WitnessChain

Smart Contract Security Assessment

Mar 23, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the WitnessChain smart contracts. Notably, the smart contracts themselves are only a part of the overall protocol functionality, and many of the desirable properties of the system depend on off-chain parties.

BACKGROUND

WitnessChain aims to introduce an incentivized “watchtower” network for rollups. The watchtowers submit “proofs of diligence” and “proofs of inclusion” that establish that they are dutifylly watching the rollup commitments on the mainnet, and they get appropriately rewarded (or, in the future, penalized). The WitnessChain functionality aims to be an EigenLayer AVS (actively validated service), with full integration with the EigenLayer infrastructure, including operators, delegation, and stakes.

Off-chain functionality plays a major role in the WitnessChain protocol. First, the protocol itself maintains its own L2 chain (independent of the L2 rollup chains that are being actively watched). The audited smart contracts are partly deployed on the Ethereum mainnet (L1) and partly on that private L2 chain. Both chains are mostly used as unforgeable records of claims. Therefore, the audited smart contracts do not directly handle funds, nor check the validity of proofs. An off-chain party is responsible for reading proofs out of events emitted and checking them.

SETTING & CAVEATS

The audit report is over the contracts of the at-the-time private repository https://github.com/kaleidoscope-blockchain/eigenlayer-avs-watchtower, branch `development`, at commit `97fdbe39f82a9f061e67b6b805133e4da0b1e823`. **2** auditors worked on the codebase for **4** days on the following contracts:

```
src/
├── core/
│   ├── AlertManager.sol
│   ├── DiligenceProofManager.sol
│   ├── L2ChainMapping.sol
│   ├── OperatorRegistry.sol
│   └── WitnessHub.sol
├── interfaces/
│   ├── IAlertManager.sol
│   ├── IDiligenceProofManager.sol
│   ├── IL2ChainMapping.sol
│   ├── IOperatorRegistry.sol
└── └── IWitnessHub.sol
```

Fixes were audited at commit `413f12748ffea74610e59f9b201b3f39cef28ef1`. The review of fixes was local, on the code sites affected and not exhaustive over the entire codebase.

The most important caveat, providing context for the rest of the report, is that most of the properties of the protocol are not ensured by the audited smart contracts, but by off-chain code. This includes the management of funds, but also the claim validation (including proof validation and validity checking of other parameters), and any signature checking. Therefore, from a smart contract security standpoint, we assume that the off-chain code behaves perfectly, validating correctly all claims and only providing correct parameters.

The same observation holds regarding the EigenLayer integration. Any consideration of stakes is left to off-chain code. The on-chain smart contracts merely check that an entity is a current EigenLayer operator.

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:

ID	Description	STATUS
M1	Off-chain signatures are accepted but the check is a no-op, at least for the smart contract	**RESOLVED/ ACKNOWLEDGED**

Some functions accept off-chain signatures but the check performed on them is a no-op. If off-chain code performs the right validation, there are no security implications. Generally, this is probably not a threat but we are elevating it to MEDIUM only to point out that this check does not ensure what the developers may think it does.

WitnessHub:67

```
function raiseAlert(
       uint256 chainID,
       uint256 l2BlockNumber,
       bytes calldata originalOutputRoot,
       bytes calldata computedOutputRoot,
       bytes calldata proofOfDiligence,
       bytes calldata signature
```

```
)
        external
        whenNotPaused
{
        // get the operator for watchtower and check if it is active
        address operator = registry.getOperator(msg.sender);

        require(registry.isActiveOperator(operator),
                "Signer's operator is not an active operator");

        // 1. get the messageHash
        bytes32 messageHash = keccak256(abi.encodePacked(proofOfDiligence));

        // 2. get the ethSignedMessageHash
        bytes32 ethSignedMessageHash = messageHash.toEthSignedMessageHash();

        // 3. recover signer from ethSignedMessageHash and signature
        address signer = ethSignedMessageHash.recover(signature);

        require(signer == msg.sender, "Signer is not the txn originator");
```

There is no point in checking that the signer of an ECDSA signature is the `msg.sender` (unless this is one possible condition of many). The `msg.sender` has already used his/her private key to sign the transaction, i.e., has signed all the inputs. There is no point in checking that they have separately, off-chain, also signed a subset of the inputs.

