Blur Finance

Smart Contract Security Assessment

January 20, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the Blur Protocol. We had already audited earlier versions of the protocol. The corresponding reports along with the descriptions of the protocol and its architecture overview can be found here (v1.0), here (v1.1) and here.

This audit report covers the recent changes of the contracts of the at-this-time private repository blur-io/contracts-snapshot-not-public of the Blur protocol, at commit hash `d3d2eda3be83ba91c3119540ffdd07a4c185f1d5`.

SETTING & CAVEATS

The audit focussed solely on the delta of changes, which consist of the following additions. The auditors did not re-audit the whole protocol.

- Minor upgrades to the `BlurExchange` contract altering the fee mechanism
- New `Governance` infrastructure

Two auditors worked on the codebase for 3 days on the following contracts:

```
contracts/
├─ BlurExchange.sol
├─ BlurExchange_old.sol
│
└─ governance/
   ├─ BlurAirdrop.sol
   ├─ BlurGovernor.sol
   ├─ BlurToken.sol
   ├─ TokenLockup.sol
   │
   └─ interfaces/
      └─ ITokenLockup.sol
```

The developers' specification of the audit scope can be found verbatim below:

The scope of this audit includes:

- *Minor upgrades to the `BlurExchange` contract to alter the fee mechanism (~50 SLOC)*
- *New governance infrastructure including*
    - *`BlurToken` - BLUR ERC20 contract (~35 SLOC)*
    - *`BlurGovernor` - core governance contract; taken from OpenZeppelin governance (~92 SLOC)*
    - *`TimelockController` - governor execution timelock; taken from OpenZeppelin governance (~215 SLOC)*
    - *`TokenLockup` - removes tokens from circulating supply; unlocking them on a predefined schedule (~112 SLOC)*
    - *`BlurAirdrop` - airdrop distributor (~33 SLOC)*

There have been architectural changes since our last audit of the `BlurExchange` contract. Namely, the `_execute` function is now public, so that it can be the target of a multicall facility, implemented using `delegatecall`. An execution-setup phase precedes external command execution, apparently to prevent the issue of having a single `msg.value` across different delegatecalls. These changes required the introduction of two different levels of reentrancy protection, implemented in modifiers `setupExecution` and `reentrancyGuard`. Although we considered these changes in the course of understanding the flow of execution (and we believe them to be safe), we did not examine them extensively, as they are explicitly outside the scope of the audit.

The `Governance` functionality has been audited in isolation. There is no contract code or external test cases to exercise its full integration with the protocol (i.e., a proposal, subsequent vote, approval of the proposal, and execution). Therefore, there needs to be an assumption that the code is "called correctly". Since the mechanism uses a standard

OpenZeppelin facility, it is reasonable to assume that the front-end or other external agent invokes it as-expected. Generally, the tests for the governance mechanism seem to still be work-in-progress.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

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

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.

MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Schedules already timed-up may not be taken into account if a preceding one hasn't expired yet	**RESOLVED**

The `_computeUnlocked()` function of the `TokenLockup` contract iterates over the schedules to calculate the unlocked amount of tokens based on the schedules which the contract has been initialized with.

However, there is no guarantee that these schedules are in ascending order based on the `endTime` field. As a result, a schedule which expires before its preceding one can lead to the amount of the schedule not being counted until the preceding one expires too. This happens due to the fact that the loop breaks once it reaches a schedule which hasn't expired yet.

`TokenLockup::_computeUnlocked()`

```
function _computeUnlocked(
    uint256 locked,
    uint256 time
) internal view returns (uint256) {
    ...
    for (uint i; i < scheduleLength; i++) {
        uint256 portion = schedule[i].portion;
        uint256 end = schedule[i].endTime;

        // Dedaub: Here the loop breaks once it finds a schedule
        //         that hasn't expired yet
        if (time < end) {
            unlocked += locked * (time - start) * portion /
                ((end - start) * INVERSE_BASIS_POINTS);
            break;
        } else {
            unlocked += locked * portion / INVERSE_BASIS_POINTS;
            start = end;
        }
    }
    return unlocked;
}
```

Hence, it could result in getting incorrect information about the unlocked tokens at any particular moment which can also lead to incorrect calculations of the voting power of the users.

L2	Schedule portions are not checked whether they add up to 100%	**RESOLVED**

Every TokenLockup contract gets a list of schedules upon construction which will release portions of the unallocated tokens. However, there is no check to ensure that the provided portions add up to 100% so that the entire amount of tokens become claimable after an amount of time.

