting `target`'s staked amount from the nodes in between.
- Detach `target` from the tree.
- Replace `old` with `target`.

(Note that in the code, "old" above is called `stake` and "target" is called `current`.)

Recommendation

Replace this code:

code/dir-ethereum/directory.sol:L266-L297

```
bytes32 direct = current.parent_;
copy(pivot, last);
current.parent_ = stake.parent_;

if (direct == key) {
    Primary storage other = stake.before_ > stake.after_ ? stake.right_ : stak
    if (!nope(other))
        stakes_[name(other)].parent_ = name(last);
```

```
    if (name(stake.left_) == key) {
        current.right_ = stake.right_;
        current.after_ = stake.after_;
    } else {
        current.left_ = stake.left_;
        current.before_ = stake.before_;
    }
} else {
    if (!nope(stake.left_))
        stakes_[name(stake.left_)].parent_ = name(last);
    if (!nope(stake.right_))
        stakes_[name(stake.right_)].parent_ = name(last);

    current.right_ = stake.right_;
    current.after_ = stake.after_;

    current.left_ = stake.left_;
    current.before_ = stake.before_;

    stake.parent_ = direct;
    copy(last, staker, stakee);
    step(key, stake, -current.amount_, current.parent_);
    kill(last);
```

with something like this code:

```
// Remember this key so we can update `pivot` later
bytes32 currentKey = name(last);

// Remove `current` from the subtree rooted at `stake`
step(currentKey, current, -current.amount_, stake.parent_);
kill(last);

// Replace `stake` with `current`
current.left_ = stake.left_;
if (!nope(current.left_))
    stakes_[name(current.left_)].parent_ = currentKey;
current.right_ = stake.right_;
if (!nope(current.right_))
```

```
    stakes_[name(current.right_)].parent_ = currentKey;
current.before_ = stake.before_;
current.after_ = stake.after_;
current.parent_ = stake.parent_;
pivot.value_ = currentKey; // `pivot` was parent's pointer to `stake`
```

6.4 Remove unnecessary `address payable` Minor Won't Fix

Resolution

The development team decided to leave this as-is. `address payable` is simply advisory. It marks a parameter as one that will have tokens transferred to it.

Description

The `address payable` type is only needed for transferring ether to an address. The `OrchidDirectory` and `OrchidLottery` contracts work with tokens, not ether, so there's no need for any parameters to be of type `address payable`.

Recommendation

Use simply `address` instead of `address payable` everywhere.

6.5 Use consistent `staker`, `stakee` ordering in `OrchidDirectory` Minor ✓ Fixed

Resolution

This is fixed in https://github.com/OrchidProtocol/orchid/commit/1cfef8881d36ef31258a7dbce2e617cf5c3a
3659f7467766ff8d915be9dbd8d05c8e.

Description

code/dir-ethereum/directory.sol:L156

```
function lift(bytes32 key, Stake storage stake, uint128 amount, address stakee
```

`OrchidDirectory.lift()` has a parameter `stakee` that precedes `staker`, while the rest of the code always places `staker` first. Because Solidity doesn't have named parameters, it's a good idea to use a consistent ordering to avoid mistakes.

Recommendation

Switch `lift()` to follow the "staker then stakee" ordering convention of the rest of the contract.

6.6 Use more descriptive function and variable names Minor Won't Fix

Resolution

This issue is about readability. Even though the audit team firmly believes that improved readability would increase trust in Orchid from its clients, this is not a correctness issue.

The Orchid team believes that making this change, particularly this late in their development cycle, would be too risky. The development team is very familiar with the current terminology, and bugs may accidentally be introduced with the change.

Description

Throughout `OrchidDirectory` and `OrchidLottery`, function and variable names are quite obscure. This makes it harder for a reader to understand the code.

