Immunefi Arbitration Protocol

Smart Contract Security Assessment

Feb 07, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the Immunefi Arbitration protocol. There were some issues found of Medium severity but with reduced likelihood. They mostly concern the breaking of some of the assumptions made by the protocol. Furthermore, a few more inconsistencies were found of Lower severity. The codebase was accompanied by sufficient documentation and a test suite with 75% of coverage.

BACKGROUND

The Immunefi Arbitration protocol allows projects providing bug bounties on the Immunefi platform to provide a "Proof of Funds" to whitehats in Immunefi's audit contests.

The protocol provides the concept of a `Vault` which is owned by the project, and which holds the funds corresponding to the bug bounty being offered. Each such vault is a `Gnosis Safe` smart contract wallet. While projects can withdraw funds from the vault, such a withdrawal has to pass through a `Timelock` contract in this protocol.

The Immunefi protocol also provides an `Arbitration` system which allows both whitehats and projects to request an arbitration for the issue of a bug bounty. The protocol will assign the `ARBITER_ROLE` to a highly trusted 3rd-party address (as statedin the documentation), and this arbiter will then be empowered to carry out an arbitration and decide whether to close the arbitration or reward the whitehat with funds held inside the project's vault. Any actor requesting an arbitration needs to pay a fee to Immunefi in the order of $10K.

A project may also decide to issue a reward of its own accord from its vault, without an arbitration. However, if the project's vault is under arbitration, issuing rewards can only take place by passing through a `RewardTimelock` contract. This mechanism is intended

to restrict funds from being disbursed when a whitehat has already made a claim on those funds and has enabled an arbitration.

On the other hand, contracts with the `ENFORCER_ROLE`, such as the `Arbitration` contract, do not have this restriction and can directly disburse funds from the vault and complete an arbitration.

The protocol is also able to assign the `VAULT_FREEZER` role to an address. Such addresses can freeze and unfreeze vaults. Projects with frozen vaults cannot move the funds contained in such vaults until the vaults are unfrozen. Only the `ENFORCER_ROLE` can bypass the freezing of a vault and move funds as described above.

The protocol also has an `EmergencySystem` which can be used to pause all movement of funds associated to the protocol.

SETTING & CAVEATS

This audit report mainly covers the contracts of the at-the-time private repository Immunefi - Vaults System at commit `30540a009ecdd98b9a854a8796897d62a3a1ecfe`.

As part of the audit, we also reviewed the fixes for the issues included in the report. The fixes were delivered as part of PR #50 and we attest that they have been implemented correctly.

Two auditors worked on the codebase for 10 days on the following contracts:

```
src/
├── Arbitration.sol
├── EmergencySystem.sol
├── ImmunefiModule.sol
├── RewardSystem.sol
├── RewardTimelock.sol
```

```
├── Timelock.sol
├── VaultFreezer.sol
├── WithdrawalSystem.sol
│
├── base/
│   ├── AccessControlBaseModule.sol
│   ├── AccessControlGuardable.sol
│   ├── ArbitrationBase.sol
│   ├── RewardSystemBase.sol
│   ├── RewardTimelockBase.sol
│   ├── TimelockBase.sol
│   └── WithdrawalSystemBase.sol
│
├── common/
│   ├── Rewards.sol
│   ├── VaultDelegate.sol
│   └── VaultFees.sol
│
├── encoders/
│   ├── ArbitrationOperationEncoder.sol
│   ├── BaseEncoder.sol
│   ├── RewardTimelockOperationEncoder.sol
│   └── TimelockOperationEncoder.sol
│
├── events/
│   ├── IArbitrationEvents.sol
│   ├── IEmergencySystemEvents.sol
│   ├── IImmunefiGuardEvents.sol
│   ├── IRewardSystemEvents.sol
│   ├── IRewardTimelockEvents.sol
│   ├── IScopeGuardEvents.sol
│   ├── ITimelockEvents.sol
│   ├── IVaultFreezerEvents.sol
│   └── IWithdrawalSystemEvents.sol
│
├── guards/
│   ├── ImmunefiGuard.sol
│   └── ScopeGuard.sol
│
├── handlers/
```

```
│   └── VaultSetup.sol
│
├── oracles/
│   ├── IPriceConsumerEvents.sol
│   ├── IPriceFeed.sol
│   ├── PriceConsumer.sol
│   │
│   └── chainlink/
│       ├── Denominations.sol
│       ├── FeedRegistryInterface.sol
│       │
│       └── shared/
│           └── interfaces/
│               ├── AggregatorInterface.sol
│               ├── AggregatorV2V3Interface.sol
│               └── AggregatorV3Interface.sol
│
└── proxy/
    └── ProxyAdminOwnable2Step.sol
```

