PUBLIC

Code Assessment

of the Argent Accounts v0.5.0 Smart Contracts

April 15, 2025

Produced for

by

Contents

1 Executive Summary

Dear Argent Team,

Thank you for trusting us to help Argent with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Argent Accounts v0.5.0 according to Scope to support you in forming an opinion on their security risks.

Argent implements Argent Account and Multisig Account, a set of smart contracts built on top of Starknet's Account Abstraction. This review covers version 0.5.0, which introduces support for multiple owners and guardians in the Argent Account. Compared to version 0.4.0 (and 0.2.0 for the Multisig), the core functionality remains largely unchanged, with most updates focused on internal refactoring and enhanced support for the WebAuthn signer.

The most critical subjects covered in our audit are functional correctness, access control and the upgrade path from previous versions. Security regarding all the aforementioned subjects is high.

The general subjects covered are code complexity, specifications, and trustworthiness. Overall, the code is of high quality, well tested and documented.

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	0
Medium-Severity Findings	0
Low-Severity Findings	9
• Code Corrected	8
• Specification Changed	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 Argent Accounts v0.5.0 repository based on the documentation files.

The following files are in scope of this review:

```
src:
    account.cairo
    introspection.cairo
    lib.cairo
    offchain_message.cairo
    recovery.cairo
    upgrade.cairo
    linked_set:
        linked_set.cairo
        linked_set_with_head.cairo
    multiowner_account:
        account_interface.cairo
        argent_account.cairo
        events.cairo
        guardian_manager.cairo
        owner_alive.cairo
        owner_manager.cairo
        recovery.cairo
        signer_storage_linked_set.cairo
        upgrade_migration.cairo
    multisig_account:
        external_recovery.cairo
        multisig_account.cairo
        signer_manager.cairo
        upgrade_migration.cairo
    outside_execution:
        outside_execution.cairo
        outside_execution_hash.cairo
    session:
        session.cairo
        session_hash.cairo
    signer:
        eip191.cairo
        signer_signature.cairo
        webauthn.cairo
    utils:
        array_ext.cairo
        asserts.cairo
        bytes.cairo
        calls.cairo
```

```
        hashing.cairo
        serialization.cairo
        transaction_version.cairo
```

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	03 Mar 2025	78e80fa1f9b3587c19a6c60a51564df56a907efa	Initial Version
2	07 Apr 2025	908731268bf5ee64ff8da68136a96d138c4b0d71	Version with fixes
3	15 Apr 2025	f091cd3535d9630f1bbbe01caaf398bddd119dbc	Contract downsized

For the cairo smart contracts, the compiler version `2.10.1` was chosen. At the time of this review (March 2025) StarkNet `v0.13.3` was live on mainnet. This review cannot account for future changes or possible bugs in StarkNet or its libraries.

2.1.1 Excluded from scope

Any file not listed above is out of the scope of this review. In particular, Starknet core and third-party libraries, like `alexandria` or `openzeppelin`, are out of the scope of this review and are assumed to work correctly.

Tests and deployment scripts are excluded from the scope.

The security of WebAuthn standard is out of scope. Attack vectors that exploit Passkey weaknesses to receive valid signatures are not considered in this review.

The review was conducted with the assumption that:

- A transaction with estimation flag set will not incur any state change.
- No external calls to other contracts can be made in the context of `__validate__`.

A complete review of a previous version of the Argent Account is available at https://www.chainsecurity.com/security-audit/argent-account.

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.

This review extends the review of the `Argent Account v0.4.0`, which can be found at https://www.chainsecurity.com/security-audit/argent-account.

Argent offers Argent Account, a SNIP-6 compliant account abstraction contract on StarkNet with enhanced functionalities and security settings (i.e. wallet guardian protection, sessions and outside executions) and supports for signature verification with extended standards (i.e. STARK curve, Secp256k1, Secp256r1, WebAuthn).

The system consists of two account contracts: the multi-owner Argent account and the multisig account.

2.2.1 Multi-owner Argent Account

The Multi-owner Argent Account, or Argent account, is the core smart contract in the repository that implements the required functionalities for account abstraction on StarkNet.

Each Argent account has at least one, and up to 32 owners, which have the ultimate control of the account. Optionally, guardians, again up to 32, can be enabled the owner(s) to cosign transactions for enhanced security.