Also in `DiligenceProofManager::validateSigner:171`

```
        // 1. get the messageHash
        bytes32 messageHash = keccak256(abi.encodePacked(_proof));

        // 2. get the ethSignedMessageHash
        bytes32 ethSignedMessageHash = messageHash.toEthSignedMessageHash();

        // 3. recover signer from ethSignedMessageHash and signature
        address signer = ethSignedMessageHash.recover(_signatureProof);
```

```
        require(signer == msg.sender, "Signer is not the txn originator");
```

M2	Any operator can claim a watchtower as affiliated with them, resulting in possible attacks	**RESOLVED**

The operator associated with a watchtower (in OperatorRegistry) is fundamental in further actions performed by the watchtower. For instance, when a watchtower submits a proof (in `DiligenceProofManager::submitProof`) or raises an alert (in `AlertManager::raiseAlert`) the watchtower's operator is checked to be active. Even more, when rewards are accounted, by the off-chain aggregator, this is done on the basis of operators:

`WitnessHub:190`

```
function updateReward(
        uint256 _chainID,
        uint256 _blockNumBegin,
        uint256 _blockNumEnd,
        address[] calldata _operatorsList,
        Types.BountyRewards[] calldata _proofRewards,
        bytes32 _rewardHash
) external whenNotPaused onlyAggregator
```

However, there is no restriction in which operator can register which watchtower, in `OperatorRegistry::register`. This call is initiated by the operator, but the watchtower address does not sign any part of the submitted data.

This raises the possibility for several attacks. (The exact functioning of each attack depends again on the checks performed by the off-chain aggregator, but it seems very hard to ascertain which operator is valid.)

1. Front-running a valid `register(operator1, watchtower1)` with a `register(maliciousOperator, watchtower1)`. This could be done based on

the hope that the valid `operator1` will not detect that their `register` transaction has reverted and the watchtower will subsequently submit proofs. In this way, the rewards can be sent to the wrong operator.

2. A malicious operator can register any number of valid watchtower addresses with no restriction. The addresses can then not be used with a different (valid) operator, resulting in denial of service.

Resolution:

While the registration flow now requires a signed message from the watchtower being registered, the structure of the signed message does not ensure that the signature is generated solely for the watchtower's registration process within the protocol:

`OperatorRegistry:94`

```
function calculateWatchtowerRegistrationMessageHash(address operator, uint256
expiry)
        public
        pure
        returns (bytes32)
{
        bytes32 structHash = keccak256(abi.encode(operator, expiry));
        bytes32 ethSignedMessageHash = structHash.toEthSignedMessageHash();
        return ethSignedMessageHash;
}
```

The fact that the signed message does not contain a separator, unique to the watchtower registration, raises concerns about the ability of an operator to use signatures that might have been constructed for a different purpose. The protocol team has acknowledged this issue, but it is not believed to pose a practical threat.

LOW SEVERITY:

ID	Description	STATUS
L1	Extraneous SLOADs, in code that otherwise emphasizes gas efficiency	**LARGELY RESOLVED**

There is a repeated code pattern that results in higher gas usage, due to extraneous SLOAD instructions. The pattern is shown in the following example, and repeated in more instances, listed later:

WitnessHub:101

```
function setRegistry(IOperatorRegistry _registry) external onlyOwner {
        address previousregistry = address(registry);
        registry = _registry;
        emit RegistryUpdated(previousregistry, address(registry));
}
```

In this code, although there is a local variable that holds the same value, the `emit` statement instead uses the storage variable, incurring high gas expense.

The same simple pattern also occurs elsewhere:

WitnessHub:109

```
function setL2ChainMapping(IL2ChainMapping _l2chainmapping) external onlyOwner {
        address previousl2ChainMapping = address(l2ChainMapping);
        l2ChainMapping = _l2chainmapping;
        emit L2ChainMappingUpdated(previousl2ChainMapping, address(l2ChainMapping));
}
```

WitnessHub:117

```
function setAggregatorAddress(address _aggregator) external onlyOwner {
        address previousaggregator = aggregator;
        aggregator = _aggregator;
        emit AggregatorUpdated(previousaggregator, aggregator);
```

```
}
```

More complex instances of the same problem occur elsewhere. To resolve those, one needs to introduce new temporary (memory, not storage!) records that copy the storage values for the duration of the transaction.