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:

ID	Description	STATUS
M1	`Vaults` can frontrun an upcoming arbitration, bypass the `Timelocks` and withdraw all their funds	**ACKNOWLEDGED**

***Comments*:**
According to the protocol team, delaying the reward distribution and adding an execution transaction creates a worse experience for the project. Furthermore, if a project decides to skip the withdrawal `Timelock` to get away with funds, they are clearly acting maliciously and should be removed from the platform.

When a `Vault` wants to withdraw its funds, it has to go through the `WithdrawalSystem` which is in turn walled behind a `Timelock`. So, the intended behaviour is to not allow for instant withdrawals at any point or any state of the system or the `Vault`. However, a mechanism that allows a `Vault` to bypass this process exists by using the `RewardSystem` and there are two cases where this situation can be exploited by the `Vault`:

- The `RewardSystem` defines the `sendRewardByVault` function that is expected to be used by the `Vaults` to reward the whitehats for a valid submission. This can be exploited by the `Vaults` by creating a fake issue submission by a `Vault`-controlled "whitehat" and then issuing a reward for this submission summing up to the total holdings of the `Vault`, which effectively performs a covert withdrawal.

- This situation can be further exploited in the scenario where a real whitehat is about to submit an arbitration for that `Vault`. The `Vault` could frontrun the arbitration transaction and use this alternative way to immediately withdraw their funds before the arbitration starts, leaving the `Vault` empty and the upcoming arbitration unable to be fulfilled in the future.

A possible solution would be to set a minimal timelock to that function too, so that to delay the reward distribution slightly and prevent the frontrunning scenario by checking the arbitration status on execution of the actual `sendReward` operation.

M2	Expired queued transactions are still able to be executed in `RewardTimelock`	**RESOLVED**

In the `RewardTimelock` contract, the queued transactions are set to expire based on the `txExpiration` global variable. More specifically, if the elapsed time from their queueing exceeds the `txExpiration` value, then these transactions can not be executed anymore. However, there is no update in their state to indicate that, which means that even if they have expired they maintain their `TxState.Queued` state. It is not either possible to cancel an expired transaction and set its state to `TxState.Canceled`.

To execute a transaction, the following checks need to succeed:

- The transaction's state should be `TxState.Queued`

- The transaction should have matured, which means to have waited `txCooldown` time from the initial submission
- The transaction should not have expired, which means that the current timestamp should not be outside the allowed time window for execution **OR** the `txExpiration` should be **0**, which means that there would not be an expiration for any queued transaction

As a result, if the `txExpiration` value becomes `0` in the future, for any reason, all the previously expired transactions would be again resumed and able to be executed.

This could potentially pose security issues in this situation in case some of the previously expired transactions could disrupt the proper execution of the protocol or any assumptions were made by any entity based on their expired status.

Similarly, transactions queued when the `txExpiration` was `0` would lose this indefinite ability for execution if the variable changes back to a non-zero value rendering them immediately expired if the new value is smaller than the already elapsed time for these transactions.

It is worth mentioning though that the similar `Timelock` contract does not suffer from that issue as each queued transaction stores its own expiration and cooldown data separate from the global variables, hence being resilient in such changes of the global parameters.