Examples

- `OrchidDirectory`:
  - `heft()` returns the total staked for a given stakee (perhaps `totalForStakee()`)
  - `Primary` is a pointer to a tree node (perhaps `NodePointer`), and its member `value_` could be named `key`

- `name()` gives the key for a given (staker, stakee) pair or a `Primary` (perhaps `getKey()`)
- `copy()` writes a key to a node pointer (probably better to remove this and just do `pointer.key = ...`)
- `kill()` sets a node pointer to zero (probably better to just remove this and use `delete pointer`)
- `nope()` checks whether a node pointer exists (probably better to just do `pointer.key == 0`)
- `have()` returns the total number of staked tokens (perhaps `totalStaked`)
- `scan()` finds a node, given a random 128-bit number (perhaps `selectNode(uint128 random)`)
- `turn()` is only used in one place and is likely better just inlined
- `step()` walks up a subtree, adjusting before/after amounts along the way (perhaps `propagate()` or `bubbleUp()`)
- `lift()` updates the stake for a given node and then calls `step()` (perhaps `updateNodeStake()`)
- `more()` is really just the body for `push()`, so it should probably be moved inside `push()` instead
- `push()` is the external method for staking (perhaps `increaseStake()` or just `stake()`)
- `wait()` increases the withdrawal delay for the sender's stake for a given stakee (`increaseDelay()`)
- `Pending` could be called `PendingWithdrawal`
- `take()` could be called `completeWithdrawal()`
- `stop()` could be called `cancelWithdrawal()`
- `delay_` could be `withdrawalDelay`
- `pull()` decreases stake and establishes a pending withdrawal (perhaps `decreaseStake()`, `unstake()` or `startWithdrawal()`)
- Within `pull()`:
  - `pivot` could be `pointerToStake`
  - `last` could be `pointerToLeaf`
  - `current` could be `leaf`
  - `direct` could be `leafParent`
  - `other` could be `sibling`

- `OrchidLottery`:
  - `Pot` could perhaps be `Fund`
  - `send()` just emits an `Update` event (perhaps `log()` or `logUpdate()`)
  - `Track` is a struct that keeps track of a ticket that has already been redeemed to prevent replay (perhaps `RedeemedTicket`)
  - `kill()` is overloaded to delete funds and used tickets (perhaps `deleteFund()` and `forgetTicket()`)
  - `take()` could be called `transferTokens()`
  - `grab()` redeems a winning ticket (perhaps `redeem()` or `redeemTicket()`)
  - `give()` and `pull()` both transfer tokens from a given `Pot`, but one is used by the signer and one by the funder. Perhaps better would be a single `transferFromPot(address funder, address signer, address target, uint128 amount)` with `require(msg.sender == funder || msg.sender == signer)`.
  - `warn()` could be `startWithdrawal()`
  - `lock()` could be `cancelWithdrawal()`
  - `pull()` could be `completeWithdrawal()`

Recommendation

Consider using longer, more descriptive names to make it easier to understand the code. Where there's no particularly good name, add comments explaining the meaning.

6.7 In `OrchidDirectory.step()` and `OrchidDirectory.lift()`, use a signed `amount` Minor Won't Fix

Resolution

The variables in question are now `uint256`s. The amount of type casts that would be needed in case the recommended change was implemented would defeat the purpose of simplification.

Description

`step()` and `lift()` both accept a `uint128` parameter called `amount`. This amount is added to various struct fields, which are also of type `uint128`.

The contract intentionally underflows this `amount` to represent negative numbers. This is roughly equivalent to using a signed integer, except that:

1. Unsigned integers aren't sign extended when they're cast to a larger integer type, so care must be taken to avoid this.
2. Tools that look for integer overflow/underflow will detect this possibility as a bug. It's then hard to determine which overflows are intentional and which are not.