2.2.1.1 Extended Signature Support

The owners of an account and the guardians represent different roles in a contract and are distinguished by their public keys. Any transaction initiated by the account should be signed by one of the owners, and one of the guardians, if any is set. Argent account supports multiple signature standards.

When registered, owners and guardians can have the option to sign with one of the following methods:

- STARK curve
- Secp256k1 curve
- EIP191 (using Secp256k1 curve)
- Secp256r1 curve
- WebAuthn (using Secp256r1 curve)

The contract always stores the information about the public key of a signer as a `felt252` onchain:

- For signers of STARK curve, the public key will be directly stored as a `felt252`.
- For signers of Secp256k1 and EIP191, the ethereum address of the corresponding public key is stored as a `felt252`.
- For signers of Secp256r1, the Poseidon hash of the signature type and the public key will be stored as a `felt252`.
- For signers of WebAuthn type, the Poseidon hash of the signature type, the `origin`, the `rp_id_hash` and the public key will be stored as a `felt252`.

WebAuthn

WebAuthn is designed to authenticate users to a Relying Party (RP). Upon registration, the authenticator will generate a public and private key pair for a specific RP ID. Upon authentication, this public key will be used to sign a challenge from the RP. WebAuthn signatures use the curve Secp256r1.

Argent account overloads the authenticator's WebAuthn feature to support signing a transaction in the form of an authentication. In case the account owner is a WebAuthn signer, it consists of an origin, a `rp_id_hash`, and a public key:

```
struct WebauthnSigner {
    origin: Span<u8>,
    rp_id_hash: NonZero<u256>,
    pubkey: NonZero<u256>
}
```

On authentication, the user agent (browser, mobile app, or OS) sends the hash of the client data json together with the RP ID to the authenticator. The client data json must contain the type of this request, the challenge from the RP in base64 url safe encoding, and the origin. The client data hash together with the authenticator data is expected to be signed by the authenticator with the private key for the specific RP ID and generate the struct `WebauthnSignature`.

To sign a transaction with authenticator, the `challenge` is set to the hash of the transaction that will be executed by the account. Each account in Starknet has a nonce which is enforced to be unique for each transaction, hence the `challenge` is different for each transaction. A struct `WebauthnSignature` will be passed as transaction signature and will be validated against the stored `WebauthnSigner` GUID:

1. Verifying Client Data Json: The request type is hardcoded to `webauthn.get`, challenge is set to be the hash of the transaction, and origin is extracted from the `SignerType` recovered from the

signature of the transaction. It is fine to use user provided origin as the signer will be checked to be a valid signer for the account.

2. Verifying Authenticator Data: The RP ID is extracted from the `SignerType` recovered from the signature of the transaction. The `user-present` and `user-verified` bits must be set to true. It is fine to use user provided RP ID hash as the signer will be checked to be a valid signer for the account.
3. Verifying WebAuthn Signature: The client data json hash and authenticator data should be signed by the correct key.

2.2.1.2 Sessions

Argent account features sessions to enhance user experience when frequently interacting with dApps (for instance onchain games). Users can authorize a session between their Argent account and a dApp, which delegates the privilege of initiating certain transactions to a public key from the dApp. When using a session, the dApp must also collect a signature from one of the guardians of the Argent account. Sessions can only be used when at least one guardian is enabled. Each session has an expiration timestamp and is restricted to predefined contract addresses and selectors (represented as leaves in a merkle tree, whose root resides in the signed session struct). A session will be signed off-chain and users can revoke a session (via `revoke_session()`) onchain given a session hash.

```
struct Session {
    expires_at: u64,
    allowed_methods_root: felt252,
    metadata_hash: felt252,
    session_key_guid: felt252,
}
```

Although the target contracts and methods are restricted in a session, the frequency of interactions is not restricted. Hence, in theory, a colluding dApp with an active session key and a guardian can drain the gas token in an argent account by triggering numerous transactions.

2.2.1.3 Transaction Validation

Before executing a `invoke` transaction, the sequencer validates (via `__validate__()`) the submitted transaction . Only after a successful validation, the transaction will be executed (`__execute__()`).

The function `__validate__()` can only be invoked by the protocol (with `0` as a caller address). The `invoke` transaction version is restricted to `V1` and `V3` with their simulation bits on or off. The paymaster data should be empty.