`Timelock::queueTransaction():42`

```
function queueTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    Enum.Operation operation,
```

```
    address vault,
    uint256 cooldown,
    uint256 expiration
) external onlyRole(QUEUER_ROLE) {
    ...
    // Dedaub: The cooldown and expiration arguments are the
    //         at-the-time-of-submission global variables provided by the
    //         WithrawalSystem and they are stored locally which makes
    //         them resistant to future changes of the global variables
    txHashData[txHash].queueTimestamp = uint40(block.timestamp);
    txHashData[txHash].cooldown = uint32(cooldown);
    txHashData[txHash].expiration = uint32(expiration);
    txHashData[txHash].state = TxState.Queued;
    ...
}
```

We suggest implementing a similar mechanism to the `RewardTimelock` as well to be sure that no expired transactions would ever be able to be executed in the future and make them resistant to this kind of change, but with that solution, you should also consider the case where transactions with indefinite execution allowance would keep the indefinite allowance even when the `txExpiration` changes which might not be desired in general. So, the solution should special case this category to account for this if it does not follow the desired behavior.

M3	Queued transactions during an arbitration could be later executed (immediately) during a different arbitration	**RESOLVED / ACKNOWLEDGED**

Comments:

According to the protocol team, allowing executing a transaction queued for an arbitration other than those that are currently active is acknowledged and acceptable. However, the immediate execution was prevented and these transactions will be enforced to wait for the cooldown period to finish again before being able to be executed.

In the `RewardTimelock`, a request to send rewards to an address can be queued for a `Vault` only if that `Vault` is currently in arbitration mode. However, the queued transaction is not tied up with the specific `arbitrationId` that allowed the queuing to happen. This allows a queued transaction to be processed even when the initial arbitration has been closed and a new one has been opened. Consider the following scenario for example:

- An arbitration with ID `1` is active for a `Vault`
- The `Vault` queues a transaction using the `RewardTimelock` which is allowed due to the active arbitration
- The arbitration with ID `1` is then closed before the queued transaction matures
- Some time elapses and the transaction matures and can be executed but there is no active arbitration to allow this operation go through
- A new arbitration with ID `2` is then initiated by a whitehat
- The matured and pending transaction can then be immediately executed as the `Vault` is now in arbitration mode again

This behaviour deviates from the assumption taken by the protocol that when an arbitration is activated, the `Vault` would not be able to send any rewards until `txCooldown` period elapses, which is not true for the scenario described above.

A possible solution would be to connect the transaction with the specific `arbitrationId` that allowed its queueing to ensure that if the arbitration closes all the pending transactions get immediately invalidated for future use.

LOW SEVERITY:

ID	Description	STATUS
L1	Inconsistent behavior between `setVaultFee` and `getFee` in `VaultFees` contract	**RESOLVED**
The `setVaultFee` function of the `VaultFees` contract allows the `feeRecipient` of a vault to be set to `address(0)` while the `feeBps` of a vault is set to a particular value. On the other hand, the `getFee` function of the `VaultFees` contract checks whether the `feeRecipient` of a vault is `address(0)`, and in that case, overwrites both the `feeRecipient` and the `feeBps` values of the vault with default values, losing the set `feeBps` value in the process.		
L2	An arbitration can be enabled when the system is in emergency mode	**ACKNOWLEDGED**
When the system is in emergency mode all the `Vaults` are locked and can not make any calls through the `ImmunefiModule`. As a result, a `Vault` would not be possible to initiate an arbitration when in emergency as the `requestArbVault` would revert on the fee payment, which goes through `ImmunefiModule`. However, this is not the case for the `requestArbWhitehat` function. Presently, there is no check for the emergency status of the system and this allows any whitehat to put a `Vault` in arbitration while the system is halted. There do not seem to be any concerning consequences, in particular, for this inconsistency, but it could lead to unintended results in a future update.		
L3	Unbounded number of reward tokens could lead to DoS of a reward distribution	**ACKNOWLEDGED**

In the `RewardTimelock` contract the `Vaults` can queue reward distribution transactions when they are in arbitration. However, if the rewards are split among a large number of tokens, then the execution of the transaction might run out of gas and not be able to be executed.

