Dedaub

Security Technology for Smart Contracts

Immunefi

Smart Contract Security Assessment

Date: July 16, 2021

Abstract

Dedaub was commissioned to perform a security audit for part of the Immunefi Protocol contracts, at commit hash a6095b9634ebd9156ba38e2cff19100618ca5bac. Three auditors worked over this codebase for 2 weeks.

Setting and Caveats

The scope of the audit covers

- the (top-level) BugReportNotary and Escrow contracts
- contracts in lib, math, and utils directories
- contracts in the token module (directory)
- contracts in the distributor module (directory)
- contracts in the vesting module (directory).

The audited code base is of large size, at around 3.5KLoC (excluding test and interface code). The audit focused on security, establishing the overall security model and its robustness, and also crypto-economic issues. Functional correctness (e.g., that the calculations are correct) was a secondary priority. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

During the audit, the code was updated, to provide us with documentation. The code edits also included new and updated functionality. We trust that the updated functionality does not affect the findings, i.e., that the code that we inspected when pulling commits to get documentation updates is essentially identical to that at our initial commit hash. In the case of changes that directly address items mentioned in the audit report, we are to perform a re-review before the report is final.

Architecture, Caveats, and Recommendations

The code is technically highly sophisticated and developed with professional attention to detail. It is, however, complex and with no compromises of expressiveness for the sake of simplicity or

verifiability. Therefore, complexity is a threat. We recommend simplifying, especially in areas of marginal importance (e.g., the use of custom quick-sort or Merkle tree computation code, employed only to create a single one-value Merkle tree).

The code is work in progress and our audit should also be considered to reflect a progressive understanding of the code base. There are parts of the code base for which our understanding is imperfect, although we expended significant effort. We bring these elements up as points for documentation improvement, especially in preparation of future audits.

The parts of the code base where we feel our confidence is conditional on elements not fully understood include:

- Complicated math calculations. We inspected but cannot verify the correctness of the "excessive precision calculation of the tropical/solar year" in TokenMinter, or of the Log2Ceil and QuadRound library arithmetic. We note that astronomical calculations of a solar year in seconds are likely unnecessarily complex for the purposes of computing a continuous-from-annualized inflation rate.
- The exact flow of reports/disclosures through the BugReportNotary contract. Although we understand the machinery, there is no detailed documentation of the workflow (e.g., the different stages that a report goes through, constraints of the form "correct triaging happens within X delay of a report; no disclosure of report keys should take place before attestation of Y"). Therefore, there could be errors that the code allows without giving us an indication that they are errors. Similarly, the BugReportNotary is not connected to vesting, therefore flows between the two cannot be verified. There is no way to check, for example, whether an adversary could cause funds to be released to vesting before a final verdict is given.
- The full richness and limitations of the Vesting domain-specific language is hard to reason about. Better documentation, with detailed examples, might help. We list a few instances of erroneous combinations of components in the report, and we learned of more during the audit, but cannot be confident that the list is complete. We may have missed subtle points of information being passed between connectors (e.g., the reason for the use of `getContext` instead of `getState` on the sub-state of a PeriodicVesting controller would have escaped us if it were not documented).

Although partial understanding is not ideal, we do not believe that the above represent likely threats to the security of the audited contracts. (Still, subtle functional correctness issues--e.g., mistakes in vesting calculations--cannot be precluded from resulting in incorrect accounting, and even loss of funds to an attacker.) We encourage more thorough testing to uncover issues.

We find direct security threats due to a flawed security model to be more likely and a greater concern. Therefore we focused on the latter. We considered protocol-level interactions in depth, including threats of an adversary obtaining or escalating privileges.

Vulnerabilities and Functional Issues

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

Category	Description
Critical	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
High	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
Medium	Examples: 1) User or system funds can be lost when third party systems misbehave. 2) DoS, under specific conditions. 3) Part of the functionality becomes unusable due to programming error.
Low	Examples: 1) Breaking important system invariants, but without apparent consequences. 2) Buggy functionality for trusted users where a workaround exists. 3) Security issues which may manifest when the system evolves. 4) Bug that significantly downgrades system's efficiency.