In case the transaction is triggered by a session (the signature starts with the magic value `session-token`), the following must hold:

1. Session calls cannot trigger functions in the account itself.
2. The session is not revoked or expired. Note the expiration check is inaccurate because the timestamp will be rounded down to the nearest hour in the context of `__validate__()`.
3. The session struct was authorized by one of the owners plus one of the guardians.
4. The session transaction is signed by the session key and the guardian who signed the session struct. A session can be cached in the contract to avoid verifying the owner and guardian signatures again and again, in this case the guardians must match only when the session is cached. When the cached session is reused, the bytes corresponding to the signatures over the session can be ignored as the signatures have been verified already.
5. For each call in the multicall, a merkle proof should be provided and validated against the `allowed_methods_root` in the session struct.

Otherwise, if the transaction is triggered by an owner and/or a guardian, the following must hold:

1. Multicall cannot contain self as a target address (for admin functions) if more than one call is done.
2. Single call to trigger or finalize escape on the owner or guardian has independent logic to verify the single party signature and the gas limit, and will return early.
3. Calls to `execute_after_upgrade()` and `perform_upgrade()` are forbidden.
4. The signature of a current owner (plus a guardian if enabled) is valid.

Sessions can be cached in the contract to avoid verifying the owner and guardian signatures again and again. If a session is cached, only the signatures of the guardian and the dApp will be checked. An owner can revoke a session with `revoke_session()`.

2.2.1.4 Transaction Execution

`__execute__()` can only be invoked by the protocol (with `0` as a caller address). If the transaction is initiated by a session, the session expiration will be checked again (formerly checked in `__validate__()` with an inaccurate timestamp). Eventually the multicall will be executed (`execute_multicall()`), which will revert the whole transaction if any of the call reverts.

2.2.1.5 Outside Execution

In addition to `__validate__()` and `__execute__()`, the execution of multicall from the account can also be triggered via outside execution (`execute_from_outside()` or `execute_from_outside_v2()`) directly. An `OutsideExecution` struct represents an one-time execution of an authorized multicall by a restricted caller within a certain period.

```
struct OutsideExecution {
    caller: ContractAddress,
    nonce: felt252,
    execute_after: u64,
    execute_before: u64,
    calls: Span<Call>
}
```

Upon a call to outside execution, a valid signature is required for the outside execution hash, which could be signed by the owner (plus guardian or backup guardian) or a valid session public key.

2.2.1.6 Escaping owner or guardian

Argent account implements an escaping feature:

- A guardian can help to change the owners key set in case the owners' keys are lost.
- The owners can escape from a compromised or malicious guardian.

Each escape is subject to a delay and an expiration window (`escape_security_period` defaulted to 7 days).

The owner has higher privilege than the guardian in Argent account: an escape from the owner (`trigger_escape_owner()`) can only be initiated if there is no ongoing escape for the guardian, whereas an escape from the guardian (`trigger_escape_guardian()`) can always be initiated by the owner, which will overwrite any existing escape.

After a delay period, the escape has a status `Ready` and can be finalized via `escape_owner()` or `escape_guardian()`. In case an escape is not finalized within a limited time window, the escape expires. The owners should set a reasonable delay period and actively monitor the events for malicious escaping attempts from a malicious guardian.

An escape attempt can be canceled via `cancel_escape()` with valid signatures from an owner and a guardian. An owner can also cancel an owner escape by initiating an escape from the guardian.

In order to avoid the contract to be drained from its gas token from a compromised owner or a malicious guardian, the timestamp of the last escape trigger and escape execution are stored during `__validate__()` and checked that a minimum period of time has elapsed between two calls. The timestamps are reset after a successful execution.

2.2.1.7 Other External Functions

The escape security period can be updated via `set_escape_security_period()`. The owners and guardians can be changed. All these functions can only be called from the account itself, and will reset the existing `escape` and `escape_timestamp`:

- `change_owners()`: change the current owners set to a new owners set. If the signer of the transaction is removed of the set in the process, one of the remaining owner must provide a proof that an owner can still sign in the form of an `OwnerAliveSignature`.
- `change_guardians()`: change the current guardians set to a new guardians set or simply remove all guardians.
- `cancel_escape()`: cancels an escape in any state unless status is already `None`.

2.2.1.8 Upgradeability

The version of the argent account in this review is `v0.5.0`. Upgradeability is implemented to support two scenarios:

1. An account from an older version can be upgraded to version `v0.5.0`.
2. The current argent account could be upgraded to a future version.