TokenLockup::_computeUnlocked()

```
function _computeUnlocked(
    uint256 locked,
    uint256 time
) internal view returns (uint256) {
    ...

    // Dedaub: This loop iterates over the schedules taking into account
    //         each schedule's portion, but there is no check that they
    //         all add up to 100%
    for (uint i; i < scheduleLength; i++) {
        uint256 portion = schedule[i].portion;
        uint256 end = schedule[i].endTime;

        if (time < end) {
            unlocked += locked * (time - start) * portion /
                ((end - start) * INVERSE_BASIS_POINTS);
            break;
        } else {
            unlocked += locked * portion / INVERSE_BASIS_POINTS;
            start = end;
        }
    }
    return unlocked;
}
```

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	Trusted "owners" can take over all Governor actions	ACKNOWLEDGED

The changes in the audit scope implement a voting-based governance facility. However, accounts that have the role of owner for different contracts can take over/override governance actions. Specific examples include:

- The owner of BlurExchange can directly set the governor.
- The owner of BlurToken can add (at any time) any lockups, containing any tokens. All token balances are summed together for voting purposes, with no check that they are over the same token. In fact, there is no guarantee that a supplied lockup is indeed a lockup and not just any contract that answers to balanceOf.

OTHER / ADVISORY ISSUES:

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

ID	Description	STATUS
A1	The significance of BlurToken::delegates should be clearly documented	**DISMISSED**

The issue was invalidated by the final revision of the code. The delegates function was removed for gas savings. We reiterate our warning about counter-intuitive behavior (without the function) and the need for documentation and user awareness.

The seemingly innocuous view function BlurToken::delegates is central to the correct functioning of the voting process. This should be documented, at least via a highly visible code comment (e.g., “**WARN**”).

Specifically, the function definition is:

BlurTokens::delegates()

```
function delegates(
    address account
) public view override returns (address) {
    address _delegate = ERC20Votes.delegates(account);
    if (_delegate == address(0)) {
        _delegate = account;
    }
    return _delegate;
}
```

This seems to suggest the function is just a no-op convention: an account is itself its delegatee if it would otherwise have none.

However, this logic is crucial in the correct functioning of the OpenZeppelin ERC20Votes protocol. Specifically, the protocol documentation warns:

```
 * By default, token balance **does not account** for voting power.
 * This makes transfers cheaper. The downside is that it
 * **requires users to delegate to themselves** in order to activate
 * checkpoints and have their voting power tracked.
```

The overridden delegates function in BlurToken achieves this exact purpose: causes every token transfer (which calls delegates() in the _afterTokenTransfer hook of the ERC20Votes contract) to update (checkpoint) the voting power of all parties. Without the definition of the delegates function, the behavior would be significantly different:

- a claim from a TokenLockup would result in lower votes than before (because the Blur token balanceOf would increase without being checkpointed into the votes), while the TokenLockup::balanceOf (which is accounted in BlurGovernor::getVotes) would decrease due to the higher totalClaimed;

- correct updates of the voting power would require delegate calls;

- gas consumption of `BlurToken` transfers would be lower.

A2	Possible out-of-bounds access due to lack of length compatibility	**RESOLVED**

The `fund()` function of the `TokenLockup.sol` contract, iterates over the `amounts[]` array for sending the funds to the corresponding recipients. However, the two arrays provided as parameters are not checked for their length compatibility. Thus, if the `amounts[]` array is larger than the `recipients[]` one, the loop could try to access items out of bounds and revert.

A3	Redundant overrides	**RESOLVED**

The `BlurGovernor.sol` contract inherits from several other contracts and some functions should be overridden as they appear in more than one inherited contract.

However, the following functions are not needed to be overridden:

- `votingDelay()`
- `votingPeriod()`
- `quorum(...)`
- `propose(...)`

Moreover, the following contracts are also not needed to be declared in the inherited list as the rest of the contracts already inherit from them:

- `Governor`
- `GovernorVotes`

A4	“Magic number” used in `BlurExchange::setFeeRate()`	**RESOLVED**

Ideally, numeric constants should be visible prominently at the top of a contract, instead of being buried in the code, for easier maintainability and readability. In this case:

`BlurExchange::setFeeRate()`

```
function setFeeRate(uint256 _feeRate) external {
    require(msg.sender == governor,
        "Fee rate can only be set by governor");

    // Dedaub: Magic constant
    require(feeRate <= 250, "Fee cannot be more than 2.5%");
    ...
}
```

A5	Compiler bugs	INFO

The code is compiled with Solidity 0.8.17. Version 0.8.17, at the time of writing, hasn't any known bugs.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.