PUBLIC

Code Assessment

of the zkBob

Smart Contracts

January 5, 2023

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Igor,

Thank you for trusting us to help BOB Protocol with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of zkBob according to Scope to support you in forming an opinion on their security risks.

BOB Protocol implements an application that uses zero-knowledge proofs (zk-SNARKs) for anonymous transfers of the BOB ERC20 stablecoin token.

The most critical subjects covered in our audit are functional correctness, access control, and front-running. Security regarding functional correctness and access control is high. The two uncovered medium severity issues, that make the system vulnerable to front-running and sandwich attacks can potentially endanger users and 3rd party integrations, but do not pose an immediate risk for the ZkBob system itself.

The general subjects covered are trustworthiness, documentation, specification and code complexity. The security regarding these subjects is good. The acknowledged and not fixed issues are of low severity and don't render the system unsafe.

In summary, we find that the codebase provides a good level of security. The remaining acknowledged but not fixed issues do not immediately impair the system, however, we still suggest addressing them in the future.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	8
• Code Corrected	4
• Specification Changed	1
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the zkBob repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	29 Oct 2022	a1ce1231e90556839db8305d2190c8ca131a1093	Initial Version
2	21 Dec 2022	354fea530948b45aedca3479a73ddcf31bf5b8e8	Version with fixes

For the solidity smart contracts, the compiler version `0.8.15` was chosen.

These three main contracts were in scope:

- zkbob/ZkBobPool.sol
- BobToken.sol
- BobVault.sol

As well as the following dependencies:

- proxy/EIP1967Admin.sol
- token/BaseERC20.sol
- token/ERC20Blocklist.sol
- token/ERC20MintBurn.sol
- token/ERC20Permit.sol
- token/ERC20Recovery.sol
- token/ERC677.sol
- utils/Claimable.sol
- utils/EIP712.sol
- utils/Ownable.sol
- utils/Sacrifice.sol
- utils/UniswapV3Seller.sol
- yield/AAVEYieldImplementation.sol
- yield/YieldConnector.sol
- zkbob/manager/MutableOperatorManager.sol
- zkbob/manager/SimpleOperatorManager.sol
- zkbob/utils/CustomABIDecoder.sol
- zkbob/utils/Parameters.sol
- zkbob/utils/ZkBobAccounting.sol

2.1.1 Excluded from scope

Although we performed basic checks on two zk-SNARK verifiers (`TransferVerifier.sol` and `TreeUpdateVerifier.sol`), they are out of scope for this audit.

Third-party libraries, test contracts and any files not listed above are not in scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

zkBob is an application that uses zero-knowledge proofs (zk-SNARKs) for anonymous stablecoin transfers. Daily limits are imposed to prevent large amounts of money that may be related to illegal activity from being deposited or withdrawn. Transfers amongst users in the pool, however, are unlimited.

The zkBob app is designed to work with the BOB stablecoin. BOB is an upgradeable ERC-20 token, which is currently available on Polygon, Optimism, BNB Chain, and Ethereum mainnet. Address blocking and token recovery functions are also implemented.

Once BOB tokens are deposited in the zkBob app, users can initiate transfers among themselves through private transactions. Typically, transactions will be routed through a relayer, to avoid identifying information being leaked because of gas fees. The relayer will also help make sure that transactions are sent to the pool in sequence.

Below is a summary of the main contracts in this system:

- `BobToken.sol`: ERC-20 token that also implements the EIP-1967 and ERC-677 standards.
- `BobVault.sol`: allows users to buy and sell BOB tokens in exchange for stablecoin collateral at a fixed rate. The locked collateral can be invested in yield-generating protocols. At the moment, the only available yield provider is AAVE.
- `ZkBobPool.sol`: the pool contract processes incoming transactions via the `transact` function. The function will check that the message sender is an authorized operator (typically a relayer).

2.2.1 Trust Model

We assume that the following parties are trusted and behave correctly:

- The proxy admins of the BOB token and the Bob vault can change the respective implementations. These contracts follow an upgradeable proxy pattern, as described in the EIP-1967 standard. The proxy admin of the Bob vault also has the privileges of the contract owner (see below).
- The contract owner of the Bob vault (implemented through `Ownable`) can add collateral types, enable/disable yield earning on a particular token, and change fee rates. The owner also has the privileges of the yield admin and the invest admin (see below). Note that we assume all tokens provided as collateral by the vault owner are not reentrant tokens, i.e. a transfer to or from a user does not give control flow over to the user.
- The yield admin can farm (i.e., collects fees and generated yield).
- The invest admin can invest excess tokens into the yield provider.
- The blocklister of the BOB token can block and unblock accounts from using the token contract.
- The recovery admin of the BOB token can recover funds from arbitrary accounts, e.g. if they are frozen. Note that there is a waiting period before the funds can be claimed, but once that process has been started a user can't stop their funds being taken except by transferring them to another account.