An upgrade can be triggered via the external function `upgrade()` which can be called from the account itself. Only some versions can be directly upgraded to `v0.5.0`:

- `v0.2.3.0` and `v0.2.3.1` can be directly upgraded only if some `data` is passed along during the `upgrade()` call.
- `v0.3.0` and `v0.3.1`
- `v0.4.0`

Upgrade from versions v0.2.3.X, v0.3.X, and v0.4.0 to the current version:

1. Ensures the new implementation supports the account interface via a library call.
2. For `v0.3.X`, initiates a system call to replace the class hash to the new implementation.
3. Initiates a library call to the new implementation's entrypoint `execute_after_upgrade()`, `perform_upgrade()` for `v0.4.0`:

    - Any ongoing escape will be cancelled due to the change of `_escape` storage layout in the new implementation.
    - `guardian_escape_attempts` and `owner_escape_attempts` will be cleared as they are no longer used in the new implementation. This is done for `v0.2.3.X` and `v0.3.X` but will impact only the latter version.
    - The existing owner will be checked and the respective events are emitted. The Starknet owner will be migrated in the new linked list with head data structure and the old owner field is reset. In the case of `v0.4.0`, if the signer is not a Starknet signer, the map will be searched until the correct signer type is found, it is then migrated to the new linked list with head data structure and the mapping is reset.

- Any existing guardian and backup guardian will be checked and the respective events are emitted. The Starknet guardian and backup guardian will be migrated in the new linked list with head data structure and the old fields are reset. In the case of `v0.4.0`, if the guardian or backup guardian is not a Starknet signer, the map will be searched until the correct signer type is found, they are then migrated to the new linked list with head data structure and the mappings are reset.
- In case a multicall (except calling self) is batched with the upgrade, it will be executed.

In the case of an upgrade from `v0.2.3.X` to the current version, the step 2. is skipped. The new class hash is cached in `_implementation` first and then, if some calldata is passed, used by `replace_class_syscall()` invoked in `execute_after_upgrade()`. The `_implementation` is then reset to 0. In the case where no calldata is passed to the `upgrade()` call, users must call `recovery_from_legacy_upgrade()` on the new implementation to finalize the upgrade.

In the case of an upgrade from `v0.4.0`, the step 2. is skipped. The system call to replace the class hash is done int the `perform_upgrade()` library call.

Upgrade from versions v0.2.0, v0.2.1, v0.2.2 to the current version:

1. Ensures the new implementation supports the account interface via a library call.
2. Sets the new implementation's class hash to the account's `_implementation`.
3. Users must call `recovery_from_legacy_upgrade()` on the new implementation to finalize the upgrade. This will perform the same steps as the previous upgrade path as for `v0.2.3.X` and `v0.3.X`.

Upgrade from the current version to a future version:

1. Ensures the new implementation supports the account interface via a library call.
2. Initiates another library call to the new implementation's `perform_upgrade()` entrypoint to execute upgrade-specific logic.
3. The upgrade is completed by a system call to replace class hash via another library call to the new implementation.

2.2.2 Multisig Account

The Argent Multisig implements a typical n-of-m multisig. A multisig account is controlled by a set of signers (up to 32 signers) and specifies a threshold of signatures that should be provided to authorize a transaction. Multisig account implements the same features as the Argent account, except sessions which are not supported.

2.2.2.1 Signer List

The multisig supports various signer types as Argent account: `StarknetSigner`, `Secp256k1Signer`, `Secp256r1Signer`, `Eip191Signer`, and `WebauthnSigner`. The guid of the signers will be stored in a linked list. The following functions have been provided to manage the signers and threshold:

- `change_threshold()`: updates the threshold to a new value. It is ensured that the threshold cannot be 0 and cannot exceed the amount of signers.
- `add_signers()`: adds a list of new signers to the list and updates the threshold. It is ensured there is no duplicated signers, and the threshold is checked as `change_threshold()`.
- `remove_signers()`: removes a list of existing signers and updates the threshold.
- `replace_signer()`: replaces an existing signer to a new signer. It is ensured the new signer does not exist in the current signer list at the beginning of the call.

