PUBLIC

Code Assessment

of the Argent Account Smart Contracts

May 30, 2024

Produced for

argent

by

CHAINSECURITY

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 Account according to Scope to support you in forming an opinion on their security risks.

Argent implements Argent account and Multisig account which are a set of smart contracts build on top of Account Abstraction of Starknet. Each Argent account is controlled by an owner who can use different signing methods to submit transactions to the account. The owner can set guardians to increase the security of their accounts, and help in recovery in case the private key is lost. Each Multisig is controlled by several signers, and a guardian can also be set for account recovery.

The most critical subjects covered in our audit are functional correctness, access control and signature handling. Security regarding all the aforementioned subjects is high. The general subjects covered are code complexity, specifications, and trustworthiness. The codebase is well structured, and specifications are satisfactory. Each Argent account has a trusted owner that has the ultimate control of an account and sets the rules for recovery. If owners enable recovery functionalities, they can choose any party to serve as guardian for their account.

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	4
• Code Corrected	4
Low-Severity Findings	9
• Code Corrected	9

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 Account repository based on the documentation files.

The following files were in the scope in **Version 1**:

```
presets/argent_account.cairo

account/interface.cairo
introspection/interface.cairo
introspection/src5.cairo

offchain_message/interface.cairo
offchain_message/precalculated_hashing.cairo

outside_execution/interface.cairo
outside_execution/outside_execution.cairo
outside_execution/outside_execution_hash.cairo

session/interface.cairo
session/session.cairo
session/session_hash.cairo

signer/eip191.cairo
signer/signer_signature.cairo
signer/webauthn.cairo

upgrade/interface.cairo
upgrade/upgrade.cairo

utils/asserts.cairo
utils/bytes.cairo
utils/calls.cairo
utils/hashing.cairo
utils/multicall.cairo
utils/serialization.cairo
utils/transaction_version.cairo
```

In **Version 2**, the following files were added to the scope:

```
presets/multisig_account.cairo
external_recovery/external_recovery.cairo
external_recovery/interface.cairo
multisig/interface.cairo
multisig/multisig.cairo
signer_storage/interface.cairo
```

```
signer_storage/signer_list.cairo
utils/array_ext.cairo
```

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

V	Date	Commit Hash	Note
1	07 Apr 2024	ca1c058daa138cdf65a9614c4b0276bc474cf1e7	Argent Account
2	29 Apr 2024	de7252f1e41f4e9a585b803c1dde7cf5e40bdf11	Multisig
3	13 May 2024	639dbb4e6924f3c95da17e01d7852af7864835c4	Version with Fixes
4	29 May 2024	0797df604602266a2270120a79cf8c850276811f	Final Version

For the **Argent Account**, the review was focused only on the Argent Account contract and its dependencies.

For the **Multisig**, the review was focused on the Multisig Account contract and its dependencies.

For the cairo smart contracts, the compiler version `2.6.3` was chosen. At the time of this review (April 2024) StarkNet v0.13.1 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 contracts not listed above are not part of this review. The files `array_store.cairo`, `user_account.cairo` and the folder `recovery`, were explicitly out of the scope. All external libraries and imports are assumed to behave correctly according to their high-level specification, without unexpected side effects. In addition, the correctness of sha256 implementations in cairo0 and cairo1 are out of scope.

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__`.

2.2 System Overview

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

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

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

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 presets: the Argent account and the multisig account, together with several components for the enriched features: sessions, outside execution, different signer types, and external recovery.

2.2.1 Argent Account

This is the core smart contract in the repository that implements the required functionalities for account abstraction on StarkNet.

Each Argent account has one owner which has the ultimate control of the account. A guardian can be enabled by the owner to cosign transactions for enhanced security. A backup guardian can be further added to assist in case the main guardian is unreachable or becomes malicious.

2.2.1.1 Extended Signature Support

The owner of an account (address `_signer`), guardian, and backup guardian represent different roles in a contract and are distinguished by their public keys. Any transaction initiated by the account should be signed by 1 or 2 of the privileged roles. Argent account supports multiple signature standards.

- All three roles could register a public key and sign on STARK curve.
- The signer or the backup guardian have the options to sign with Secp256k1, Secp256r1, or EIP191.
- The signer or the backup guardian have the option to sign via WebAuthn (detailed in the next section).

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 WebAuthn type, the Poseidon hash of 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 `WebauthnAssertion`. On (**Version 2**), a new struct `WebauthnSignature` is used for providing signatures.