The `RewardTimelock::executeRewardTransaction` function calculates the value in dollars of the tokens sent for distribution. It iterates over all of them, fetches their prices from Chainlink or other custom oracles and if the sum of the supplied tokens is inside the allowed dollar amount, taking slippage into account, then it continues with the actual execution of the reward distribution through the `ImmunefiModule`. This in turn iterates over all the tokens again to send them to the recipient and also sends a fee to Immunefi for processing the operation for each token.

All this indicates that there are several gas consuming operations happening over the unbounded number of tokens given for execution. Of course, this is user controlled and the caller can adjust the provided tokens so that the transaction does not revert, but if the required dollar amount can not be fulfilled by a smaller number of tokens than the one that runs out of gas, then the Vault would not be able to process the reward distribution. This could happen if the Vault has sent several tokens to the Vault that sum up to the final bounty. This is of course related to the state where an arbitration is enabled, as when it is not, the `Vaults` can use the `RewardSystem::sendRewardByVault` function multiple times to distribute the rewards to the whitehats.

A possible mitigation could be to only allow a bounded number of tokens to be used as possible reward tokens by a Vault.

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	Significant centralisation components	**ACKNOWLEDGED**

***Comments*:**
According to the protocol team, the system allows for these roles to be granted to decentralized organisations if the protocol decides to decentralize those roles in the future.

The ImmuneFi team is able to upgrade most of the contracts of the protocol, arbitrarily changing their logic, and is able to give the `ARBITER_ROLE` and `EXECUTOR_ROLE` to various addresses. The arbiter has wide discretion on how to distribute the funds inside a vault once a claim is filed by a whitehat, and addresses with the `EXECUTOR_ROLE` are able to execute transactions directly on the vault, potentially draining it of funds. The ImmuneFi team have recognised and disclosed these aspects of the protocol to the auditing team and acknowledged that projects using the protocol need to trust ImmuneFi not to abuse these powers. Furthermore, it was stated that the `ARBITRER_ROLE` should be granted to a highly trusted 3rd-party entity.

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	`feeRecipient` can break the `Arbitration` assumptions	**ACKNOWLEDGED**

***Comments*:**

According to the protocol team, this scenario can only happen if the trusted `feeRecipient` address is compromised. Even then, no meaningful impact can happen.

In the `Arbitration` contract, the function `requestArbWhitehat` is meant to be used by a whitehat (or on behalf of them) to initiate an arbitration process on a specific `Vault`. The protocol defines that an Arbitration can be opened only by two ways:

- Either by the `Vault` itself, by calling `requestArbVault`
- Or by a whitehat, by calling `requestArbWhitehat`

In both cases, the caller should have to pay a fee (in the order of $10K) to the `feeRecipient` which is assumed to be a trusted entity. However, the `feeRecipient` has the ability to set a `Vault` in arbitration at no cost, by calling the `requestArbWhitehat` function. The steps to accomplish it are as follows:

- The `feeRecipient` calls the `requestArbWhitehat` function providing the target `vault` as an argument

- For the `whitehat` argument they provide a specially crafted contract that trivially accepts all `ERC1271` signatures which effectively bypasses the check of the `SignatureCheckerUpgradeable.isValidSignatureNow` function
- Finally, `_feeAmount` tokens are self transferred and the execution completes successfully, with the targeted `vault` having registered a new (invalid) `arbitrationId` which immediately puts the `Vault` in arbitration mode