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

Critical Severity

Id	Description	Status
C1	**The adversary can alter the amount in Distributor.deposit**	**Resolved (but since entire VestingNFTReceiver is removed, similar threats need to be considered in the context of the new architecture upon future audits)**

`Distributor::deposit` computes the `withdrawAmount` by comparing the balance before and after the transfer:

```
uint256 initialBalance = _thisBalance(token);
if (token == NATIVE_ASSET) {
  payable(receiver).sendValue(amount);
} else {
  token.safeTransfer(receiver, amount);
}
uint256 finalBalance = _thisBalance(token);
require(initialBalance > finalBalance, "Distributor: did not withdraw");
uint256 withdrawAmount = initialBalance - finalBalance;
```

An adversary who controls the deposit of funds to the distributor can start withdrawing, and deposit funds back to the distributor from within his receive hook. This will cause the distributor to register a possibly much smaller `withdrawAmount` than the amount actually withdrawn.
When used in combination with `vestingNFTReceiver`, an attack can be executed as follows:

- First, the adversary withdraws an amount from vesting into the distributor, by calling `VestingNFTReceiver::withdraw` via `Distributor::call`
- Then the adversary starts withdrawing the same amount from the distributor (even if the amount is larger than his own share)
- From within his receive hook, the adversary releases an equal amount (minus 1 wei) from vesting to the distributor (again by calling `VestingNFTReceiver::withdraw` via `Distributor::call`)
- As a result, the distributor registered a withdrawal of just 1 wei, and the adversary can withdraw again.

Using the above procedure, an adversary with only 1% share can withdraw all funds from the distributor in a single transaction. An exploit of this vulnerability has been implemented and will be provided together with this report.

This vulnerability can be prevented by a cross-contract lock that prevents entering `VestingNFTReceiver::withdraw` while `Distributor::withdraw` is active.
A lighter (but less robust) solution is to add the following check:

```
require(withdrawAmount >= amount)
```

One should also keep in mind a symmetric but harder to exploit vulnerability: if the **victim** calls `Distributor::withdraw`, and in his receive hook triggers some untrusted code (e.g., transfers the received funds), the adversary can do a nested `Distributor::withdraw`, causing the distributor to register a larger withdrawn amount for the victim that the real one (hence increasing the adversary's share). A nonReentrant guard in `Distributor::withdraw` prevents this.

The general recommendation at the end of C2 also applies here.

C2	**The adversary can transferOwnership on vestingNFTReceiver**	**Resolved**

Via `Distributor::call`, an adversary can call `VestingNFTReceiver::transferOwnership` and change the ownership to himself, which then allows him to call `VestingNFTReceiver::withdraw` directly (not via the distributor) and receive all vesting funds.

This can be solved by removing the `transferOwnership` method and baking the owner into the `VestingNFTReceiver` during initialization.

As a general recommendation, having a general-purpose Distributor contract which allows arbitrary interactions with `VestingNFTReceiver` via `Distributor::call`, makes it much harder to design a safe interface. We recommend using a distributor contract with exactly the needed functionality, possibly even merged with `VestingNFTReceiver`. This would easily solve C2, and would also make it easy to add a lock that solves C1.

High Severity

Id	Description	Status

H1	Merkle trees can be extended by the adversary	Resolved

A Merkle Tree can be easily **"extended"** by an adversary **only knowing its root**, to form a new tree having the original one as a subtree, together with **any other data** chosen by the adversary.

This could hypothetically be exploited as follows: when `reportTree` is submitted, the adversary extends it into an `advReportTree`, containing the original tree and an extra "reporter=adversary" leaf, and submits it (possibly front-running the victim to get an earlier timestamp). When the victim discloses the report leaf with a proof of inclusion in `reportTree`, the adversary can extend this proof into a proof that the same report leaf belongs to `advReportTree`. A proof of the "reporter=adversary" leaf can be also provided, so the adversary could claim the bounty for himself.

Such an attack is likely to be detected during the offchain triaging/evaluation procedure. However, relying on such a detection is a weak defense strategy. Moreover, the project owner has strong incentives to participate in this attack (to claim his own bounty), so his involvement could increase the probability of success.