2.2.2.2 Transaction Validation and Execution

`__validate__()` can only be invoked by the protocol (with `0` as a caller address). The `invoke` transaction version is restricted to `V1` and `V3` with their simulation bits on or off. The paymaster data and the account deployment data are ensured to be empty.

If there is only one call, it is forbidden to call `execute_after_upgrade()` or `perform_upgrade()` on the account itself. If it is a multicall, it cannot contain self as a target address (for admin functions). The transaction hash should be signed by exactly `threshold` signers. The signatures are verified individually (or skipped if the estimation bit is on), while redundant signatures from the same signer are forbidden by enforcing a strictly ascending signer guid order.

After a successful validation, the function `__execute__()` is executed by the protocol.

Declare transaction is not supported by the multisig. For the deployment of a multisig, the transaction version is restricted to `V1` and `V3`. Differently from `invoke` transaction, only one signature from the signers is required for account deployment.

Outside execution is also supported in multisig. An outside execution works in the same way as in Argent Account, which requires a valid outside execution struct signed by `threshold` amount of signers. In addition, the calls are subject to the same checks in `__validate__()`.

2.2.2.3 External Recovery

No execution from the multisig itself is possible in case some keys are lost hence the threshold cannot be reached. An external recovery feature is provided to resolve this scenario.

By default, external recovery is disabled. Signers can trigger a call to `toggle_escape()` on the multisig to enable / update / or disable it.

- To enable or update it, it is ensured there is no ongoing escape. A non-zero guardian address need to be provided, which is a privileged role to trigger an escape (`trigger_escape()`). In addition, a security period and an expiry period should be set to non-zero.
- To disable it, the guardian and both periods should be reset to 0.

The guardian can trigger an escape by directly calling `trigger_escape()`. The escape call option is restricted to `replace_signer()`, `change_threshold()`, `add_signers()`, or `remove_signers()`. The escape will always cancel any ongoing escape.

Once `security_period` has elapsed after a triggered escape, anyone can finalize the escape by calling `execute_escape()` with the same call (the hash of the call will be ensured to match the escape triggered). If it is not finalized within another `expiry_period`, the escape will expire and cannot be finalized anymore.

The signers of the multisig can cancel the escape at any time by providing signatures above `threshold`. The `security_period` should be chosen carefully and take into consideration the time required to collect the signatures.

2.2.2.4 Upgradeability

The version of the multisig account in this review is `v0.5.0`. Upgradeability is implemented to support two scenarios:

1. A multisig account from an older version can be upgraded to version `v0.5.0`.
2. The current multisig account could be upgraded to a future version.

All the previous versions (`v0.1.0, v0.1.1, v0.2.0`) can directly be upgraded to `v0.5.0`.

Upgrade from versions 0.1.0, 0.1.1 to the current version:

1. Ensures the new implementation supports the account interface via a library call.
2. Initiates a system call to replace the class hash to the new implementation.

3. Initiates a library call to the new implementation's entrypoint `execute_after_upgrade()`:
    - The number of pubkeys is checked against the threshold.
    - The LegacyMap is migrated to the new linked list by adding an end marker and the pubkeys are migrated to guids.

Upgrade from versions 0.2.0 to the current version:

1. Ensures the new implementation supports the account interface via a library call.
2. Initiates a library call to the new implementation's entrypoint `perform_upgrade()`:
    - The version is checked for the upgrade to be only a forward update.
    - The number of pubkeys is checked against the threshold.
    - The LegacyMap is migrated to the new linked list by adding an end marker

Upgrade from the current version to a future version:

1. Ensures the new implementation supports the account interface via a library call.
2. Initiates another library call to the new implementation's `perform_upgrade()` entrypoint to execute upgrade-specific logic.
3. The upgrade is completed by a system call to replace class hash via another library call to the new implementation.

2.2.3 Roles and Trust Model

2.2.3.1 Multi-owner Argent Account

The owners of the Argent account have the ultimate controls of the account and the funds held by it. Thus, it is assumed that the owners are fully trusted and sign only legit transactions. Owners should actively monitor the escape events from their Argent account and choose a security period that allows them to cancel malicious escape attempts. Otherwise, a malicious guardian can get control of an account by escaping the owner. We assume the owner's key will not be leaked and lost at the same time.

We assume the holder of a session key and a guardian do not collude (malicious at the same time). Otherwise, they can drain the gas token from the argent account by initiating and signing numerous transactions together, or misuse any permission provided by the owners.

