PUBLIC

Code Assessment

of the PositionManager Smart Contracts

January 10, 2023

Produced for

by

Contents

1 Executive Summary

Dear Oasis team,

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

PositionManager implements a way for users to easily create proxy contracts to manage various DeFi positions without built-in asset separation. The proxies should be capable of replacing instances of MakerDAO's DSProxy.

The most critical subjects covered in our audit are functional correctness, system design and safety of user funds. We uncovered two medium severity issues regarding functional correctness, which have been addressed. There was one high severity issue regarding system design, which also has been remedied.

The general subjects covered are gas efficiency, code complexity, trustworthiness and access control. Some improvements can be made to the gas efficiency of the contracts. Security regarding the remaining subjects is high.

In summary, we find that the codebase provides a high level of security.

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	17
• Code Corrected	12
• Code Partially Corrected	4
• Acknowledged	1

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 PositionManager 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	21 Oct 2022	92a8d03278a637bcf6c5d109d0d7c8992a711d5a	Initial Version
2	22 Nov 2022	7283b901a3e4b183f9514c898ba3e6e7af5a5a64	First iteration of fixes
3	25 Nov 2022	d70319e59d1337fad9101965ac40366cdf530466	Second iteration of fixes
4	19 Dec 2022	19b75d52660ae72c9b510efaf0b57f8916745016	Minor change

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

The following files are in scope:

- AccountFactory.sol
- AccountGuard.sol
- AccountImplementation.sol
- ImmutableProxy.sol

2.1.1 Excluded from scope

Third-party libraries, test files, 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.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Oasis has implemented the DeFi Position Manager, a set of smart contracts allowing users to create multiple proxies from one address to manage different positions on protocols without built-in asset separation. The proxies should be capable of replacing MakerDAO's DSProxy, acting in an equivalent way.

The creation of a new proxy mechanism was used as an opportunity to redesign some parts of the Maker proxy design to achieve the following properties:

1. Cheaper deployment of new proxies.
2. Ability for a single user to own multiple proxies.
3. Whitelisting of contracts, preventing the execution of untrusted logic within the proxy. This is a mitigation for DNS attacks on frontends.
4. Ability to do normal calls to contracts, not just delegatecalls.

2.2.1 AccountFactory

AccountFactory is responsible for the creation of new proxies. It deploys minimal proxies that call into the *AccountImplementation*. Once assigned, the *AccountImplementation* cannot be changed.

AccountFactory has the following public functions:

- `createAccount(address user)` creates a new proxy for the given user using the OpenZeppelin Clones library and makes them the owner. It emits the `AccountCreated` event, which contains a unique ID.
- `createAccount()` calls `createAccount(address user)` with `msg.sender` as `user`.

2.2.2 AccountGuard

AccountGuard is responsible for storing and managing permissions of all proxies created with *AccountFactory*. The *AccountGuard* owner can add addresses to a whitelist. Proxies can only call and delegatecall to whitelisted addresses. The *AccountGuard* owner cannot tamper with any proxy permissions. Every proxy also has a separate owner, which is stored in the `owners` mapping. The proxy owner can transfer ownership to a new account and can grant execution permissions to other addresses. Permitted addresses cannot remove the owner's permissions, but can add or remove permissions from all other addresses.

AccountGuard has the following public functions:

- `isWhitelisted()` returns true if the given address is whitelisted.
- `setWhiteList()` adds or removes an address in the whitelist. Only callable by the *AccountGuard* owner.
- `canCall(address target, address operator)` returns true if the `operator` is either the proxy owner or permitted on the `target` proxy.
- `initializeFactory()` is used to set the *AccountFactory* after deploying. This can only be done once and it must be ensured that it was done correctly, as there is a risk of frontrunning.
- `permit(address caller, address target, bool allowance)` adds or removes the `caller` from being permitted on the `target` proxy. Can only be called by an address that is already permitted. It has a special case when called by the factory contract, which then causes the `caller` address to be set as owner, as well as giving the proxy the permission to call itself.
- `changeOwner(address newOwner, address target)` transfers ownership of the `target` proxy to the `newOwner` and removes permissions from the previous owner. Any other permitted addresses keep their permissions.