Such an `advReportTree` will contain duplicate reporter leaves, and a method to invalidate reports with duplicate keys is envisioned in the README. However, apart from not being implemented, this is again a weak defense strategy, putting the burden of defense on the victim.

A good practice to avoid such attacks is to "sign" the report leaf by including the address of the reporter in the leaf (or a hash of the reporter, if we wish to disclose the report before the reporter). The reporter leaf could still be available if we wish to be able to disclose it alone (in this case a check can be made when everything is disclosed, that the two reporter values match).

Another design solution could be to give a **fixed structure** to the MT. Currently, there can be any number of leaves in any order, and a proof that a leaf is present **anywhere** in the tree is accepted (as a consequence key labels are needed to distinguish the leaves). This could be replaced by a design where the tree necessarily has 4 leaves in a specific order (leaf 0 is the reporter, leaf 1 the report, etc), and a proof is provided that the claimed leaf is present **at its expected position**. This would have the following advantages:

- extending a tree into a larger one is not possible
- there is no way to have duplicate keys (since there is only one "leaf 0")
- there is no need to annotate the leaves with key labels.

On the other hand, extending this design in the future is harder, so one should think of ways to future-proof it.

See also later advisory item concerning Merkle Trees.

Medium Severity

Id	Description	Status
M1	**Error in UnstakingDelay.getWeight**	**Resolved**

The following computation is incorrect:

```
nextInput.presentTime = input.startTime - delay;
if (unstakingStart > 0 && input.presentTime >= unstakingStart + delay) {
  nextInput.presentTime += input.presentTime - unstakingStart;
}
```

The time `input.startTime - delay` corresponds to the earliest possible starting time in case **UNSTAKE hasn't been triggered** yet, so the condition should be

```
if (unstakingStart != 0) {
```

Low Severity

Id	Description	Status
L1	**External calls made unnecessarily**	**Open**

In the contract BugReportNotary.sol functions `disclose()` and `discloseCommentary()` are declared `external`. However, they are both called from within the contract itself as `this.disclose()` and `this.discloseCommentary()`, resulting in increased gas costs, the beginning of an internal transaction, and confusion by the change of `msg.sender`.

Since an obligation for external calls to these functions has no special reason, we suggest that they be declared `public` for clarity and gas efficiency, and the "`this.`" calling pattern be removed.

Other/Advisory Issues

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

Id	Description	Status
A1	**Possible logic error in `SumVesting` combinator schedule**	**Dismissed (intended behavior, assumptions on vesting schedules will be clearly stated)**

In combinator schedule `SumVesting.sol` it is implicitly assumed that the result of the sub-controllers for both `getVested()` and `getWeight()` is linearly dependant on the input amount

```
function getVested(CommonParameters calldata input) external pure override
returns (uint256 result) {
    [...]
    for (uint256 i; i < subControllers.length; i++) {
      IVestingController subController = subControllers[i];
      uint256 share = subShares[i];
      // Dedaub: should be input.amount * share/totalShares
      // Dedaub: but the division happens in the end
      nextInput.amount = share * input.amount;
      totalShares += share;
      [...]
      result += subController.getVested(nextInput);
    }
    result /= totalShares;
}
```

Thus the whole input amount is passed to all sub-controllers only to divide the accumulated result amount to the `totalShares` at the very end.
While this assumption holds in the case of simple schedules, such as CliffVesting and LinearVesting, it may not hold for more complex ones that may be added in the future.

Similarly, an inaccurate input amount is passed to the sub-controllers in functions `getContext()`, `createInitialState()` and `triggerEvent()`.

A2	Incorrectly testing that a transaction succeeded	Resolved

The following test is taken from `test/commentary_tests.js` :

```
await expect(Notary.connect(Operator).submitCommentary(BYTES32_STRING));
await
expect(Notary.connect(Operator).submitCommentary(BYTES32_ZERO)).to.be.reverted;
```

It seems that the intention of the first line is to test that `submitCommentary` succeeded without reverting. However this line does not really check anything, the test will pass even if `submitCommentary` reverts.
The correct test would be:

```
await
expect(Notary.connect(Operator).submitCommentary(BYTES32_STRING)).not.to.be.rever
ted;
```

Many similar cases exist in `commentary_tests.js`, `contract_tests.js` and `test/distributor_tests.js` (and possibly elsewhere).