The guardians are partially trusted. They should facilitate the owners operations, but the owners can remove them or escape from them if they fail to fulfill their obligations. In addition, the following assumptions are made:

- The guardians are trusted to assist the owners on account recovery when all of the owners' keys are lost, otherwise the account cannot be recovered, or malicious guardian can take over the account.
- The guardians are trusted to serve as an additional layer of security and protect the account if one of the owners' key is leaked. Otherwise, an attacker can drain all funds from the account.

2.2.3.2 Multisig Account

The following assumptions are made for the multisig:

- No more than `threshold` keys can be leaked / compromised at the same time, otherwise the adversary controls the account.
- If set, the guardian is partially trusted to assist on account recovery.

- Signers should actively monitor the escape events from their account and choose a security period that allows them to cancel malicious escape attempts. Otherwise, a malicious guardian can get control of the multisig by replacing the signers.
- The signer who deploys the multisig is trusted, otherwise it can burn the gas tokens by deploying the account with a high gas tips.

The following assumptions are made on the sequencer in this review:

- A transaction with estimation flag on will not trigger any state changes. Otherwise one can leverage the flag to bypass signature verification and drain any account.
- No external call to other contracts can be made within the `__validate__()` context.

2.2.3.3 WebAuthn

The following assumptions are made on WebAuthn in this review:

- The authenticator ensures that key pairs created for a Relying Party can only be used in operations requested by the same RP ID.
- The user agent (e.g. browser and mobile app) is fully trusted, the client data json prepared by the user agent should only contain the legitimate values and there is no duplicated, nested, or maliciously injected data.
- The relying party is trusted to request signatures from the authenticator only for transactions intended by the legit owner.

In addition, the new implementation that users choose to upgrade is fully trusted to never misbehave during the library calls of `supports_interface()` and `perform_upgrade()`.

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 Open 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	0
Low-Severity Findings	0

6 Resolved Findings

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

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	0
Low-Severity Findings	9

- Backup Guardian Check Performed Only for <0.4.0 Migrations Code Corrected
- Dead Code Code Corrected
- Inaccurate Variable Name Code Corrected
- Inconsistent Event Order on Replacing Signer Code Corrected
- Leftover current nonce in Storage Code Corrected
- Malleable Starknet Signatures Code Corrected
- Reverts When Escaping to Current Signer Code Corrected
- Unneeded Mutability Code Corrected
- Wrong Specification Specification Changed

Informational Findings	1

- Outdated Libraries Code Corrected

6.1 Backup Guardian Check Performed Only for <0.4.0 Migrations

Design Low Version 1 Code Corrected

CS-AGAC050-009

`migrate_from_before_0_4_0()` asserts that a `backup_guardian` can only be set if a `guardian` is set. Since this check is only in `migrate_from_before_0_4_0()`, it's not explicitly enforced when upgrading from `0.4.0`. All earlier versions of ArgentAccount enforce that setting a backup guardian requires a guardian set hence this understood to be a sanity check during migration. Without explicit documentation of the check's purpose, omitting it during migrations from `0.4.0` appears inconsistent.

Code corrected:

The same check has been added to `migrate_from_0_4_0()`. Since `migrate_from_before_0_4_0()` calls `migrate_from_0_4_0()`, the check is now performed twice when migrating from prior to `0.4.0`.

6.2 Dead Code

Design Low Version 1 Code Corrected

CS-AGAC050-010

For the sake of code clarity and maintainability, dead code should be removed. Dead code is code that is not executed in any circumstances. It can be a leftover from previous development stages, or it can be a result of code refactoring. Dead code can be a source of confusion for developers and can lead to bugs and other issues. It is important to remove dead code to keep the codebase clean and easy to understand.

1. The function `migrate_owner()` of the private trait of the `upgrade_migration` component is never called.

Code corrected:

The function `migrate_owner()` has been removed from the codebase.

6.3 Inaccurate Variable Name

Design Low Version 1 Code Corrected

CS-AGAC050-007

In `LinkedSetReadImpl::get_all()`, the array `all_hashes` is storing items of type `T` which are not necessarily hashes.

Code corrected:

The variable was renamed to `all_items`.

