CAIRO SECURITY CLAN

ARGENT GIFTING CONTRACT

SECURITY ASSESMENT REPORT

JUNE 2024

Prepared for

ARGENT

Contents

1 About Cairo Security Clan

Cairo Security Clan is a leading force in the realm of blockchain security, dedicated to fortifying the foundations of the digital age. As pioneers in the field, we specialize in conducting meticulous smart contract security audits, ensuring the integrity and reliability of decentralized applications built on blockchain technology.

At Cairo Security Clan, we boast a multidisciplinary team of seasoned professionals proficient in blockchain security, cryptography, and software engineering. With a firm commitment to excellence, our experts delve into every aspect of the Web3 ecosystem, from foundational layer protocols to application-layer development. Our comprehensive suite of services encompasses smart contract audits, formal verification, and real-time monitoring, offering unparalleled protection against potential vulnerabilities.

Our team comprises industry veterans and scholars with extensive academic backgrounds and practical experience. Armed with advanced methodologies and cutting-edge tools, we scrutinize and analyze complex smart contracts with precision and rigor. Our track record speaks volumes, with a plethora of published research papers and citations, demonstrating our unwavering dedication to advancing the field of blockchain security.

At Cairo Security Clan, we prioritize collaboration and transparency, fostering meaningful partnerships with our clients. We believe in a customer-oriented approach, engaging stakeholders at every stage of the auditing process. By maintaining open lines of communication and soliciting client feedback, we ensure that our solutions are tailored to meet the unique needs and objectives of each project.

Beyond our core services, Cairo Security Clan is committed to driving innovation and shaping the future of blockchain technology. As active contributors to the ecosystem, we participate in the development of emerging technologies such as Starknet, leveraging our expertise to build robust infrastructure and tools. Through strategic guidance and support, we empower our partners to navigate the complexities of the blockchain landscape with confidence and clarity.

In summary, Cairo Security Clan stands at the forefront of blockchain security, blending technical prowess with a client-centric ethos to deliver unparalleled protection and peace of mind in an ever-evolving digital landscape. Join us in safeguarding the future of decentralized finance and digital assets with confidence and conviction.

2 Disclaimer

Disclaimer Limitations of this Audit:

This report is based solely on the materials and documentation provided by you to Cairo Security Clan for the specific purpose of conducting the security review outlined in the Summary of Audit and Scoped Files. The findings presented here may not be exhaustive and may not identify all potential vulnerabilities. Cairo Security Clan provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, occurs entirely at your own risk.

Inherent Risks of Blockchain Technology:

Blockchain technology remains in its developmental stage and is inherently susceptible to unknown risks and vulnerabilities. This review is specifically focused on the smart contract code and does not extend to the compiler layer, programming language elements beyond the reviewed code, or other potential security risks outside the code itself.

Report Purpose and Reliance:

This report should not be construed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. No third party should rely on this report for any purpose, including making investment or purchasing decisions.

Liability Disclaimer:

To the fullest extent permitted by law, Cairo Security Clan disclaims all liability associated with this report, its contents, and any related services and products arising from your use. This includes, but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services:

Cairo Security Clan does not warrant, endorse, guarantee, or assume responsibility for any products or services advertised by third parties within this report, nor for any open-source or third-party software, code, libraries, materials, or information linked to, referenced by, or accessible through this report, its content, and related services and products. This includes any hyperlinked websites, websites or applications appearing on advertisements, and Cairo Security Clan will not be responsible for monitoring any transactions between you and third-party providers. It is recommended that you exercise due diligence and caution when considering any third-party products or services, just as you would with any purchase or service through any medium.

Disclaimer of Advice:

FOR THE AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, ACCESS, AND/OR USE, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Executive Summary

This document presents the security review performed by Cairo Security Clan on the Argent wallet.

Argent is the original smart wallet. Pioneered "Guardians" for social recovery, and have been using Account Abstraction since 2018. Argent Wallet offers a simple onboarding process, allowing users to create a wallet in a matter of minutes, without the need for seed phrases or private keys.Learn more from docs.

The audit was performed using

- manual analysis of the codebase,
- automated analysis tools,
- simulation of the smart contract,
- analysis of edge test cases

5 points of attention, where 0 is classified as Critical, 0 is classified as High, 0 is classified as Medium,1 is classified as Low,1 is classified as Informational and 3 are classified as Best Practices. The issues are summarized in Fig. 1.

This document is organized as follows. Section 1 About Cairo Security Clan. Section 2 Disclaimer. Section 3 Executive Summary. Section 4 Summary of Audit. Section 5 Risk Classification. Section 6 Issues by Severity Levels. Section 7 Test Evaluation.