In the following case, adding the `.not.to.be.reverted` revealed logic errors in the test:

```
it("Validating the attestation on disclosed report `AFTER` ATTESTATION_DELAY",
async function () {
    await Notary.connect(Triager).attest(reportRoot, kk, commit)
    await expect(Notary.connect(Triager).disclose(reportRoot, key, salt, value,
merkleProofval))

    const increaseTime = ATTESTATION_DELAY * 60 * 60 // ATTESTION in `hour`
format x 60 min x 60 sec
    await ethers.provider.send("evm_increaseTime", [increaseTime]) //  1.
increase block time
    await ethers.provider.send("evm_mine") // 2. then mine the block
...
```

Here, `disclose` is executed **before** the `ATTESTATION_DELAY` so it should fail, although the test makes it look like it should succeed. The reason why the test passes is that:

1. The `await expect(...)` line performs no checks

2. Moreover this line does not wait for the transaction to finish, so although `disclose` is launched before moving time forward, it is executed in the future block, after the time delay, and as a consequence it succeeds.

So, if `.not.to.be.reverted` is added to the `await expect(...)` line, the test will fail, unless the line is moved after the time increase.

A3	Immutable variables	Resolved

There are some variables in contracts Distributor.sol and TokenMinter.sol that are assigned during contract construction and could never change thereafter.

In Distributor.sol:

```
/// Only settable by the initializer.
bool public override callEnabled;
address public override nftHolder;
uint256 public override maxBeneficiaries
```

In TokenMinter.sol:

```
/// This initialized by the deployer. The token is completely trusted.
IImmunefiToken public override token;
```

We suggest these variables be declared immutable for clarity and gas efficiency.

A4	Unnecessary receive hook	Dismissed (hook needed by IVestingNFTFunder.vestingNFTCallback)

The `receive()` hook in `VestingNFT` is not to be used intentionally, since ETH is received via `mint()`. It would be better to `revert` to avoid accidentally receiving ETH.

A5	The need to construct a Merkle Tree can be easily avoided	Open

A large amount of code (MerkleTree.sol / QuickSort.sol) is aimed at constructing (rather than verifying) a MT. However, this is only used by `BugReportNotary.assignNullCommentary` to construct a tree for a trivial empty commentary.

This can be easily avoided by having a hard-coded constant value `NULL_COMMENTARY` that denotes an empty commentary. The call to `discloseCommentary` can be omitted in this case

(or `discloseCommentary` can simply check that the `value` is empty) and `NULL_COMMENTARY` can be immediately set as canonical.

A6	Dead code	Partially resolved (dead code still present in LinearVesting.sol)

In vesting schedule CliffVesting.sol function `_decodeParams()` is supposed to return a uint256 value

```
function _decodeParams(bytes calldata params) internal pure returns (uint256
cliffTime) {
    cliffTime = abi.decode(params, (uint256));
}
```

However, this schedule requires an empty parameter list

```
function checkParams(CommonParameters calldata input) external pure override {
    require(input.params.length == 0);
}
```

All three internal functions `_decodeParams()`, `_decodeState()` and `decodeContext()` are never called for CliffVesting, while the later two are also never called for LinearVesting schedules.

We suggest that all unused functions be removed for clarity and gas savings. Alternatively, the current body of `CliffVesting::_decodeParams` should be removed.

A7	Unused function argument	Resolved (argument is not redundant for code extensibility reasons)

In `KeeperRewards::keeperRewards` the first argument is redundant

```
function keeperRewards(address, uint256 value) external pure override returns
(uint256) {
    return value / 1000;
}
```

We suggest it be removed for clarity.

Also, the constant 1000 in the same code is an arbitrary “magic constant”, best given a name to document intent.