- The operator is the only entity that can directly send transactions to the zkBob pool for processing. Typically, the operator is a single relayer. However, if the operator is defined as `address(0)` through `MutableOperatorManager`, then any user can act as an operator. BOB Protocol also plans to support a multi-relayer model in the future. Note that the operator is also responsible for ensuring that users do not spread their funds between multiple EOAs to avoid the daily limits imposed by the *ZkBobAccounting* contract. In the case of users being able to call `transact` themselves, this type of attack cannot be mitigated. See Daily Limits can be avoided.

In general, we consider that entities assuming such roles are honest. In the case of decentralized governance, we assume that there is an honest majority behind each administrative role.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- ZkBobPool Withdrawal Sandwich Attack Risk Accepted

Low-Severity Findings	3

- BaseERC20 Overflow Risk Accepted
- Daily Limits Can Be Avoided Risk Accepted
- ERC20Permit.receiveWithPermit Signature Can Be Front-Run Risk Accepted

5.1 ZkBobPool Withdrawal Sandwich Attack

Design Medium Version 1 Risk Accepted

When the withdrawal with `native_amount` is submitted to the ZkBobPool, the sale of tokens for ETH happens using the `UniswapV3Seller` contract. However, the `amountOutMinimum` parameter of this swap is 0. A potential attacker can place orders that would manipulate the price, forcing the `sellForETH` trade to be executed with a bad price. Thus, due to the lack of spread control, any use of `UniswapV3Seller` can result in a bad trade, allowing price manipulators to pocket the profit from this trade.

Risk accepted:

BOB Protocol responded:

> This feature is only intended to swap small amounts of tokens, purely for funding wallets with gas tokens. UI will strongly dissuade users for doing swaps that are larger than e.g. 100$ in value. Added a warning comment to the `sellForETH` function.

5.2 BaseERC20 Overflow

Correctness Low Version 1 Risk Accepted

The `_increaseBalance` function of the *BaseERC20* contract can overflow. While it is checked that the account is not frozen (i.e. the first bit of the balance is zero), it is not guaranteed that the addition will result in a number smaller than 2^255. Hence, an account could become frozen by increasing its balance above this value.

Risk accepted:

Assuming a reasonable total supply of the token (less than 2^255), it is impossible for any individual account to have a balance large enough to cause this overflow to happen. Thus, the overflow cannot occur under normal circumstances.

5.3 Daily Limits Can Be Avoided

Trust **Low** **Version 1** **Risk Accepted**

If the *MutableOperatorManager* is used and the `operator` variable is set to 0, then effectively every user becomes an operator / relayer. This means that any user could spread funds between multiple addresses and easily avoid the daily limits imposed by the *ZkBobAccounting* contract.

Risk accepted:

BOB Protocol accepts the risk regarding users avoiding daily limits and states:

> Allowing users to submit transactions themselves introduces multiple potential problems, including the one described with the limits. For now, it can be assumed that deposit transactions might only go through the chosen relayer, which is also responsible for detecting abnormal limit usage. Even though we cannot assume that one address is equal to one user, we think that making per-address limits in the contract can still be useful in certain use-cases.

5.4 ERC20Permit.receiveWithPermit Signature Can Be Front-Run

Design **Low** **Version 1** **Risk Accepted**

Similar to issue ERC20Permit.receiveWithSaltedPermit signature can be front-run, the signatures between `ERC20Permit.permit` and `ERC20Permit.receiveWithPermit` are interchangeable as well.

Thus, the attacker can front-run the signatures and use them in unintended functions to cause a user's transactions to fail. However, this does not render the ZkBob system unsecure itself but might cause problems for 3rd party integrations. Thus, the severity of this issue is low.