6.4 Inconsistent Event Order on Replacing Signer

Design Low Version 1 Code Corrected

CS-AGAC050-006

In the edge case where `replace_signer()` is called with the same signer to be added and removed, the emitted events will first be a `OwnerRemovedGuid` event, followed by a `OwnerAddedGuid` event. This does not allow an observer to correctly reconstruct the contract's storage from the events, as they are emitted in the wrong order.

Code corrected:

The function implements an additional check to ensure that the signer being added is not the same as the signer being removed.

6.5 Leftover `current nonce` in Storage

Design Low Version 1 Code Corrected

CS-AGAC050-012

Prior to StarkNet v0.10.0, account implementations were responsible for managing their nonces independently. Argent accounts up to version 0.2.2 stored the `_current_nonce` within their contract state, but this was discontinued starting from version 0.2.3 when nonce management transitioned to protocol-level handling. However, during migration from Argent account versions 0.2.x to 0.5.0, the `_current_nonce` in storage is not reset.

Code corrected:

During migration from versions prior to `v0.5.0`, the `_current_nonce` value is now explicitly reset to `0`. This has no effect for accounts that were initially deployed with a version using protocol-managed nonces, but is necessary for accounts that started from earlier versions (e.g. `0.2.x`) and may still have a non-zero nonce stored locally.

6.6 Malleable Starknet Signatures

Correctness | Low | Version 1 | Code Corrected

CS-AGAC050-011

`is_valid_starknet_signature()` does not check that `s` and `r` are smaller than the curve order before calling `check_ecdsa_signature()`. The library specifies that this function validates that `s` and `r` are not 0 or equal to the curve order, but does not check that `r, s < stark_curve::ORDER`, which should be checked by the caller. As consequence signatures are more malleable.

Code corrected:

`is_valid_starknet_signature()` now checks that `s` and `r` are smaller than the curve order before calling `check_ecdsa_signature()`.

Argent states:

```
We don't think it poses a risk to the account as the Starknet signatures are
already malleable and this issue only allows for additional malleability for
some signatures that happen to be in an extremely narrow band. This was
communicated to Starkware, and they agree with the assessment and are planning
to address this issue in the core library. We added checks on the account to
ensure it doesn't accept these signatures for extra precaution.
```

6.7 Reverts When Escaping to Current Signer

Design | Low | Version 1 | Code Corrected

CS-AGAC050-008

`trigger_escape_owner()` and `trigger_escape_guardian()` allow specifying a new signer that is already in the current set of owners or guardians respectively. However, such escapes will always revert during `complete_owner_escape()` or `complete_guardian_escape()`, since inserting a signer already present into the linked_set will lead to a revert.

This behavior is likely unintended, as the code appears to try handling the case where the new signer is already present.

This is illustrated in the guardian escape flow (the same applies to owner escape):

`complete_guardian_escape()` builds the list of guardians to remove, and separately specifies the new guardian to add:

```
for guardian_to_remove_guid in self.guardians_storage.get_all_hashes() {
    if guardian_to_remove_guid != new_guardian_guid {
        guardian_guids_to_remove.append(guardian_to_remove_guid);
    };
};
self.change_guardians_using_storage(:guardian_guids_to_remove, guardians_to_add: array![new_guardian]);
```

However, this is ineffective. If the new guardian is already present and thus not removed, the call to `change_guardians_using_storage()` will still fail when attempting to add the same signer again:

```
fn change_guardians_using_storage(
    ref self: ComponentState<TContractState>,
    guardian_guids_to_remove: Array<felt252>,
    guardians_to_add: Array<SignerStorageValue>,
) {
    let guardian_to_remove_span = guardian_guids_to_remove.span();
    for guid_to_remove in guardian_guids_to_remove {
        self.guardians_storage.remove(guid_to_remove);
        self.emit_guardian_removed(guid_to_remove);
    };

    for guardian in guardians_to_add {
       assert(!guardian_to_remove_span.contains(guardian.into_guid()), 'argent/duplicated-guids');
        let guardian_guid = self.guardians_storage.insert(guardian);
        self.emit_guardian_added(guardian_guid);
    };

    self.assert_valid_storage();
}
```

`self.guardians_storage.insert(guardian)` will revert if the guardian is already present as the underlying linked_set_with_head used not allow duplicate entries. The same issue applies to the owner escape flow.

Code corrected:

Escaping to an existing owner/guardian is no longer supported.