A8	Inconsistent calling pattern	Resolved

In BugReportNotary the MerkleProof::verify function is called with different syntax. Once as:

```
merkleProof.verify(reportRoot, leafHash)
```

and once as:

```
MerkleProof.verify(merkleProof, commentaryRoot, leafHash)
```

We recommend making uniform for consistency.

A9	ReentrancyGuard in vestingNFT	Info

The README asks for possible ways to remove ReentrancyGuard from vestingNFT. We believe that these guards are critical and advise against trying to remove them (we see no safe way to do so, while keeping the dynamic way of computing the amount of transferred tokens).

In particular, a reentrancy to `mint` from `withdraw` will directly lead to a severe loss of funds. Currently this is indirectly protected by the `nonReentrant` flag in `_deposit` and `_beforeTokenTransferInner` (we recommend clearly documenting the importance of these flags, to prevent them from getting accidentally removed).

A10	Storing funds in a single contract (VestingNFT)	Info

The architecture stores all ERC-20 tokens (assets) in a single contract (VestingNFT), and accounting for how they are shared among many different NFTs/bounties. This is a decision that puts a significant burden on asset accounting. It should be simpler/safer to have a treasury contract that indexes assets by NFT and keeps assets entirely separate. However, the current design seems to exist in order to support ERC-20 tokens that change in number, with time. This certainly necessitates a “shares” model instead of a “separate accounts” model. It may be good to document exactly the behavior of tokens that the designer of the contract expects, with specific token examples. There are certainly token models that will not be supported by the current design, and others that are.
A more radical approach could also be to use a clone of VestingNFT for each bounty (similarly to how clones of vestingNTFReceiver are used), so that funds for each bounty are kept in a separate contract. Apart from facilitating the accounting (no need for a "shares" model), this design would likely mitigate the losses from a critical bug (the adversary could drain a single bounty but not all of them).

A11	Simplification/dead code	Resolved

The function `QuickSort::sort` admits some simplifications/dead-code elimination. Some of these are only possible under the invariant `left < right` (which is true in the current uses of the function), others regardless. We highlight them in the four code comments below.

```
function sort(
  bytes32[] memory arr,
  uint256 left,
  uint256 right  // Dedaub: invariant: left < right
) internal pure {
  uint256 i = left;
  uint256 j = right;
  if (i == j) return;  // Dedaub: dead code, under invariant
  bytes32 pivot = arr[left + (right - left) / 2];
  while (i <= j) {  // Dedaub: definitely true the first time, under invariant,
                    // loop could be a do..while
    while (arr[i] < pivot) i++;
    while (pivot < arr[j]) j--;
    if (i <= j) {  // Dedaub: always the case, no need to check
      (arr[i], arr[j]) = (arr[j], arr[i]);
      i++;
      j--;
    }
  }
  if (left < j) sort(arr, left, j);
  if (i < right) sort(arr, i, right);
}
```

A12	ImplOwnable cannot recover from renouncing	Dismissed (intended behavior)

In the ImplOwnable contract (currently unused) if the owner calls `renounceOwnership`, no new owner can be installed. It is unclear whether this is intentional and whether the contract will be used in the future.

A13	Suggestion for contract size	Info

For the bytecode size issues of VestingNFT, our suggestion would be to create a VestingNFT library contract (containing all functions that do not heavily involve storage slots, such as pure functions, some views that only affect 1-2 storage slots) and have calls in VestingNFT delegate to the library versions. Shorter-term solutions might exist (e.g., removing one of the super-contracts, such as DelegateGuard, in some way) but they will not save a large contract from bumping against size limits for long.		
A14	**Floating `pragma`**	**Open**
Use of a floating pragma: The floating pragma `pragma solidity ^0.8.6;` is used, allowing contracts to be compiled with any version of the Solidity compiler that is greater or equal to v0.8.6 and lower than v.0.9.0. Although the differences between these versions should be small, for deployment, floating pragmas should ideally be avoided and the pragma be fixed.		
A15	**Compiler known issues**	**Info**
Solidity compiler v0.8.6, at the time of writing, has no 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 status of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

About Dedaub

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