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 `WebauthnAssertion` will be passed as transaction signature and will be validated against the stored `WebauthnSigner`:

1. Verifying Client Data Json: The request type, challenge, and origin will be parsed according to the offset, and then compared with the expected type, the transaction hash, and the stored origin respectively.
2. Verifying Authenticator Data: The RP ID hash signed by the authenticator should match the hash of the stored `rp_id_hash` in the signer. The `user-present` and `user-verified` bits must be set to true.
3. Verifying Assertion 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. 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 the current owner plus either guardian or the backup guardian. On **Version 2**, only the guardian can authorize a session.
4. The session transaction is signed by the session key and the main guardian. Backup guardian is not supported to sign such transactions.
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 the owner or the guardian, the following must hold:

1. Multicall cannot contain self as a target address (for admin functions).
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 the current owner (plus guardian or backup guardian) is valid.

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:

- The guardian can help to change the owner key in case the owner key is lost.
- The owner 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 owner 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 the owner and the guardian / backup guardian. The owner can also cancel an escape by initiating an escape from the guardian.

2.2.1.7 Other External Functions

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

- `change_owner()`: change the current owner to a new owner key. A signature from the new owner will be verified.
- `change_guardian()`: change the current guardian to a new guardian or simply remove the guardian if there is no backup guardian.
- `change_guardian_backup()`: change the current backup guardian or simply remove the backup guardian. This can only be done if a guardian exist.
- `cancel_escape()`: cancel an escape when it is not ready, or ready.

2.2.1.8 Upgradeability

The version of the argent account in this review is `0.4.0`. Upgradeability is implemented to support two scenarios:

1. An account from version `0.2.3` or older can be upgraded to version `0.4.0`.
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.

Upgrade from version 0.3.0+ 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()`:

    - 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.
    - The existing owner will be checked and the respective events are emitted.
    - In case a multicall (except calling self) is batched with the upgrade, it will be executed.

Upgrade from `0.2.3` to the current version shares a similar logic, while the new class hash is cached in `_implementation` first and then used by `replace_class_syscall` invoked in `execute_after_upgrade`.

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 `complete_upgrade()` by a system call to replace class hash.

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 Changes in Version 2

- In Version 2, in Argent account, the last attempt of triggering and finalizing an escape has been separated with the introduction of two new state variables: `last_guardian_trigger_escape_attempt`, `last_owner_trigger_escape_attempt`.
- Functions to parse single signature have been added.
- In addition, the return value of function `upgrade()` has been removed.

2.2.2.5 Changes in Version 3

In Version 3, the following changes were made:

- The session verification has been updated. The `session_callback()` in argent account has been renamed to `parse_and_verify_authorization()`. A cache mechanism has been implemented for session: a valid session that has been authorized by an owner and a guardian does not need to be verified again with the same owner and guardian if the flag `use_cache` has been set. In addition, a session cannot be authorized by the backup guardian anymore, it must be authorized by the guardian with the owner.
- The WebAuthn verification has been updated. Instead of reading and comparing fields from specified offsets in the client data json, a limited verification algorithm has been implemented which always expects a fixed order of fields.

2.2.3 Roles and Trust Model

2.2.3.1 Argent Account

The owner of the Argent account has the ultimate controls of the account and the funds held by it. Thus, it is assumed that the owner is 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 owner.

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

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

For upgrade-specific logic, we assume that the contract to be upgraded has the same storage layout as argent account v0.3.1.

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.

The Cairo0 implementation of sha256 is assumed to be correct and fully trusted, otherwise, malicious code may be executed in the context of an account during the hash computation. 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 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

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 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	4

- Concise Signature Is Not Support for Escape Owner Code Corrected
- Floating Offset May Be Leveraged to Verify Injected Data Code Corrected
- Reentrancy May Violate the Validation Logic Code Corrected
- Validation of Signature Values Code Corrected

Low-Severity Findings	9

- Execute Escape Can Be Reentered Code Corrected
- New Escape Params May Turn an Expired Escape to Ready Code Corrected
- Backup Guardian Can Authorize Sessions Code Corrected
- Gas Grief Protection May Interfere With Escape Security Period Code Corrected
- Incorrect Call Hash When Escape Is Cancelled Code Corrected
- Incorrect EIP191 Signature Type Code Corrected
- Incorrect Outside Execution Version in Domain Separator Code Corrected
- Minimum Security Period Is Not Restricted Code Corrected
- Update Escape Security Period Will Influence the Current Escape Code Corrected