`Arbitration::requestArbWhitehat():71`

```
function requestArbWhitehat(
    uint96 referenceId,
    address vault,
    address whitehat,
    bytes calldata signature
) external {
    bytes32 arbitrationId =
      computeArbitrationId(referenceId, vault, whitehat);
    ...
    // Dedaub: The feeRecipient can register a new arbitration at no cost
    vaultsOngoingArbitrations[vault].add(arbitrationId);
    ...
    require(
        // Dedaub: This check can be bypassed by providing as the whitehat
        //         a specially crafted contract that would accept any
        //         ERC1271 signature
        SignatureCheckerUpgradeable.isValidSignatureNow(
          whitehat, inputHash, signature),
        "Arbitration: invalid request arbitration by whitehat signature"
    );
    ...
    if (_feeAmount > 0) {
        // Dedaub: Self-transfer for the feeRecipient
        feeToken.safeTransferFrom(msg.sender, feeRecipient, _feeAmount);
    }
}
```

As mentioned, the `feeRecipient` is considered to be trusted, but we bring this up for visibility and because it also deviates from the assumptions taken on how the protocol is expected to operate.

A2	Missing checks could report misleading results in `Timelock`	**RESOLVED**

In the `Timelock` contract, the `canExecuteTransaction` function returns whether a `txHash` can be executed or not. However, this function does not check all the parameters that are checked by the actual execution function resulting in misleading answers. More specifically, the function does not check whether the `Vault` is frozen or not. It only checks if the timestamps are in the valid time window that would allow the transaction to execute.

`Timelock::canExecuteTransaction():154`

```
function canExecuteTransaction(
    bytes32 txHash
) public view returns (bool) {
    TxStorageData memory txData = txHashData[txHash];
    // Dedaub: Missing check to ensure that the Vault is not frozen
    return
        txData.state == TxState.Queued &&
        txData.queueTimestamp + txData.cooldown <= block.timestamp &&
        (txData.expiration == 0 || txData.queueTimestamp + txData.cooldown
          + txData.expiration > block.timestamp);
}
```

A3	`Gnosis Safe` used version consideration	**INFO**

The protocol uses the `Safe` contracts at `v1.3.0`. However, in the later versions, an additional check was added to the code of the `checkNSignatures` function when the signature is a contract signature (ref. Safe.sol at v1.4.0).

```
require(keccak256(data) == dataHash, "GS027");
```

We report this here for visibility and possible examination of the differences between the used and the available versions to ensure that all the fixes of future versions have been taken under consideration.

A4	Possible gas optimisations #1	**RESOLVED**

In the `RewardTimelock` contract, there are several places where some gas optimisations are possible by caching the used state variables. For example:

- In the `executeRewardTransaction`, `cancelTransaction` and `canExecuteTransaction` functions, there are several checks to ensure that the cooldown and the expiration periods are being respected for a specific `txHash`. However, the statements use the `txCooldown` and `txExpiration` storage variables twice during that process. You could cache these values to save the unnecessary `SLOADs` (`2` for the `executeRewardTransaction` and the `canExecuteTransaction` and `1` for the `cancelTransaction`).

  `RewardTimelock::executeRewardTransaction():105`

```
function executeRewardTransaction(
    bytes32 txHash,
    ...
) external {
    ...
    // Dedaub: You could cache both txCooldown and txExpiration
    //         to save 2 SLOADs
    require(
        txData.queueTimestamp + txCooldown <= block.timestamp,
        "RewardTimelock: transaction is not yet executable"
    );
    require(
        txExpiration == 0 || txData.queueTimestamp + txCooldown +
          txExpiration > block.timestamp,
        "RewardTimelock: transaction is expired"
    );
```

```
    ...
}
```

RewardTimelock::cancelTransaction():140

```
function cancelTransaction(
    bytes32 txHash
) external {
    ...
    // Dedaub: You could cache txExpiration to save 1 SLOAD
    require(
        txExpiration == 0 || txData.queueTimestamp + txCooldown +
          txExpiration > block.timestamp,
        "RewardTimelock: transaction is expired"
    );
    ...
}
```

RewardTimelock::canExecuteTransaction():184

```
function canExecuteTransaction(
    bytes32 txHash
) external view returns (bool) {
    ...
    // Dedaub: You could cache both txCooldown and txExpiration
    //         to save 2 SLOADs
    return
        txData.state == TxState.Queued &&
        txData.queueTimestamp + txCooldown <= block.timestamp &&
        (txExpiration == 0 || txData.queueTimestamp + txCooldown +
          txExpiration > block.timestamp);
}
```