Risk accepted:

BOB Protocol accepted the risk and stated:

> Third party integrations relying on permit/receiveWithPermit are advised to implement necessary fallbacks for failing permit/receiveWithPermit calls, avoiding entire transaction failures.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- ZkBobPool Fees Can Drain the Deposits of the User Code Corrected

Medium-Severity Findings	1

- ERC20Permit.receiveWithSaltedPermit Signature Can Be Front-Run Code Corrected

Low-Severity Findings	5

- Admin Reentrancy in ERC20Recovery Code Corrected
- Avoiding Recovery by Admin Specification Changed
- BobVault Uint Conversions Code Corrected
- Missing Sanity Checks Code Corrected
- No Events on State Changes Code Corrected

6.1 ZkBobPool Fees Can Drain the Deposits of the User

Security High Version 1 Code Corrected

When depositing using the `transact` function, the caller can specify a negative token amount. Normally, this would revert as it is checked that the deposit amount is positive. However, if the user also specifies the fee to be greater than the absolute value of the deposit amount, the total `token_amount` will be positive, hence passing the check. Thus, the deposit will go through. This can be exploited by a malicious operator in the following scenario:

1. Operator (msg.sender) specifies `txType = 0` (deposit), `_transfer_token_amount = -400`, `fee = 500`.
2. Computed `token_amount` will be `100`.
3. `100 * TOKEN_DENOMINATOR` will be transferred to the ZkBobPool from the user address.
4. `accumulatedFee[msg.sender]` will be increased by 500.
5. Operator withdraws `500 * TOKEN_DENOMINATOR` of fees.

Thus, by depositing only 100 tokens malicious operator was able to withdraw 500 tokens as fees. The malicious operator can drain the contract via the fees.

Note that the `transact` function is only callable by the privileged *Operator* role. However, the *OperatorManager* contracts can be configured such that every user would be an operator.

Code corrected:

BOB Protocol confirmed that the case of asset drain was prevented by the verifier and the snark circuits which are out of scope for this engagement. However, the deposit of a negative value still could have been used as an undesired withdrawal. Stronger checks were introduced, namely a requirement that `_transfer_token_amount` must be positive for deposits. The check during the withdrawal correctly constrains the `token_amount`, since otherwise the same issue would occur in the other direction - a small negative `_transfer_token_amount` + big positive fee can be positive, causing a withdrawal to count as a deposit.

The full response of BOB Protocol team:

> This confusing case is handled correctly by the verifier and snark circuits.
>
> During usual deposit, the following happens:
>
> 1. User deposit amount is 400 (_transfer_token_amount is 400)
> 2. Relayer adds a 100 fee on top
> 3. Pool contract executes transferFrom for 500 (400 + 100) tokens
> 4. User shielded balance is increased by _transfer_token_amount (400), which is verified by the circuit verifier
> 5. Relayer receives a 100 fee
> 6. So 500 transferred tokens were divided between user (+400) and relayer (+100)
>
> During the suggested negative deposit, the following happens:
>
> 1. User deposit amount is -400 (_transfer_token_amount is -400, better to think of it as a balance delta, rather than deposit amount)
> 2. Relayer adds a 500 fee on top
> 3. Pool contract executes transferFrom for 100 (-400 + 500) tokens
> 4. User shielded balance is increased by _transfer_token_amount (-400), which is verified by the circuit verifier. As the balance delta is negative, the balance is actually being decreased by 400.
> 5. Relayer receives a 500 fee.
> 6. In the end, relayer receives 500 tokens, comprised of user shielded balance decrease (400) and external token transfer (100)
> 7. Essentially, this turned a deposit into a very strange version of withdrawal
>
> Although balance accounting works correctly here, this situation is indeed very confusing. It cannot be triggered via the UI or SDK, as it just does not make sense for users to do something like that. To get rid of this unnecessary source of confusion, we will introduce a bit stricter validation on the deposit amounts, so that negative _transfer_token_amount are not allowed.

6.2 ERC20Permit.receiveWithSaltedPermit Signature Can Be Front-Run

Design **Medium** **Version 1** **Code Corrected**

The `ZkBobPool` permittable deposit relies on the `ERC20Permit.receiveWithSaltedPermit` function. However, the signature used in this function can be used in the `ERC20Permit.saltedPermit` function as well. An attacker can intercept the deposit transaction, extract the signature and use it in the call to `saltedPermit`. As a result of this action, the permittable deposit will fail due to the nonce already having been used. Thus, the attacker can front-run the signatures and use them in unintended functions to cause a user's transactions to fail.

Code corrected:

The `saltedPermit` function was removed. Hence, a permittable transaction can't be front-run with a call that uses the same signature for another function.