Informational Findings	9

- WebAuthn Origin Is Not Restricted to Be Non Zero Specification Changed
- Inconsistent Escape Cancellation Event Code Corrected
- Signature Malleability Code Corrected
- Unused Calls in Assert_Valid_Signatures Code Corrected
- Base64UrlDecoder Accepts Both URL Safe and Trivial Base64 Encode Code Corrected
- Account Can Call Outside Execution on Itself Code Corrected
- Commented Code Code Corrected
- Different Span of U8 May Cast to Same Felt252 or U256 Specification Changed
- Function U8s to U32s Will Auto-Pad the Input Specification Changed

6.1 Concise Signature Is Not Support for Escape Owner

Correctness **Medium** **Version 1** **Code Corrected**

CS-AGAC-001

Argent account supports extended signer types while maintaining backward compatibility. When parsing the signature (`parse_signature_array()`), the signature will be interpreted as a legacy signature if it only has 2 or 4 elements:

- Length 2: it will always be interpreted as an owner signature.
- Length 4: it will be interpreted as an owner signature followed by a guardian signature.

Consequently, in case a guardian (or guardian backup) wants to initiate (`trigger_escape_owner()`) or finalize an escape (`escape_owner()`) with a concise signature, its signature will always be treated as an owner signature and fail to be verified, because these entrypoints expect only one signature from the guardian (or guardian backup).

Note: This issue was reported by Argent at the beginning of the review.

Code corrected:

Functions to parse single signature have been added: `parse_single_guardian_signature()`, `parse_single_owner_signature`. Hence a single concise signature from the owner or guardian can be correctly parsed for escaping functionalities.

6.2 Floating Offset May Be Leveraged to Verify Injected Data

Security **Medium** **Version 1** **Code Corrected**

CS-AGAC-002

When verifying the signature of a WebAuthn signer, the client data json will be checked to ensure the following fields are correct:

- The request `type` is `webauth.get`.
- The `challenge` should match the transaction hash.
- The `origin` should match the origin stored in the signer.

Argent account does not implement a full json parser, instead, it provides the flexibility to load the value at the offset and length specified by the user input. During the verification, it only loads the u8 at the indicated indices without checking other parts of the data.

This does not comply to the limited verification algorithm in the WebAuthn specification, where the `type`, `challenge`, and `origin` comes first in a fixed order. Consequently, in case there are duplicated / nested / injected fields in the client data json, the verification may still succeed. For instance, in case a user signs a client json data that contains two challenge fields, the signature could be verified successfully twice and both transaction could be executed.

```
{
    type: webauthn.get,
    challenge: base64.encode(hash(tx1)),
    origin: domain.com,
    {
        challenge: base64.encode(hash(tx2)),
    }
}
```

Although the user agent (browser) is assumed to always be honest, it is safer to eliminate the parsing ambiguity in the contract level.

Code corrected:

A limited verification algorithm has been implemented following WebAuthn specifications. This algorithm assembles the client data fields with a fixed offset and a predefined structure.

6.3 Reentrancy May Violate the Validation Logic

Security Medium Version 1 Code Corrected

CS-AGAC-003

When initiating a multicall from an Argent account, all the calls are firstly validated and then executed one by one. The transaction validation logic in `__validate__` and outside execution is dependent on the current account state (owner, guardian, and guardian backup). In theory, if an intermediate call changes the Account state, the consecutive calls may become invalid due to the changes (e.g., guardian update). Such changes should happen in separate transactions that execute a single call, hence the account cannot be a target contract in a multicall.

However, this restriction can be circumvented on a multicall initiated by outside execution. An intermediate call to another contract can reenter and modify the account's state. For instance, assume the owner and guardian sign two outside executions which can be triggered by anyone:

- Outside execution A: Call1 (interact with a malicious contract), Call2.
- Outside execution B: Call3 (change the current guardian to another one).

During the execution of outside execution A, an attacker can hijack the execution flow and reenter the argent account to trigger the outside execution B, which changes the guardian. The execution order would be: [Call1, Call3, Call2]. However, Call2 is still executed successfully, even though it could be invalid based on the latest account changes (applied by Call3).

Code corrected:

A reentrancy guard has been added to wrap the __**execute**__ entrypoint and outside execution, hence an outside execution cannot be reentered during an __**execute**__ or another outside execution.