A5	Possible gas optimisations #2	RESOLVED

In the `ScopeGuard` and in `AccessControlGuardable` contracts, there are several places where gas optimisations are possible by using the function arguments to emit the events instead of the state variables. For example:

- All the setter functions of the `ScopeGuard` (`setTargetAllowed`, ..., `setAllowedFunction`), simply update the values of the storage variables and emit an event related to this operation. However, the event is emitted by fetching the newly updated value from storage when the function argument could be used instead to save that unnecessary SLOAD.

  `RewardTimelock::executeRewardTransaction():105`

```
function setTargetAllowed(
    address target,
    bool allow
) public onlyOwner {
    // Dedaub: You could use the argument allow to emit the event
    //         instead of retrieving the recently updated with the
    //         same value storage variable
    allowedTargets[target].allowed = allow;
    emit SetTargetAllowed(target, allowedTargets[target].allowed);
}
```

- The `setGuard` of the `AccessControlGuardable` also emits the event using the recently updated storage variable instead of using the provided argument.

  `AccessControlGuardable::setGuard():30`

```
function setGuard(
    address _guard
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    ...
    // Dedaub: Use the _guard variable to save an SLOAD
    guard = _guard;
    emit ChangedGuard(guard);
```

```
}
```

A6	Possible extra validation in `RewardTimelockBase`'s `_getVaultTxsPaginatedReversed` function	**INFO**

The `_getVaultTxsPaginatedReversed` function of the `RewardTimelockBase` contract has a for loop which counts down by initially setting the loop index `i` to `txHashes.length - start - 1`. However if `start >= txHashes.length`, the loop will revert because variable `i` is of type `uint256`. It may make sense to require that `start < txHashes.length` as an additional validation in order to avoid this scenario and return an indicative error message instead.

A7	Inconsistent parameter datatype in `WithdrawalSystem`'s `setTxCooldown` functions	**RESOLVED**

The `WithdrawalSystem` contract has two functions `setTxCooldown` and `_setTxCooldown` which take a cooldown parameter of type `uint256`, which is eventually cast to `uint32` when either of the functions are run.

This is inconsistent with the `setTxCooldown` and `_setTxCooldown` functions of the `RewardTimelockBase` contract, which take a `uint32` as the `newTxCooldown` parameter and avoid this cast.

One of these representations should be chosen unless there is a good reason for this divergence.

A8	Missing check for the `txCooldown` of the `WithdrawalSystem` used by the `Timelock` contract	**INFO**

In the `WithdrawalSystemBase` contract, the `_setTxCooldown` function directly sets the given value to the `txCooldown` variable without requiring it to be greater than `0`. In

contrast, the `RewardTimelockBase` similar setter has such a check ensuring that the new `txCooldown` is `> 0`.

A9	Commented out `require` statement	**RESOLVED**

In the `ArbitrationBase` contract, the `_setFeeAmount` function has a commented out require statement that if enabled would prevent the `feeAmount` to be set to `0`. We bring this up for visibility in case it was left commented out by mistake and the desired behavior would be to have non-zero `feeAmount`s.

A10	Typos in comments	**RESOLVED**

There are a few places where there are a few typos in the comments:

- `AccessControlBaseModule:39`
  `AccessControlBaseModule:78`

```
// Zero out the redundant transaction information only used for Safe multisig trans[a]ctions.
```

- `AccessControlGuardable:18`

```
// Dedaub: It says guard_ when the variable used is named _guard
// `guard_` does not implement IERC165.
```

- `ScopeGuard:75`

```
@notice Sets whether or not a target can be sent to (inclue[d]es fallback/receive functions).
```

A11	Compiler bugs	**INFO**

The code is compiled with Solidity `0.8.18`. Version `0.8.18`, in particular, has some known bugs, which we do not believe 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.