6.3 Admin Reentrancy in ERC20Recovery

Security | Low | Version 1 | Code Corrected

The `executeRecovery` function in *ERC20Recovery* can only be called by the owner or the recovery admin. When recovering the tokens, they are transferred to the `recoveredFundsReceiver` address. If this address is a contract, the `onTokenTransfer` function is called. This call could be used to reenter the `executeRecovery` function in order to double-claim the funds to recover. This would allow the recovery admin or the owner to exceed the intended `recoveryLimit`.

As the `recoveredFundsReceiver` can only be set by the owner, and both the owner and recovery admin are trusted addresses, the impact of this issue is limited.

Code corrected:

The `recoveryRequestExecutionTimestamp` and `recoveryRequestHash` are now deleted before any external calls are made. Hence, if a reentrant call later calls `executeRecovery` again, there will be no stored timestamp or hash, so the funds can't be double-claimed anymore.

6.4 Avoiding Recovery by Admin

Design | Low | Version 1 | Specification Changed

If a user sees that their account is marked for recovery (using `ERC20Recovery.requestRecovery()`), they can simply transfer funds to another account to stop them from being recovered. It may also make sense from the perspective of trustworthiness to only allow recovery of funds e.g. if the account is already frozen, or at least enforcing that an account must be frozen in order to recover its funds.

Specification corrected:

BOB Protocol responded:

> Recovery functionality is intended to be used only on dormant or non-existing users, if the user is able to move his funds to a different address, his token should not be allowed for recovery. Recovering frozen is a different use-case, although it can be also executed through the same functionality.

With the assumption, that the proper checks will be performed before account recovery, this issue is resolved.

6.5 BobVault Uint Conversions

Design | Low | Version 1 | Code Corrected

To track the `token.balance` the BobVault contract uses uint128 values. Theoretically, it is possible to provide a value that is great than `type(uint128).max`. This case will not be handled correctly by the code due to the unsafe conversion to uint128, which truncates the value. As a result, internal accounting will be broken. This happens in multiple functions such as: `buy`, `sell`, `swap`, `give`.

```
token.balance += uint128(sellAmount);
```

The amount before conversion in most cases is used as an argument for token transfer. However, the practical safety of this conversion depends on the external contract, which is not optimal.

Code corrected:

BOB Protocol responded:

> Although such extremely high amounts won't be seen in practice, we added additional overflow checks where necessary.

The checks were introduced in `buy`, `swap`, `give`. Check performed in `sell` is sufficient to prevent the overflow.

6.6 Missing Sanity Checks

Design | Low | Version 1 | Code Corrected

Many state-changing operations do not include sanity checks to ensure incorrect values are not set. Consider adding checks to ensure these values aren't accidentally set incorrectly. This can happen e.g. due to a bug in a front-end application resulting in empty values in calldata.

These operations include:

ERC20Blocklist:

- `updateBlocklister()` does not check that `_newBlocklister` is not `address(0)`.

ERC20Recovery:

- `setRecoveryAdmin()` does not check that `_recoveryAdmin` is not `address(0)`.
- `setRecoveredFundsReceiver()` does not check that `_recoveredFundsReceiver` is not `address(0)`.

Claimable:

- `setClaimingAdmin` does not check that `_claimingAdmin` is not `address(0)`.

ZkBobPool:

- `constructor()` does not check any of the provided addresses.
- `initialize()` does not check that `_root` is not `0`.
- `setTokenSeller()` does not check that `_seller` is not `address(0)`.
- `setOperatorManager()` does not check that `_operatorManager` is not `address(0)`.

BobVault:

- `constructor()` does not check that `_bobToken` is not `address(0)`.
- `setYieldAdmin()` does not check that `_yieldAdmin` is not `address(0)`.
- `setInvestAdmin()` does not check that `_investAdmin` is not `address(0)`.

Code corrected:

Checks were added where necessary. Explanation was added why certain cases do not need sanity checks.

BOB Protocol responded:

> We added a few sanity checks in places there we think they might be important:
>
> - ZkBobPool: constructor(), initialize(), setOperatorManager()
> - BobVault: constructor()
>
> In other places, zero addresses are used for unsetting the specific privileges and rights:
>
> - ERC20Blocklist: updateBlocklister() – zero address is used to limit the ability to block/unblock accounts only by the governance.
> - ERC20Recovery: setRecoveryAdmin() – zero address is used to limit the ability to recover funds only by the governance. setRecoveredFundsReceiver() – there is a check that recoveredFundsReceiver is not zero in _remainingRecoveryLimit (link) so it is safe to not introduce additional checks
> - Claimable: setClaimingAdmin() – zero address is used to limit the ability to claim tokens only by the governance.
> - ZkBobPool: setTokenSeller() – zero address is used to disable the ability for users to swap small amount of BOB tokens to MATIC during the withdrawal process.
> - BobVault: setYieldAdmin() - zero address is used to limit the ability to collect generated yield by the governance; setInvestAdmin() – zero address is used to limit the ability invests tokens into the yield provider only by the governance.
>
> Moreover, these functions should only be called by the admin via governance process (e.g. from Safe UI), making real UI typos very unlikely to happen.