OperatorRegistry::register:112

```
        // Store the Operator details
        Operator storage newOperator = operators[operatorAddress];

        newOperator.operatorAddress = operatorAddress;
        newOperator.isRegistered = true;
        newOperator.watchStatus = WatchStatus.INACTIVE;
        newOperator.operatorType = OperatorType.EIGENLAYER;
        newOperator.isActive = true;
        …
        emit OperatorRegistered(newOperator);
```

OperatorRegistry::deRegister:132

```
        Operator storage currOperator = operators[operatorAddress];

        require(currOperator.isRegistered == true, "Address is not registered");

        currOperator.isRegistered = false;

        emit OperatorDeRegistered(currOperator);
```

OperatorRegistry::suspend:143

```
        Operator storage currOperator = operators[operatorAddress];

        require(
                currOperator.isRegistered == true && currOperator.isActive == true,
                "Cannot suspend if operator not registered + active"
        );
```

```
        currOperator.isActive = false;

        emit OperatorSuspended(currOperator);
```

L2	Unnecessary inner transaction	**RESOLVED**

The code pattern below starts a separate (inner) transaction, for no reason. If this function is never called on-chain, the issue is mostly ignorable, but if the contracts evolve to ever call it, the code is unnecessarily wasteful.

`OperatorRegistry::isValidWatchtower:181`

```
    return this.isActiveOperator(operator);  // Dedaub: why “this.”?
```

L3	An operator can prevent his suspension	**RESOLVED**

Since `OperatorRegistry::suspend` requires for the operator to be registered at the time of the suspension:

`OperatorRegistry:146`

```
function suspend(address operatorAddress) external onlyOwner {
        Operator storage currOperator = operators[operatorAddress];

        require(
                currOperator.isRegistered == true && currOperator.isActive == true,
                "Cannot suspend if operator not registered + active"
        );
```

an operator may prevent a future suspension by calling `OperatorRegistry::deRegister`:

`OperatorRegistry:129`

```
function deRegister(address operatorAddress) external {
        require(operatorAddress == msg.sender, "Sender != Operator Address");
        Operator storage currOperator = operators[operatorAddress];
        require(currOperator.isRegistered == true, "Address is not registered");
        currOperator.isRegistered = false;
        emit OperatorDeRegistered(currOperator);
}
```

Additionally, as far as the system is concerned, the operator will continue to be considered active. This means that operations like `WitnessHub::updateReward` and `DiligenceProofManager::submitProof` continue to work the deregistered operator:

`WitnessHub::updateReward:207`

```
if (registry.isActiveOperator(_operatorsList[i])) {
```

`WitnessHub::submitProof:208`

```
require(registry.isActiveOperator(operator), "Signer's operator is not an activeoperator");
```

The project should consider making the operator appear as inactive upon de-registration.

CENTRALIZATION ISSUES:

ID	Description	STATUS
N1	Permissioned accounts and centralized services	**INFO**
As mentioned in the protocol caveats section, much of the responsibility for the correct functioning of the protocol lies with an off-chain aggregator account. The aggregator is mainly responsible for correct reward distribution and claim validation. Additionally,		

the owner of key contracts has significant power, including the ability to pause contracts, change the aggregator, and more. However, since the contracts do not directly manage funds, this off-chain power is entirely expected, and since it is restricted to the correct incentivization of behavior, it is reasonably limited.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them. These issues are explicitly labeled "Info" and not "Open". Before acting on such issues, developers should also confirm them to the best of their ability.

ID	Description	STATUS
A1	Some validation is left to off-chain parties, although it could be done in smart contracts	**INFO**

Ultimate correctness of the protocol rests with the off-chain aggregator and other off-chain parties. We note here *some* of the checks that could have been performed on-chain, also to help examine for possible omissions in off-chain checks.

- In `DiligenceProofManager::submitProof` the parameter `_l2BlockNumber` is not checked to be greater than the latest claimed block number, merely different. The parameter could, in theory, list an arbitrarily old (or future) block number, although the off-chain claim checking will probably reject this.
  `DiligenceProofManager::submitProof:212`

```
    require(minerLatestClaimedBlock[msg.sender][_chainID] != _l2BlockNumber,
            "Miner has already submitted claim");
```

- The input from the aggregator is fully trusted, with little sanity checking on chain. For instance, there is no check in the function below that `_blockNumBegin <= _blockNumEnd`

WitnessHub:190

```
function updateReward(
       uint256 _chainID,
       uint256 _blockNumBegin,
       uint256 _blockNumEnd,
       address[] calldata _operatorsList,
       Types.BountyRewards[] calldata _proofRewards,
       bytes32 _rewardHash
) external whenNotPaused onlyAggregator
```