2.2.3 AccountImplementation

AccountImplementation contains the implementation that the minimal proxies point to. It does access control through the *AccountGuard* and can call or delegatecall arbitrary whitelisted addresses.

AccountImplementation has the following public functions:

- `send(address _target, bytes memory _data)` calls the `_target` address with `_data` as calldata. This can call arbitrary functions at a `_target` contract by including the function selector in

the calldata. It also forwards any ETH included in the transaction to the called contract, so it can be used to send ETH. It reverts if the call fails.

- `execute(address _target, bytes memory _data)` does a delegatecall to `_target` with `_data` as calldata. It returns the first 32 bytes of return data from the delegatecall and reverts if the call fails.

Both functions can only be called by addresses permitted by the proxy and will revert unless `_target` is whitelisted. Users *must be careful* not to delegatecall into two different contracts that both read or write the same storage slot, as this will break the storage.

2.2.4 ImmutableProxy

ImmutableProxy is an instantiation of OpenZeppelin's *Proxy.sol* with a fixed implementation address that is set during deployment.

2.2.5 Trust assumptions

The owner of *AccountGuard* is trusted not to remove any addresses from the whitelist that would lead to funds being stuck on a proxy. Users are untrusted and can give permissions for the proxies they own to anyone they trust. They cannot affect proxies owned by other users.

2.2.6 Changes in Version 2

The *ImmutableProxy* contract was removed.

The whitelist was split into 2 parts - one for delegate calls and one for regular calls. Hence, there are the new functions `isWhitelistedSend()``and ``setWhitelistSend()` to mirror the existing whitelist functions.

The *AccountGuard* has a new function `canCallAndWhitelisted(address proxy, address operator, address callTarget, bool asDelegateCall)`. It returns true if the `operator` address is permitted to use the specified `proxy` and if the `callTarget` is whitelisted either as a delegatecall target or a regular call target, depending on the `asDelegateCall` parameter.

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:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

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

- execute() Gas Calculation Can Underflow Risk Accepted

Low-Severity Findings	5

- Inefficient Boolean Mappings Code Partially Corrected Acknowledged
- External Calls Could Be Avoided Acknowledged
- Function Arguments Can Be In Calldata Code Partially Corrected Acknowledged
- Inefficiencies in execute() Code Partially Corrected Risk Accepted
- Various Event Issues Code Partially Corrected Acknowledged

5.1 `execute()` Gas Calculation Can Underflow

Correctness Medium Version 1 Risk Accepted

In *AccountImplementation*, `execute()` subtracts `5000` from `gas()` using assembly subtraction. If `gas()` is less than `5000`, this will underflow and wrap around to a very high value.

The solidity overflow checks are disabled when using assembly, so this will not revert.

Risk accepted

Oasis accepts the risk of this underflow occurring and states that under normal use this will not happen, as the proxy is intended for state-changing operations and hence the delegate call should never cost less than 5000 gas to execute.

5.2 Inefficient Boolean Mappings

Design Low Version 2 Code Partially Corrected Acknowledged

The *AccountGuard* has the following mappings:

```
mapping(address => mapping(address => bool)) private allowed;
mapping(address => mapping(bool => bool)) private whitelisted;
```

Mappings that contain boolean variables are inefficient in general, as the Solidity compiler masks the values when saving and loading these values from storage to prevent dirty bits from contaminating the values.

For the `whitelisted` mapping, there is a superfluous level of indirection. The `(bool => bool)` mapping could be replaced by a different type, e.g. an enum or an integer, which would reduce the number of SLOADs needed to retrieve an entry. Alternatively, two separate mappings could be used for the two whitelists.