`trigger_escape_owner()` / `trigger_escape_guardian()` and `complete_owner_escape()` / `complete_guardian_escape()` now revert if the new address is already the current one. This allowed the latter functions to be simplified, as they no longer need to handle this case separately.

6.8 Unneeded Mutability

Design | Low | Version 1 | Code Corrected

CS-AGAC050-004

The functions `assert_no_self_call`, `execute_multicall` and `execute_multicall_with_result` in `asserts.cairo` and `calls.cairo` take a mutable reference to a `Span<Call>`, even though the functions do not need to modify the `Span<Call>`.

Code corrected:

Mutability of the variables has been removed where not needed across the codebase.

6.9 Wrong Specification

Correctness | Low | Version 1 | Specification Changed

CS-AGAC050-005

1. A comment in `calculate_eip191_hash()` specifies `// This functions allows to verify eip-191 signatures`, but the function only constructs the EIP191 hash from the message.

Furthermore, the following parts of the documentation could be improved:

1. argent_account_escape.md: Using different terms “change owner requested”/”change guardian requested” in the graphic instead of trigger owner/guardian escape might be confusing for people learning the system.
2. signers_and_signatures.md: In the Concise Signatures section it would be helpful to mention that these are legacy signatures. In the code, `argent_account.parse_account_signature()` refers to this as the “legacy signatory array.” The same terminology should be used in the code and documentation.
3. argent_account_upgrades.md seems to be outdated / does not describe `recovery_from_legacy_upgrade()` which allows recovery of a stuck account after updating from `0.2.x` to `0.5.0`.

Specification changed:

The incorrect comment about `calculate_eip191_hash()` has been removed.

In addition, the following improvements have been made:

1. The graphic was updated to use consistent terminology.
2. The description of `argent_account.parse_account_signature()` has been updated to use the correct terminology.
3. The documentation was updated accordingly.

6.10 Outdated Libraries

Informational | Version 1 | Code Corrected

CS-AGAC050-003

Some of the libraries version set in `Scarb.toml` are outdated. Using the latest versions of libraries is usually good practice as they contain bugfixes and may be more efficient.

The outdated libraries are:

- `alexandria`: the latest version at the time of writing is `0.4.0`, or `rev=ae3d960`
- `openzeppelin`: the latest version at the time of writing is `1.0.0`

Code corrected:

The libraries have been updated to the latest for the cairo version used.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Gas Optimizations

Informational | Version 1 | Acknowledged

CS-AGAC050-001

Below is a list of possible gas optimizations (non-exhaustive):

1. In `SignerManager::add_signers()` and `SignerManager::remove_signers()` `self.threshold.write(new_threshold);` can be moved in the `if` block
2. In `Session::assert_valid_session()`, the `is_session()` check can be skipped as every callpath ending up there is already doing the check
3. In `GuardianManager::migrate_guardians_storage()`, `assert_valid_guardian_count()` may be skipped as the callpaths using the function are doing the same check later

---

Acknowledged:

Argent states:

```
No changes were made to the code. We think the gas optimizations are too small
to justify the changes in the code.
```

7.2 Misleading Event Name

Informational | Version 1 | Acknowledged

CS-AGAC050-002

The events `OwnerAddedGuid` and `OwnerRemovedGuid` in `signer_manager.cairo` mention an owner, while the actual entity being added or removed is a signer in the context of a multisig.

Acknowledged:

Client states:

```
No changes were made. We want to keep backward compatibility. Also, even if
the exact word is not the same, the signers can also be referred to as owners.
So it won't lead to mistakes.
```

8 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.

8.1 Breaking Changes for Integrators

Note Version 1

Some functions used by integrators with earlier versions of ArgentAccount may revert unexpectedly in `v0.5.0`. Examples of such functions are:

- `get_owner()`, `get_owner_type()`, `get_owner_guid()`: will revert if more than one owner is set
- `get_guardian()`, `get_guardian_type()`, `get_guardian_guid()`: will revert if more than one guardian is set
- `get_guardian_backup()`, `get_guardian_backup_type()`, `get_guardian_backup_guid()`: does not exist anymore in the new codebase

8.2 `is_valid_signature()` Return Value

Note Version 1

Users and integrators must be aware that `is_valid_signature()` in `Argent Account` will never return `false`, as it will revert on an invalid signature. In contrary, the `is_valid_signature()` of `Argent Multisig` will return `false` when given an invalid signature.