- The `chainID` parameter in `AlertManager::raiseAlert` is not checked to belong in the list of supported chains

WitnessHub:190

```
alert.chainID = chainID;
```

A2	Implicit correctness assumptions regarding off-chain accounting	**INFO**

We note some instances inside the codebase in which non-trivial off-chain accounting is required:

- Regardless of whether a submitted proof gets validated, the bounty for a particular chain and block number is always set. Since the `claimBounties` field cannot be unset on-chain, the off-chain components have to account for the claimed blocks that are based on invalid proofs:

DiligenceProofManager::submitProof:218

```
if (proofType == 0) {
              bounty = podBounties[_chainID][_l2BlockNumber];
              bounty.claimBounties = currentPODBountiesForRollup[_chainID];
              …
} else if (proofType == 1) {
              bounty = poiBounties[_chainID][_l2BlockNumber];
              bounty.claimBounties = currentPOIBountiesForRollup[_chainID];
              …
```

```
}
```

- Since the `lastUpdateBlock` field of an entry inside the `WitnessHub::operatorRewards` mapping is not initialized, the function `WitnessHub::getNextBlockByChainID` will return 0 the first time it is invoked inside `WitnessHub::updateReward`. This has the effect of not being able to call `WitnessHub::updateReward` with the latest L2 block number once the Witness Chain protocol is launched; the aggregator will instead have to submit offsetted L2 block numbers:

`WitnessHub:203`

```
function updateReward(
       uint256 _chainID,
       uint256 _blockNumBegin,
       uint256 _blockNumEnd,
       address[] calldata _operatorsList,
       Types.BountyRewards[] calldata _proofRewards,
       bytes32 _rewardHash
) external whenNotPaused onlyAggregator {
       …
       require(_blockNumBegin == getNextBlockByChainID(_chainID), "Incorrect
_blockNumBegin");
```

`WitnessHub:172`

```
function getNextBlockByChainID(uint256 _chainID) public view returns (uint256) {
       if (operatorRewards[_chainID].lastUpdateBlock == 0) {
              return 0;
       }
       return operatorRewards[_chainID].lastUpdateBlock + 1;
}
```

A3	Unnecessary argument	**ACKNOWLEDGED, needed for compliance with interface**

Both `OperatorRegistry::register` and `OperatorRegistry::deRegister` accept an `operatorAddress` argument which is enforced to be equal to the `msg.sender` of the call. Both functions could be re-written so that every occurrence of the `operatorAddress` argument is replaced with the `msg.sender` variable.

`OperatorRegistry:94`

```
function register(address operatorAddress, address watchtowerAddress) external {
        require(operatorAddress == msg.sender, "Sender != Operator Address");
```

On a similar note, `WitnessHub::registerOperatorToAVS` could have all occurrences of the `operator` argument be replaced with the `msg.sender` variable:

`WitnessHub:137`

```
function registerOperatorToAVS(
        address operator,
        ISignatureUtils.SignatureWithSaltAndExpiry memory operatorSignature
) external whenNotPaused {
        require(msg.sender == operator, "WitnessChain AVS: Operator should be the
sender");
```

A4	`OperatorRegistry::isValidWatchtower` does not accept `address(0)` as a watchtower address, but, earlier, `register` does	**RESOLVED**

`OperatorRegistry::register` allows for the registration of `address(0)` as `watchtowerAddress`. However, reverts when called with address(0):

`OperatorRegistry:176`

```
function isValidWatchtower(address watchtower) external view returns (bool) {
        require(watchtower != address(0), "watchtower address cannot be 0");
        ...
}
```

Even though this does not affect the protocol in any practical terms, `OperatorRegistry::isValidWatchtower` will become unusable for operators that might mistakenly register `address(0)` as a watchtower. The check should probably have come earlier, at registration time.

A5	Unused argument	**ACKNOWLEDGED, needed for compliance with interface**

The `operator` argument of `WitnessHub::getOperatorRestakedStrategies` is never used:

`WitnessHub:261`

```
function getOperatorRestakedStrategies(address operator) external view
returns (address[] memory) {
      return _getRestakeableStrategies();
}
```

A6	Boolean expressions are inelegant	**RESOLVED**

Boolean expressions are often checked with the pattern `boolExpr == false` or `boolExpr == true`, instead of the more idiomatic `!boolExpr` or `boolExpr`.