Examples

code/dir-ethereum/directory.sol:L247

```
lift(key, stake, -amount, stakee, staker);
```

code/dir-ethereum/directory.sol:L296

```
step(key, stake, -current.amount_, current.parent_);
```

Recommendation

Use `int128` instead, and ensure that amounts can never exceed the maximum `int128` value. (This is trivially achieved by limiting the total number of tokens that can exist.)

6.8 Document that math in `OrchidDirectory` assumes a maximum number of tokens Minor ✓ Fixed

Resolution

This is fixed in https://github.com/OrchidProtocol/orchid/commit/f2efe427367970de88986af0e58f1fecb901 3659f7467766ff8d915be9dbd8d05c8e by using `uint256` values everywhere. For a compliant ERC20 token, the token's total supply cannot overflow a `uint256`.

Description

`OrchidDirectory` relies on mathematical operations being unable to overflow due to the particular ERC20 token being used being capped at less than `2**128`.

Examples

The following code in `step()` assumes that no before/after amount can reach `2**128`:

code/dir-ethereum/directory.sol:L145-L148

```
if (name(stake.left_) == key)
    stake.before_ += amount;
else
    stake.after_ += amount;
```

The following code in `lift()` assumes that no staked amount (or total amount for a given stakee) can reach `2**128`:

code/dir-ethereum/directory.sol:L157-L164

```
uint128 local = stake.amount_;
local += amount;
stake.amount_ = local;
emit Update(staker, stakee, local);

uint128 global = stakees_[stakee].amount_;
global += amount;
stakees_[stakee].amount_ = global;
```

The following code in `have()` assumes that the total amount staked cannot reach `2**128`:

code/dir-ethereum/directory.sol:L103

```
return stake.before_ + stake.after_ + stake.amount_;
```

Recommendation

Document this assumption in the form of code comments where potential overflows exist.

Consider also `assert` ing the ERC20 token's total supply in the constructor to attempt to block using a token that violates this constraint and/or checking in `push()` that the total amount staked will remain less than `2**128`. This recommendation is in line with the mitigation proposed for issue 6.7.

6.9 Unneeded named return parameter Minor ✓ Fixed

Resolution
Fixed in https://github.com/OrchidProtocol/orchid/commit/21d56d5fc33cbba6838447231a575fee54f

Description

In the `heft` function in the `OrchidDirectory` contract, there is an unused and unneeded named return parameter (that actually instantiates a new variable in memory which is not used).

Remediation

Change `returns (uint128 amount)` to `returns (uint128)`.

6.10 Improve function visibility Minor ✓ Fixed

Resolution
Fixed in https://github.com/OrchidProtocol/orchid/commit/68fb26acd9d3831616b69744967fb9512a

Description

The following methods are not called internally in the token contract and visibility can, therefore, be restricted to `external` rather than `public`. This is more gas efficient

because less code is emitted and data does not need to be copied into memory. It also makes functions a bit simpler to reason about because there's no need to worry about the possibility of internal calls.

- `OrchidDirectory.heft()`
- `OrchidDirectory.scan()`
- `OrchidDirectory.push()`
- `OrchidDirectory.wait()`
- `OrchidDirectory.take()`
- `OrchidDirectory.stop()`
- `OrchidDirectory.pull()`
- `OrchidLocation.move()`
- `OrchidLocation.look()`
- `OrchidLottery.size()`
- `OrchidLottery.keys()`
- `OrchidLottery.seek()`
- `OrchidLottery.look()`
- `OrchidLottery.push()`
- `OrchidLottery.move()`
- `OrchidLottery.kill()`
- `OrchidLottery.grab()`
- `OrchidLottery.pull()`
- `OrchidLottery.warn()`
- `OrchidLottery.lock()`
- `OrchidLottery.pull()`
- `OrchidCurator.list()`
- `OrchidCurator.good()`
- `OrchidUntrusted.good()`

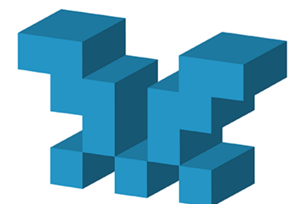
Recommendation

Change visibility of these methods to `external`.