6.4 Validation of Signature Values

Security Medium Version 1 Code Corrected

CS-AGAC-010

The function `is_valid_secp256r1_signature()` does not validate that the Signature values `r` and `s` are valid in the `secp256r1` curve:

```
fn is_valid_secp256r1_signature(..., signature: Secp256Signature) -> bool {
    let recovered = recover_public_key::<Secp256r1Point>(hash, signature).expect('...');
    let (recovered_signer, _) = recovered.get_coordinates().expect('argent/invalid-sig-format');
    recovered_signer == signer.pubkey.into()
}
```

Although no concrete attack scenario was identified in this usage, the signature values should be checked that they are valid on curve.

Code corrected:

Additional checks have been added to the Secp256 signature (for both Secp256k1 and Secp256r1 curves) verification to enforce that both `r` and `s` values are non-zero and within the curve's order. Furthermore, the s value is restricted to be in the lower half order to avoid signature malleability.

6.5 Execute Escape Can Be Reentered

Security | Low | Version 2 | Code Corrected

CS-AGAC-019

Reentrancy guard has been implemented to protect the outside execution entrypoints. However, the external function `execute_escape` is not protected by the reentrancy guard and could be called by anyone once an escape is ready. As a result, `execute_escape()` can be reentered if the execution flow is hijacked during a multicall from this multisig. As `execute_escape()` can change the signers and thresholds, the consecutive calls, which become invalid given the updated states, can still be executed successfully.

Code corrected:

Code has been corrected by adding a reentrancy guard to protect the `execute_escape` entrypoint.

6.6 New Escape Params May Turn an Expired Escape to Ready

Design | Low | Version 2 | Code Corrected

CS-AGAC-013

Similar to Update Escape Security Period Will Influence the Current Escape, signers of a multisig (with threshold signatures) can change the escape configurations via `toggle_escape()` when there is no ongoing escape (none or expired). However, it will not reset the escape. Hence, if the expire period is extended, an expired escape may become valid again.

Code corrected:

The function `toggle_escape()` in the external recovery module is revised to panic if the escaping params are updated while there is an ongoing escape in the states `Ready` or `NotReady`, while an expired escape is deleted from storage. This behavior is now in line with the update of security period in the Argent account.

6.7 Backup Guardian Can Authorize Sessions

Correctness **Low** **Version 1** **Code Corrected**

CS-AGAC-023

Functionalities related to the sessions are supported only for Argent accounts that have a guardian. A session should be authorized by both the owner of an account and the guardian. Afterwards, session transactions should be signed by a 3rd-party (dApp) and the guardian.

The session authorizations is checked in function `session_callback` which relies on function `is_valid_span_signature` and accepts signatures from both guardian and backup guardian of an account. However, session transactions should be signed only by the guardian (not backup guardian):

```
// checks that its the account guardian that signed the session
assert(state.is_guardian(token.guardian_signature.signer()), 'session/guardian-key-mismatch');
```

This behavior is inconsistent as the backup guardian can authorize a session but not sign the relevant transactions.

Code corrected:

The session authorization logic has been changed in **Version 3**. The function `assert_valid_session_authorization` enforces that a session can only be authorized by the owner and the guardian (backup guardian is not accepted).

6.8 Gas Grief Protection May Interfere With Escape Security Period

Correctness **Low** **Version 1** **Code Corrected**

CS-AGAC-004

Argent account enforces a limit on the frequency of triggering escape operations to protect against gas griefing attack. This limit is currently set to 12 hours (`TIME_BETWEEN_TWO_ESCAPES`). This is enforced to owner and guardian separately and applies to both trigger and finalize escape when validating a transaction in `__validate__()`. As a result, in case the `escape_security_period` is set too short, there is no chance to finalize the escape from the account itself due to the gas griefing protection. In this scenario the user or guardian can only use an outside execution to finalize the escape.

Note: This issue was reported by Argent at the beginning of the review.

Code corrected:

The gas griefing protection has been separated for triggering and finalizing escape. Hence, it no longer interferes with finalizing an escape.

6.9 Incorrect Call Hash When Escape Is Cancelled

Correctness **Low** **Version 1** **Code Corrected**

CS-AGAC-021

When a guardian triggers an escape in the multisig account, any ongoing escape (not ready or ready) will be automatically cancelled and overwritten by the new escape. However, the event `EscapeCanceled` is emitted with the new escape call hash instead of the existing one.