Code partially corrected

The type of the `whitelisted` mapping was changed to `mapping(address => uint8)`. The `allowed` mapping was not modified.

Acknowledged

Note that reading and writing `uint8` and `bool` values to and from storage results in inefficient bytecode, as the Solidity compiler masks the values to ensure no dirty bits are propagated. As there is no possibility of writing to arbitrary storage locations, these checks are superfluous. Using mappings that store the `uint256` type eliminates these checks. Oasis has acknowledged this inefficiency.

5.3 External Calls Could Be Avoided

Design | Low | Version 1 | Acknowledged

In the EVM, external calls to other contracts cost more gas than internal calls to a contract's own functions.

The *AccountFactory* makes external calls to *AccountGuard*. Both contracts are dependent on one another - they need to know each other's addresses in order to set the owners of newly created proxies. This introduces a circular dependency.

As neither contract is upgradeable and neither contract allows setting a new address for the other, the increased gas cost and circular dependency could be avoided by combining both contracts' functions into a single contract. This also allows for the removal of some storage variables.

Acknowledged

Oasis has acknowledged the issue.

5.4 Function Arguments Can Be In Calldata

Design | Low | Version 1 | Code Partially Corrected | Acknowledged

The `bytes memory _data` arguments in `AccountImplementation.send()` and `AccountImplementation.execute()` can be in calldata instead of memory in order to save gas.

Code partially corrected

The `_data` argument of the `send()` function was changed to `bytes calldata`.

Acknowledged

For `execute()`, Oasis acknowledged the issue and decided not to make code changes.

5.5 Inefficiencies in `execute()`

Design | Low | Version 1 | Code Partially Corrected | Risk Accepted

The `execute()` function in *AccountImplementation* has multiple inefficiencies:

- Only `gas() - 5000` gas is passed to the delegate call. This may force users to attach more gas than needed to their calls. While it does guarantee that the remaining operations after the delegatecall will complete, it also makes it more likely for the delegatecall to run out of gas. It is simpler to just attach all the gas to the call and assume it returns before consuming all the gas.
- `execute()` only returns 32 bytes of return data from the delegate call. If any contract that is called returns more than 32 bytes of data, it will be lost.
- The return data is not directly returned in the assembly block, but by the function itself. This means that it will be copied before returning, which expands the memory and costs more gas. Instead, it could be directly returned in the assembly block using a `return` call.
- The switch condition can be changed to `succeeded` instead of `iszero(succeeded)`. There is no reason to invert the condition.

For an example implementation of a similar function, see the `_delegate` function in OpenZeppelin's *Proxy* contract. It allows returning more than 32 bytes of data. Note that its functionality may not be completely identical, so the code should not be directly copied.

Code partially corrected

The return data is now directly returned in the assembly block. Additionally, the switch condition was changed to `succeeded`, eliminating the extra `iszero` operation.

The newly introduced line `returndatacopy(0, 0, returndatasize())` is used to copy the revert data to memory in case of failure. It is redundant in the success case, as the delegatecall already copies the first word of returndata to memory. If the delegate call returns more than 32 bytes of data, it will expand the memory unnecessarily.

Risk accepted

Oasis accepts the risks regarding users being forced to attach too much gas to their calls, and only being able to return the first 32 bytes of return data. These issues may reduce the ways in which users can interact with the proxy contract.

5.6 Various Event Issues

Design | Low | Version 1 | Code Partially Corrected | Acknowledged

Some event parameters are not indexed, even though they might be useful for filtering events.

- *AccountFactory*: The *AccountCreated* event does not have an indexed `proxy` address.
- *AccountGuard*: The *OwnershipTransfered* event does not have an indexed `newOwner` address.

The *OwnershipTransfered* event is very similarly named to the *OwnershipTransferred* event which it inherits from OpenZeppelin's *Ownable* contract. Consider renaming it so that these two events cannot be confused.