6.7 No Events on State Changes

Design | Low | Version 1 | Code Corrected

Many state-changing operations do not emit events. Consider emitting events for important state changes.

These operations include:

ZkBobPool:

- `initialize()`
- `setTokenSeller()`
- `setOperatorManager()`

ZkBobAccounting:

- `_setLimits()`
- `_resetDailyLimits()`
- `_setUsersTier()`

BobVault:

- `setYieldAdmin()`

- `setInvestAdmin()`

MutableOperatorManager:

- `_setOperator()`

ERC20Recovery:

- `setRecoveryAdmin()`
- `setRecoveredFundsReceiver()`
- `setRecoveryLimitPercent()`
- `setRecoveryRequestTimelockPeriod()`

Claimable:

- `setClaimingAdmin()`

Code corrected:

Events were added to the following functions:

ZkBobPool:

- `setTokenSeller()`
- `setOperatorManager()`
- `withdrawFee()`

ZkBobAccounting:

- `_setLimits()`

MutableOperatorManager:

- `_setOperator()`

The remaining functions are either not expected to be called regularly, or it was deemed unimportant for the functions to emit events.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 BobVault.disableCollateralYield Potential Reentrancy

Note Version 1

The token buffer, dust and yield fields are updated after the external call. If the yield contract has a reentrancy point, where BobVault can be called again - this update can happen in an invalid state. The external calls should happen after all the state variable updates.

Code corrected:

Statements were reordered to make the reentrancy impossible.

7.2 ERC20Permit Deletes Existing Approvals

Note Version 1

Using any of the public functions in *ERC20Permit* will zero out any pre-existing approval a user may have had from the signer. Hence, a user should use any existing approval from the signer before calling `permit` or its variations.

7.3 Incorrect Comment

Note Version 1

In the *CustomABIDecoder* contract, the `_memo_fixed_size` function features the following comment:

```
else if (t == 2) {
    // fee + recipient + native amount
    // 8 + 20 + 8
    r = 36;
}
```

However, in the case of a *Withdraw* operation the order in calldata is actually `... | fee | native_amount | receiver | ...`. The given sizes for the fields are correct (but also in the wrong order).

Code corrected:

The comment was fixed.

7.4 Reentrant Tokens

Note **Version 1**

The *BobVault* contract should not use any tokens with reentrant transfers, such as an ERC777 token, as collateral. This could lead to inconsistent event orderings or potentially more severe issues. This audit was performed with the assumption that any tokens used as collateral do not have reentrant functionality.

Similarly, the *ZkBobPool* contract should not use an underlying token with reentrant calls, as it would open up critical vulnerabilities such as draining the contract's balance through the `withdrawFee` function.

7.5 Unused Constant

Note **Version 1**

In `CustomABIDecoder.sol`, the `sign_r_vs_size` constant is defined but never used.

BOB Protocol responded:

> We won't delete the constant, as keeping it does not impact the size/gas cost of the produced bytecode (most likely it is being pruned by the optimizer), but it might become useful in the future, for adding more extra fields.

7.6 Unused Return Data in `YieldConnector`

Note **Version 1**

The *YieldConnector*'s `_delegateFarmExtra` function does not return anything, even though the `farmExtra` function of the *IYieldImplementation* interface returns a `bytes` type.

Similarly, the `claimTokens` function of the *Claimable* contract does not check the return value of `IERC20(_token).transfer(_to, balance)`. Hence, `false` could be returned (meaning the transfer did not actually take place).

Code corrected:

BOB Protocol responded:

> Acknowledged and added missing return statement from farmExtra() function. The _delegateFarmExtra() function is unused in the context of existing AAVE deployments, however, it might be used for other lending markets integrations (e.g. Compound). Calls to IERC20(_token).transfer(_to, balance) in the Claimable contract are only intended to be executed within the manual governance process, thus actual transfer result does not imply any considerable impact on the system.