Code corrected:

Code has been corrected to emit the event with the existing escape call hash.

6.10 Incorrect EIP191 Signature Type

Correctness Low Version 1 Code Corrected

CS-AGAC-005

In the enum `SignerSignature`, the signature type of the EIP191 signer is defined as `Secp256r1Signature`. This is incorrect as the Ethereum uses Secp256k1 curve instead of Secp256r1, though both signatures are a tuple of `(r, s, y_parity)`.

```
enum SignerSignature {
    ...
    Eip191: (Eip191Signer, Secp256r1Signature),
    ...
}
```

Code corrected:

The signature struct of Secp256k1 and Secp256r1 has been renamed to `Secp256Signature` as they share the same signature fields.

6.11 Incorrect Outside Execution Version in Domain Separator

Correctness Low Version 1 Code Corrected

CS-AGAC-012

According to SNIP-9, the outside execution with revision 1 should use a domain separator with `version==`2. However, in ``OffChainMessageOutsideExecutionRev1` the domain separator with `version==`1 is used.

Code corrected:

The domain separator's version used for outside execution message has been corrected to 2.

6.12 Minimum Security Period Is Not Restricted

Design Low Version 1 Code Corrected

CS-AGAC-022

In argent account, an escape will be ready after a security period has elapsed, and it will expire after another security period. In addition, there is no restrictions on the security period set by the owner. In case the security period is too short (less than the block interval), it will expire in the next block and there is no time to finalize it.

Note: This issue was reported by Argent at the beginning of the review.

In Version 2, the escape security period is restricted by 10 minutes (`MIN_ESCAPE_SECURITY_PERIOD`) which is above the average block interval of StarkNet (April 2024). However, the escape security / expiry period is not restricted in the Multisig account.

Code corrected:

In Version 3, the minimum security period has also been enforced in the Multisig account.

6.13 Update Escape Security Period Will Influence the Current Escape

Design Low Version 1 Code Corrected

CS-AGAC-006

In Argent account, an escape is considered ready to be finalized after a delay of `escape_security_period`. Once ready, it can be finalized within a limited time window `escape_security_period`. The owner and guardian can update this period at any time by co-signing a transaction that calls `set_escape_security_period()`, which has the following consequences:

- The `ready_at` timestamp has been written to the storage of current escape while the expiry is computed in runtime. Such updates influence only the expiration window for existing escapes, hence the security delay may be inconsistent with the expiration window.
- An escape that is ready may become expired if the new `escape_security_period` is shorter.
- An escape that has expired may become ready to be finalized again if the new `escape_security_period` is longer.

Code corrected:

The function `set_escape_security_period()` in Argent account has been revised to panic if the security period is updated while there is an ongoing escape in the states `Ready` or `NotReady`, while an expired escape is deleted from storage.

6.14 Account Can Call Outside Execution on Itself

Informational Version 1 Code Corrected

CS-AGAC-014

Outside execution (`execute_from_outside()`, `execute_from_outside_v2()`) is provided to sponsor the multicalls of an account, so the account does not pay the gas fee on itself. However, there is no restriction on the caller of outside execution, as a result, an account can call the outside execution entrypoint from itself (i.e., `__execute__`).

Code corrected:

A reentrancy guard has been added to `__execute__()` and outside execution, which prohibits an account to call outside execution on itself.

6.15 Base64UrlDecoder Accepts Both URL Safe and Trivial Base64 Encode

Informational **Version 1** **Code Corrected**

CS-AGAC-007

In alexandria's implementation, the Base64UrlDecoder has the same logic as Base64Decoder. As a result, a decode will succeed if:

- the input is a non-url-safe encoding.
- the input is a mixed use of base64 (+, `/`) and base64-url-safe (_, `-`) chars.

Code corrected:

The WebAuthn verification logic has been changed and the Base64UrlDecoder is not used anymore.

6.16 Commented Code

Informational **Version 1** **Code Corrected**

CS-AGAC-015

The file `offchain_message/interface.cairo` has the following commented code:

```
// impl StructHashStarknetDomain<-IStructHash<StarkNetDomain>>
of IStarknetDomainHash<StarkNetDomain>
```

Code corrected:

The redundant commented code has been removed.