7 Tool-Based Analysis

Several tools were used to perform automated analysis of the reviewed contracts. These issues were reviewed by the audit team, and relevant issues are listed in the Issue Details section.

7.1 MythX



MythX is a security analysis API for Ethereum smart contracts. It performs multiple types of analysis, including fuzzing and symbolic execution, to detect many common vulnerability types. The tool was used for automated vulnerability discovery for all audited contracts and libraries. More details on MythX can be found at mythx.io.

The output of the MythX Pro vulnerability scan was reviewed by the audit team and no vulnerabilities were identified as part of the process.

7.2 Ethlint

Ethlint is an open source project for linting Solidity code. Only security-related issues were reviewed by the audit team.

Below is the raw output of the Ethlint vulnerability scan:

```
contracts/curator.sol
  35:8    warning    Provide an error message for require().    error-reason

contracts/directory.sol
  107:8     warning    Provide an error message for require().          erro
  141:4     error      "step": Avoid assigning to function parameters.    secu
  141:4     error      "step": Avoid assigning to function parameters.    secu
  176:8     warning    Provide an error message for require().          erro
  180:12    warning    Provide an error message for require().          erro
  202:8     warning    Provide an error message for require().          erro
  209:8     warning    Provide an error message for require().          erro
  211:8     warning    Provide an error message for require().          erro
  226:8     warning    Provide an error message for require().          erro
  226:35    warning    Avoid using 'block.timestamp'.                   secu
  228:8     warning    Provide an error message for require().          erro
  233:8     warning    Provide an error message for require().          erro
```

```
  233:35    warning    Avoid using 'block.timestamp'.                     secu
  244:8     warning    Provide an error message for require().            erro
  245:8     warning    Provide an error message for require().            erro
  305:8     warning    Provide an error message for require().            erro
  306:26    warning    Avoid using 'block.timestamp'.                     secu

contracts/location.sol
  38:24    warning    Avoid using 'block.timestamp'.    security/no-block-memb

contracts/lottery.sol
  66:8      warning    Provide an error message for require().            erro
  104:8     warning    Provide an error message for require().            erro
  111:8     warning    Provide an error message for require().            erro
  117:8     warning    Provide an error message for require().            erro
  131:8     warning    Provide an error message for require().            erro
  131:32    warning    Avoid using 'block.timestamp'.                     secu
  140:4     error      "take": Avoid assigning to function parameters.    secu
  153:12    warning    Provide an error message for require().            erro
  156:4     error      "grab": Avoid assigning to function parameters.    secu
  156:4     warning    Line exceeds the limit of 145 characters           max-
  157:8     warning    Provide an error message for require().            erro
  158:8     warning    Provide an error message for require().            erro
  163:12    error      Only use indent of 8 spaces.                       inde
  165:12    error      Only use indent of 8 spaces.                       inde
  166:12    error      Only use indent of 8 spaces.                       inde
  167:12    error      Only use indent of 8 spaces.                       inde
  167:12    warning    Provide an error message for require().            erro
  167:28    warning    Avoid using 'block.timestamp'.                     secu
  168:12    error      Only use indent of 8 spaces.                       inde
  168:12    warning    Provide an error message for require().            erro
  169:12    error      Only use indent of 8 spaces.                       inde
  171:12    error      Only use indent of 8 spaces.                       inde
  172:0     error      Only use indent of 8 spaces.                       inde
  175:20    warning    Avoid using 'block.timestamp'.                     secu
  176:64    warning    Avoid using 'block.timestamp'.                     secu
  182:8     warning    Provide an error message for require().            erro
  200:22    warning    Avoid using 'block.timestamp'.                     secu
  214:8     warning    Provide an error message for require().            erro
  215:8     warning    Provide an error message for require().            erro
  215:31    warning    Avoid using 'block.timestamp'.                     secu
```

```
  219:8     warning    Provide an error message for require().         erro

✖ 12 errors, 38 warnings found.
```

7.3 Surya

Surya is an utility tool for smart contract systems. It provides a number of visual outputs and information about structure of smart contracts. It also supports querying the function call graph in multiple ways to aid in the manual inspection and control flow analysis of contracts.