`OperatorRegistry:101`

```
require(whitelisted[operatorAddress] == true …
```

`OperatorRegistry:107`

```
if (checkIsDelegatedOperator == true) ...
```

`OperatorRegistry:109`

```
require(isDelegatedOperator == true …
```

OperatorRegistry:134

```
require(currOperator.isRegistered == true ...
```

OperatorRegistry:145

```
require(
      currOperator.isRegistered == true && currOperator.isActive == true,
```

OperatorRegistry:157

```
require(checkIsDelegatedOperator == false ...
```

OperatorRegistry:164

```
require(checkIsDelegatedOperator == true
```

A7	It is unclear how `WitnessHub::_proofCommitments` is intended to be accessed	**RESOLVED**

The `WitnessHub::_proofCommitments` internal variable is updated with the aggregator's proof commitments and not used in any other way:

WitnessHub::updateReward:226

```
_proofCommitments.push(
            Types.AggProofCommitment({
                chainID: _chainID,
                l2BlockNumberBegin: _blockNumBegin,
                l2BlockNumberEnd: _blockNumEnd,
                rewardHash: _rewardHash,
                submissionBlock: block.number
            })
```

```
);
```

It seems that the intent is for the variable to be accessed off-chain, and while it is entirely possible to access the corresponding storage slots in a calculated manner, it seems more reasonable to declare `WitnessHub::_proofCommitments` as a public variable.

A8	The effects of using `delete` on a storage array should be noted	**INFO**

Using the `delete` keyword on a storage array sets the array's length to 0, but does not reset the array elements' value.

`WitnessHub::setStrategyParams:240`

```
delete strategyParams;
```

Solidity code that attempts to access the deleted array will cause a revert. However, since it is possible to access the storage slots of the deleted array via inline assembly, this note is worth considering in upcoming iterations of the code.

A9	Very minor optimization	**RESOLVED**

We generally refrain from recommending the use of `unchecked`, as well as of `++i` instead of `i++` for the purpose of (extremely minor) gas savings. However, we note that the code currently uses these optimization patterns in most loops, but keeps `i++` in two instances:

(The second is merely a view function, currently not called on-chain, but the first is an on-chain function.)

`WitnessHub::setStrategyParams:242`

```
      for (uint256 i = 0; i < params.length;) {
```

```
            require(address(params[i].strategy) != address(0), …);

            strategyParams.push(params[i]);

            unchecked {
                  i++;
            }
      }
```

WitnessHub::_getRestakeableStrategies:267

```
      for (uint256 i = 0; i < strategyParams.length;) {
            strategies[i] = address(strategyParams[i].strategy);
            unchecked {
                  i++;
            }
      }
```

A10	Storage variables not listed contiguously in source file	**INFO**

Storage variables in `OperatorRegistry` are listed interspersed with events. This is unidiomatic.

OperatorRegistry:19

```
      // operator => operator attributes
      mapping(address => Operator) public operators;

      // operator => if operator has been whitelisted (for registration)
      mapping(address => bool) private whitelisted;

      // list of registered watchtower addresses
      address[] public watchTowerAddressesList;

      // watchtower address => operator address
      mapping(address => address) watchtowerToOperator;
```

```
    /// @notice Emitted once an operator has successfully registered
    event OperatorRegistered(Operator operator);

    /// @notice Emitted once an operator has deregistered
    event OperatorDeRegistered(Operator operator);

    /// @notice Emitted once an operator has been suspended
    event OperatorSuspended(Operator operator);

    address public delegationManagerAddress;

    address public slasherAddress;

    // Flag to check if operator has registered for delegation with EigenLayer
    bool checkIsDelegatedOperator;
```

A11	Typos in comments	**RESOLVED**

There are small typos in some comments, including:

`WitnessHub:22`

```
*           aggreagtor to submit
```

`WitnessHub:49`

```
    /// @notice The EL startergy param
```

`WitnessHub:186`

```
    /// @param _operatorsList   The list of operators to updare rewards for
```

A12	Compiler bugs	**INFO**

The code has the compile pragmas `^0.8.15`. For deployment, we recommend no

floating pragmas, i.e., a specific version, so as to be confident about the baseline guarantees offered by the compiler. Version `0.8.15`, in particular, has some known bugs, which we do not believe to affect the correctness of the contracts.

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.