6.17 Different Span of U8 May Cast to Same Felt252 or U256

Informational **Version 1** **Specification Changed**

CS-AGAC-008

Type casting is implemented from a span of u8 to u256 (`SpanU8TryIntoU256`) or felt252 (`SpanU8TryIntoFelt252`), where different span of u8 may be casted to the same output. For instance, both a=`[1]` and b=`[0, 1]` will be casted to `1`. Specifications are missing for these implementations, and external systems using them should be aware of this behavior.

Specification changed:

The following description is added above the functions:

```
/// @dev Leading zeros are ignored and the input must be at most
    32 bytes long (both [1] and [0, 1] will be casted to 1)
```

6.18 Function U8s to U32s Will Auto-Pad the Input

Informational **Version 1** **Specification Changed**

CS-AGAC-024

Function `u8s_to_u32s(mut input: Span<u8>)` converts a span of u8 to an array of u32. It does not enforce the input contains a multiple of 4 elements. Instead, it will auto-pad the input with 0 in case the u8 at the tail cannot make a u32. As a result, different input may lead to the same output.

Specification changed:

Specification and the function name (now `u8s_to_u32s_pad_end`) has been updated to reflect the expected behavior of this function.

6.19 Inconsistent Escape Cancellation Event

Informational **Version 1** **Code Corrected**

CS-AGAC-020

Signers of a multisig can cancel an escape via `cancel_escape()`, which will also emit an event for cancelling an escape that has already expired. This is inconsistent with the escape cancellation in `trigger_escape()`, where overwriting an expired escape will not emit a cancellation event.

Code corrected:

Code has been corrected and the cancellation event will not be emitted if the escape has already expired.

6.20 Signature Malleability

Informational **Version 1** **Code Corrected**

CS-AGAC-017

The Starknet core library does not enforce a range check for the s value of a signature in functions such as `check_ecdsa_signature` and `recover_public_key`. As a result, signature malleability is not eliminated inherently. However, in argent account, signatures of transactions (via `__execute__`, outside execution, and session interaction) are signed with a unique nonce to prevent replay attacks.

Code corrected:

Checks of the s value have been added to `is_valid_secp256k1_signature` and `is_valid_secp256r1_signature` which addressed the malleability of Secp256k1 and Secp256r1 signatures. The s value for Stark curve is still unrestricted and Argent states:

```
We didn't address the issue for signature with the Stark Curve. Because that will
break compatibility with most devtools and clients. We might instead bring this issue
to the community so we can coordinate to remove malleability for Starknet Signatures.
```

6.21 Unused Calls in Assert_Valid_Signatures

Informational Version 1 Code Corrected

CS-AGAC-018

The function `assert_valid_signatures` in `ArgentMultisigAccount` verifies signatures given an execution hash. The function takes an extra input `calls` which is not used.

Code corrected:

The unused function argument `calls` has been removed in Version 3.

6.22 WebAuthn Origin Is Not Restricted to Be Non Zero

Informational Version 1 Specification Changed

CS-AGAC-011

According to the specification of `WebauthnSigner`, the origin field of the signer should not be zero. Nevertheless, this is neither enforced on the origin type nor checked in the account constructor.

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

Specification changed:

Specification has been changed and `origin` is no longer restricted to be non-zero.

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 Last Escape Attempt Is Rounded Down

Informational **Version 1** **Acknowledged**

CS-AGAC-009

At the time of this review (April 2024), the return value of `get_block_timestamp()` will be rounded down to the nearest hour in the context of `__validate__`. Hence the timestamp of the last escape attempt written into storage will be rounded down. Currently it is not ensured that there is at least 12 hours (`TIME_BETWEEN_TWO_ESCAPES`) between two escapes. As a result, the guardian may abuse the rounding feature to trigger another escape sooner. For instance:

- The guardian can trigger an escape at 11:59, and the timestamp will be rounded down to 11:00.
- After 23:00, the guardian will be able to trigger another escape even though only 11 hours have elapsed.

Argent is aware of this behavior and considers it to be compliant with specifications as the rounding is factored in `TIME_BETWEEN_TWO_ESCAPES`.

7.2 Multisig `__validate__` Can Be Restricted to View

Informational **Version 1**

CS-AGAC-016

The function `__validate__` of multisig is defined as an external function and modifies state. Differently, `Multisig.__validate__()` does not modify any state variable but it is not restricted as a view function. This is intended by Argent to be compliant to the `IAccount` interface.