Lastly, the `vaultId` parameter of the *AccountCreated* event is confusingly named, as there is no other place in the code which refers to vaults. Renaming this to a more suitable description of the parameter should be considered.

Code partially corrected

The indexed keyword has been added to the addresses in the mentioned events.

OwnershipTransfered has been renamed to ProxyOwnershipTransfered.

Acknowledged

The `vaultId` parameter name is unchanged. The *ProxyOwnershipTransfered* event has a misspelling, it should be *ProxyOwnershipTransferred*.

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

- ImmutableProxy Is Unnecessary Code Corrected

Medium-Severity Findings	1

- New Owner Cannot Permit Code Corrected

Low-Severity Findings	12

- AccountGuard Variable Could Be Immutable Code Corrected
- Checking Conditions in AccountImplementation Code Corrected
- Execute Does Not Forward Revert Reason Code Corrected
- Floating Compiler and Dependency Versions Code Corrected
- Function Visibility Can Save Gas Code Corrected
- Incorrect Error Messages Code Corrected
- Increment Can Be Done in Event to Save Gas Code Corrected
- Missing Sanity Checks Code Corrected
- Redundant Initializations Code Corrected
- Separate Whitelists Code Corrected
- Unnecessary Checks Code Corrected
- Unused Imports Code Corrected

6.1 ImmutableProxy Is Unnecessary

Design High Version 1 Code Corrected

The ImmutableProxy is designed as a small proxy contract which delegatecalls into a fixed implementation address. Except for the `implementation()` getter, its functionality is identical to the minimal proxy contract described in EIP1167.

OpenZeppelin's *Clones* library, which is used to deploy clones of the *ImmutableProxy* contract, deploys clones by using the contract described in EIP1167 - so it simply deploys a contract which delegatecalls the target contract. All in all, this means that we have a minimal proxy, which calls the *ImmutableProxy*, which then calls the *AccountImplementation*. This double proxy setup is unnecessary and wastes gas.

Instead, it would be much simpler and cheaper to directly clone the *AccountImplementation* and discard the *ImmutableProxy* contract. The minimal proxy contract is hand-written bytecode which is designed to be as cheap as possible both to deploy and execute.

Code corrected

The *ImmutableProxy* contract was removed. Instead, the *Clones* library is used to directly clone the *AccountImplementation*.

6.2 New Owner Cannot Permit

Correctness Medium Version 1 Code Corrected

In *AccountGuard*, when an owner is initially set by the factory they also receive an entry in the `allowed[caller][target]` mapping. However, when `changeOwner()` is called, the new owner only receives an entry in the `owners` mapping but is not added to the `allowed` mapping. Hence, the new owner cannot call `permit` on the proxy they now own.

Note that not even someone else who is allowed to permit on the proxy can admit the owner, as permitting the owner will revert with `"account-guard/cant-deny-owner"`. The only way to allow the new owner would be to transfer ownership to someone else who is already allowed to permit.

The owner would still be able to make transactions from the proxy, as `canCall()` always returns true for the owner, even if they are not `allowed`.

Code corrected

`changeOwner` now adds the new owner to the allowed addresses.

6.3 AccountGuard Variable Could Be Immutable

Design Low Version 1 Code Corrected

In *AccountFactory*, the `AccountGuard` variable is set in the constructor and then never changed. It could be made immutable to be more gas-efficient.

Code corrected

The variable has been made immutable.

6.4 Checking Conditions in AccountImplementation

Design Low Version 1 Code Corrected

When calling `send()` or `execute()` in the *AccountImplementation*, the following checks are performed:

1. First, the `auth` modifier checks that `guard.canCall(address(this), msg.sender)` returns true.
2. Next, it is checked that `_target != address(0)`.
3. Lastly, the condition `guard.isWhitelisted(_target)` is checked.

Here, the *AccountGuard* contract is called twice to check whether the call should be allowed. These calls could be combined into a single cross-contract call to reduce gas costs.