Below is a complete list of functions with their visibility and modifiers:

8 Sūrya's Description Report

8.1 Files Description Table

File Name	SHA-1 Hash
contracts/curator.sol	5ea6c8374cec289bcf4dfe1adb3bb157ca9bab74
contracts/directory.sol	811ab3b049d570c4236ee965d76ed1a9f5cb929e
contracts/location.sol	1ea56960f41ca3a299c4fd35fab9ef1fdd494d5b
contracts/lottery.sol	e63f3c86b3abba57d0a7e3ca36436bfee4d9ac1b
contracts/token.sol	faf15f117ac160641adfe56c2a01ad14bff931f3

8.2 Contracts Description Table

Contract	Type	Bases		
L	**Function Name**	**Visibility**	**Mutability**	**Modifiers**
OrchidCurator	Implementation			
L	<Constructor>	Public ❗️	🛑	
L	list	Public ❗️	🛑	NO ❗️
L	good	Public ❗️		NO ❗️

Contract	Type	Bases		
OrchidUntrusted	Implementation			
└	good	Public ❗️		NO❗️
IOrchidDirectory	Interface			
└	have	External ❗️		NO❗️
OrchidDirectory	Implementation	IOrchidDirectory		
└	<Constructor>	Public ❗️	🛑	
└	heft	Public ❗️		NO❗️
└	name	Public ❗️		NO❗️
└	name	Private 🔐		
└	copy	Private 🔐	🛑	
└	copy	Private 🔐	🛑	
└	kill	Private 🔐	🛑	
└	nope	Private 🔐		
└	have	Public ❗️		NO❗️
└	scan	Public ❗️		NO❗️
└	turn	Private 🔐		
└	step	Private 🔐	🛑	
└	lift	Private 🔐	🛑	
└	more	Private 🔐	🛑	
└	push	Public ❗️	🛑	NO❗️
└	wait	Public ❗️	🛑	NO❗️
└	take	Public ❗️	🛑	NO❗️
└	stop	Public ❗️	🛑	NO❗️
└	pull	Public ❗️	🛑	NO❗️
OrchidLocation	Implementation			

Contract	Type	Bases		
L	move	Public ❗	🛑	NO❗
L	look	Public ❗		NO❗
OrchidLottery	Implementation			
L	<Constructor>	Public ❗	🛑	
L	send	Private 🔐	🛑	
L	find	Private 🔐		
L	kill	Private 🔐	🛑	
L	size	Public ❗		NO❗
L	keys	Public ❗		NO❗
L	seek	Public ❗		NO❗
L	page	Public ❗		NO❗
L	look	Public ❗		NO❗
L	push	Public ❗	🛑	NO❗
L	move	Public ❗	🛑	NO❗
L	kill	Private 🔐	🛑	
L	kill	Public ❗	🛑	NO❗
L	take	Private 🔐	🛑	
L	grab	Public ❗	🛑	NO❗
L	give	Public ❗	🛑	NO❗
L	pull	Public ❗	🛑	NO❗
L	warn	Public ❗	🛑	NO❗
L	lock	Public ❗	🛑	NO❗
L	pull	Public ❗	🛑	NO❗
OrchidToken	Implementation	ERC20, ERC20Detailed		
L	<Constructor>	Public ❗	🛑	ERC20Detailed

8.3 Legend

Symbol	Meaning
🛑	Function can modify state
💵	Function is payable

Appendix 1 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") – on its website. CD hopes that by making these analyses publicly available, it can help the

blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.