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 A Transaction of a Multisig Could Become Valid if Threshold Is Lowered

Note **Version 1**

A transaction or an outside execution operation of the multisig requires a threshold number of signatures from the signers to be executed. If one operation receives less signatures than the current threshold, it cannot be executed. However, a previously signed operation below quorum could become valid in the future if the following conditions are met:

- threshold is lowered.
- the nonce is still valid.
- signers are still active.

This is more likely for signed outside execution which relies on random nonces. Signers of multisig should be aware of this scenario and take measures if the threshold of multisig is lowered.

8.2 Current Limitation on StarkNet

Note **Version 1**

At the time of this review (April 2024), the StarkNet sequencer is centralized. Users are subject to the censorship and the potential downtime of the sequencer. Hence an user operation (e.g. finalize, cancel escape) may be blocked or delayed.

Currently, only the following fee is charged for a StarkNet transaction:

- The marginal cost of verifying the transaction on L1.
- The cost of posting the state diffs induced by the transaction to L1.

Due to the cheap transaction cost, it may be profitable for a malicious guardian / guardian backup to control an account with a short security period by congesting the network to take over accounts:

- The attacker triggers an owner escape for one or multiple accounts.
- The attacker spams the network with large amount of transactions, which forbid the owner's transaction to cancel the escape.
- The attacker finalizes the escape attempts after the (short) security period and takes over accounts.

8.3 Different Signers May Use the Same Private Key

Note **Version 1**

Both `Secp256k1` and `EIP191` signers compute signatures on the `Secp256k1` curve. Hence, it is possible that two signers (assumed to be distinct) can be added in a multisig account even if they use the same private key. In addition, in argent account it is not enforced onchain that the signer, guardian, and backup guardian keys are different.

Users are responsible to secure their private keys and setup their accounts correctly. Theoretically, users can anyway use the same private key for different signer types, therefore, reducing the security of the multisig or argent account.

8.4 Escape Guardian Can Be Cleared in an Edge Case

Note Version 1

In argent account, the owner has higher priority than the guardian / guardian backup. In case the owner has triggered an escape from the guardian, the guardian cannot initiate an escape from the owner before it has been cleared, cancelled, or expired. In an edge case if there are signed but unused outside executions to modify account state (for instance `change_guardian_backup()`), the guardian may leverage them to clear an owner's escape.

8.5 Guardian's Signature Is Required for Argent Account in a Deploy Account Transaction

Note Version 1

The multisig account requires only one signature from a valid signer for deployment. Differently, the function `__validate_deploy__` in the Argent account requires that the transaction is signed by both the owner and the guardian in case the account is initialized with a guardian.

If the precomputed address of an argent account has been pre-funded but the guardian becomes malicious or is unavailable anymore, the account cannot be deployed by a deploy account transaction. In this case, the account can be deployed via `deploy_syscall` with the same parameters and salt (to deploy at the same precomputed address), where the signature of the guardian is not required.

8.6 Guardians Can Invalidate Certain User's Transactions

Note Version 1

A guardian or a backup guardian can frontrun a previously consigned transaction by submitting another transaction that requires a single signature, such as calling `trigger_escape_owner()`, and use the same nonce as the user's transaction. As a result, the previously cosigned transaction becomes invalid due to using a consumed nonce. This is possible for both: Starknet account nonces, and nonces used in the outside execution.

A similar approach to consume nonces is possible through sessions. The session and the guardian cosign the outside execution hash. Sessions are allowed to choose any nonce for the outside execution. Hence, it can consume any outside execution nonce by triggering an outside execution. Consequently, a user transaction signed by the owner and guardian might become invalid due to the nonce reuse.

The owner has the possibility to replace a malicious guardian by initiating a guardian escape process.

8.7 Race Condition Between the Guardian and Guardian Backup

Note Version 1

In case the owner key is lost, the guardian and guardian backup can compete to control the account. They can call *trigger_escape_owner()* via outside execution to skip the gas griefing protection and always overwrite each other's escape before it is ready.

8.8 Users Are Responsible for Account Upgrade

Note Version 1

Both Argent Account and Multisig allow users to upgrade to a new implementation by specifying a class hash. This functionality is powerful and should be handled correctly and carefully by users.

The current versions of the Argent Account (`0.4.0`) and Multisig (`0.2.0`) perform some sanity checks to ensure the compatibility between implementations before upgrading. However, users should be aware that the contracts do not prevent upgrades to incorrect implementations (which might lead to locked accounts), or malicious implementations (which might lead to account takeover and lost funds).