Assuming that `address(0)` is never whitelisted, the second check is redundant.

Note also that the same conditions are checked in both cases, hence they could be included into the `auth` modifier in order to reduce code duplication.

Code corrected

The calls to *AccountGuard* were combined into a single call of the new `canCallAndWhitelisted` function and moved into the `auth` modifier. The `_target != address(0)` check was removed.

6.5 Execute Does Not Forward Revert Reason

Design | Low | Version 1 | Code Corrected

In *AccountImplementation*, `execute()` reverts with `(0,0)` if the delegatecall fails. It does not revert with the revert reason from the delegatecall.

It would be easier for users to debug their reverted transactions if the revert reason contained information from the delegatecall.

Code corrected

The return data is now copied and returned. If the delegatecall fails, the revert reason will be returned.

6.6 Floating Compiler and Dependency Versions

Design | Low | Version 1 | Code Corrected

Oasis uses the floating pragma `solidity ^0.8.9`. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively (https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md).

In `hardhat.config.ts`, the compiler version `0.8.17` is specified.

The versions of the contract libraries in `package.json` are also not fixed. In particular:

```
"@openzeppelin/contracts": "^4.7.3"
```

The caret `^version` will accept all future minor and patch versions while fixing the major version. With new versions being pushed to the dependency registry, the compiled smart contracts can change. This may lead to incompatibilities with older compiled contracts. If the imported and parent contracts change the storage slot order or change the parameter order, the child contracts might have different storage slots or different interfaces due to inheritance.

In addition, this can lead to issues when trying to recreate the exact bytecode.

Code corrected

The Solidity compiler version has been specified as exactly `0.8.17` for all contracts in scope.

The OpenZeppelin version has been specified as exactly version `4.7.3` in `package.json`.

6.7 Function Visibility Can Save Gas

Design | Low | Version 1 | Code Corrected

Reducing a function's visibility from public to external saves gas. The following functions could be defined as external:

- `AccountFactory.createAccount()`
- `AccountGuard.isWhitelisted()`
- `AccountGuard.setWhitelist()`
- `AccountGuard.initializeFactory()`
- `AccountImplementation.send()`
- `AccountImplementation.execute()`
- `MakerMigrationsActions.migrateMaker()`
- `MakerMigrationsActions.migrateAdditionalVaults()`
- `McdView.getRatio()`
- `McdView.getMakerProxy()`

Additionally, the `self` storage variable in `MakerMigrationsActions` should not be public, as it only contains the address of the contract itself. Hence, making it private or internal reduces bytecode size as the getter is unnecessary.

Code corrected

The mentioned functions in *AccountGuard*, *AccountFactory*, and *AccountImplementation* have been changed to external. The visibility of the `self` storage variable in `MakerMigrationsActions` was changed to private.

6.8 Incorrect Error Messages

Design | Low | Version 1 | Code Corrected

The `auth` modifier in *AccountImplementation* and the `permit()` function in *AccountGuard* both revert with the error message `"account-guard/not-owner"`.

However, the owner is actually not the only user that is checked for. In both cases, it is checked whether the `msg.sender` is the owner OR has been permitted. The error message makes it seem like only the owner is allowed to use these functions.

Also, `migrateAdditionalVaults` in *MakerMigrationsActions* reverts with the message "factory/already-migrated" in case it is called from an address that has not yet migrated. This error message states the opposite of what actually happened.

Code corrected

The `permit()` function and `authandWhitelisted` modifier now revert with `"account-guard/no-permit"`. The revert message in `migrateAdditionalVaults` was also changed.

6.9 Increment Can Be Done in Event to Save Gas

Design Low Version 1 Code Corrected

In *AccountFactory*, `createAccount()` increments the `accountGlobalCounter` variable. Later, it is emitted in the `AccountCreated` event. This means it must be loaded from storage twice.

The `accountGlobalCounter` could be cached or incremented in the event emission, which would save one SLOAD operation and be more gas-efficient.

Code corrected

The value of the counter is now cached after being incremented.

6.10 Missing Sanity Checks

Design Low Version 1 Code Corrected

Sanity checks on values are missing in several places. This makes it more likely for incorrect values to accidentally be set.

- In `AccountFactory.createAccount(address user)`, it is possible for `user` to be `address(0)`.
- In `AccountGuard.changeOwner(address newOwner, address target)`, it is possible for `newOwner` to be `address(0)`.
- In the *AccountImplementation* constructor, `_guard` could be `address(0)`.
- In the *MakerMigrationsActions* constructor, `_serviceRegistry` could be `address(0)`.

Code corrected

Sanity checks have been added to all mentioned functions.

6.11 Redundant Initializations

Design Low Version 1 Code Corrected

Some storage variables are initialized to their default values, which is redundant and wastes gas.

- In the *AccountFactory* constructor, `accountsGlobalCounter` is initialized to `0`.
- In *AccountGuard*, `factory` is initialized to `address(0)`.

Code corrected

Both redundant initializations were removed.

6.12 Separate Whitelists

Design Low Version 1 Code Corrected

The *AccountImplementation* allows calls and delegatecalls to any whitelisted contracts. Currently, there is just one whitelist for both cases. However, there may be some contracts that are safe to call but not delegatecall or vice versa.

Code corrected

The `whitelisted` mapping of the *AccountGuard* now contains a `(bool => bool)` mapping. This extra bool lookup parameter allows specifying whether a specific address is whitelisted for regular calls, delegatecalls, or both.

6.13 Unnecessary Checks

Design | Low | Version 1 | Code Corrected

`permit()` in *AccountGuard* has the following check:

```
if (msg.sender == factory && allowance)
```

The *AccountFactory* only has one function that calls `permit()`. It always calls it with `allowance` set to `true`. Hence, just checking for `msg.sender == factory` is equivalent, as allowance is always `true` if this holds (unless *AccountFactory* is changed in the future).

There are also unnecessary checks in `send()` and `execute()` in *AccountImplementation*, which are described in Checking Conditions in AccountImplementation.

These checks can be removed to save gas.

Code corrected

The check for `allowance` has been removed.

6.14 Unused Imports

Design | Low | Version 1 | Code Corrected

There are several imported files which are never used:

In *AccountFactory*:

- `"./interfaces/IProxyRegistry.sol"`
- `"./interfaces/ManagerLike.sol"`
- `"./interfaces/IServiceRegistry.sol"`
- `"./utils/Constants.sol"` (AccountFactory inherits from it but no values are used)

In *AccountGuard*:

- `"@openzeppelin/contracts/proxy/Proxy.sol"`

Code corrected

The unused imports were removed.

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 Permissions Stay When Changing Owner

Note Version 1

When the `changeOwner()` function of *AccountGuard* is called, the permissions of the old owner are revoked. However, the permissions of all other addresses are unaffected. Any addresses that were permitted by the old owner will still have permissions unless the new owner calls `permit()` to remove them.

Anyone receiving ownership over a proxy should ensure they trust all permitted addresses before using it.

7.2 Proxy Storage Layout

Note Version 1

The *AccountImplementation* allows delegatecalls into arbitrary whitelisted contracts. Such a contract could read/write to storage. If two different contracts are delegatecalled that both use the same storage slots, they may conflict with each other and read data written by the other contract.

The *AccountGuard* owner should keep this in mind when whitelisting contracts. They should either not whitelist conflicting contracts or explicitly warn users not to use multiple contracts that use the same storage from the same proxy.

7.3 `send` Function Name Is Confusing

Note Version 1

The `send` function of *AccountImplementation* has a lot more functionality than just sending ETH. It can be used to call arbitrary functions on any whitelisted contract.

This is not obvious from the function name and could be confusing to users.