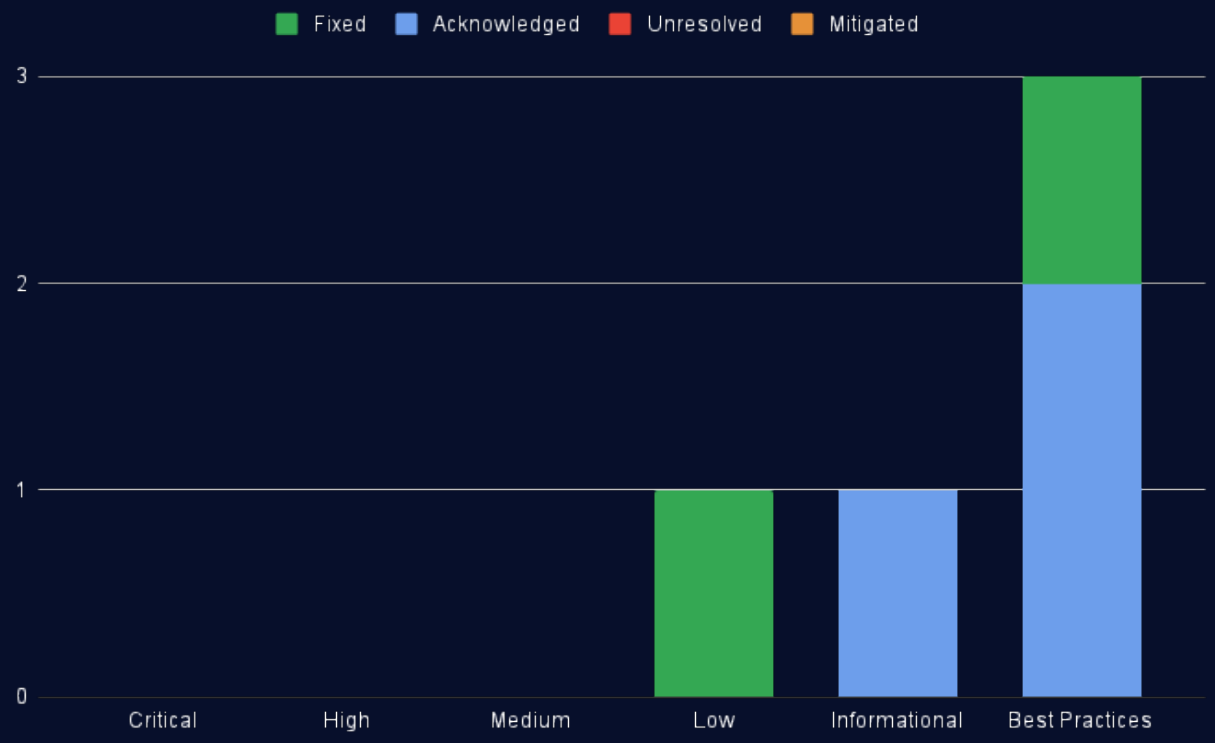

Fig 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Informational** (1), **Best Practices** (3). **Distribution of status: Fixed** (2), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

4 Summary of Audit

Audit Type	Security Review
Cairo Version	2.6.3
Response from Client	12/07/2024
Final Report	15/07/2024
Repository	argent-gifting-contracts
Initial Commit Hash	d63be1937c065da6e6e56afcbfe26de44af393ec
Final Commit Hash	fc1f40dc255cf2f4264c0d9f801781e8371d6406
Documentation	Website documentation
Test Suite Assessment	High

4.1 Scoped Files

	Contracts
1	/src/contracts/claim_hash.cairo
2	/src/contracts/escrow_account.cairo
3	/src/contracts/escrow_library.cairo
4	/src/contracts/gift_data.cairo
5	/src/contracts/gift_factory.cairo
6	/src/contracts/outside_execution.cairo
7	/src/contracts/timelock_upgrade.cairo
8	/src/contracts/utils.cairo
9	/src/lib.cairo

4.2 Issues

	Findings	Severity	Update
1	Gifts can be cancelled when they have been claimed.	Low	Fixed
2	Missing event emission `UpgradeExecuted`.	Informational	Acknowledged
3	Improve checks on `execute_action(...)` function.	Best Practices	Acknowledged
4	Unnecessary assertion.	Best Practices	Acknowledged
5	Access control for function `perform_upgrade(...)`.	Best Practices	Fixed

5 Risk Classification

The risk rating methodology used by Cairo Security Clan follows the principles established by the CVSS risk rating methodology. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Likelihood		
		High	**Medium**	**Low**
Impact	**High**	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Info/Best Practices

To address issues that do not fit a High/Medium/Low severity, Cairo Security Clan also uses three more finding severities: **Informational**, **Best Practices** and **Gas**

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

b) **Gas** findings are used when some piece of code uses more gas than it should be or have some functions that can be removed to save gas.

6 Issues by Severity Levels

6.1 Low

6.1.1 Gifts can be cancelled when they have been claimed.

File(s): /src/contracts/escrow_library.cairo

Description: According to the documentation, after the gifts are created, they can be canceled by the sender provided they have not been claimed yet. The sender will retrieve both the `gift_amount` and the `fee_amount` they deposited for that gift. In the codebase, this is done through the function `cancel(...)`.

```
fn cancel(ref self: ContractState, gift: GiftData) {
    let contract_address = get_contract_address();
    assert(get_caller_address() == gift.sender, 'escr-lib/wrong-sender');
    let gift_balance = balance_of(gift.gift_token, contract_address);
    assert(gift_balance > 0, 'escr-lib/already-claimed');
}
```

In this function, the check to ensure that the gift has not been claimed is `gift_balance > 0`. However, this condition can be bypassed in two ways:

- If `gift_token == fee_token`, then after claiming, some dust remains in the contract, making `gift_balance > 0`.
- The sender intentionally sends 1 wei of `gift_token` to the contract to bypass the check, thus allowing them to cancel the gift.

Recommendation(s): Change the check in the function `cancel(...)` to make it consistent with other checks in `proceed_with_claim(...)` and `claim_dust(...)`

```
- assert(gift_balance > 0, 'escr-lib/already-claimed');
+ assert(gift_balance >= gift.gift_amount, 'escr-lib/already-claimed');
```

Status: Fixed

Update from client: We have updated the code to the proposed alternative, for consistency, but we wanted to make clear that the change won't prevent the sender/receiver from collecting the dust. Both Sender and Receiver can collect the dust by sending more gift tokens to the escrow contract, and then using the `cancel(...)` or `claim_external(...)` methods. Fixed in commit.

6.2 Informationals

6.2.1 Missing event emission `UpgradeExecuted`.

File(s): /src/contracts/timelock_upgrade.cairo

Description: In the `timelock_upgrade` contract, there are 3 defined events, one of which is `UpgradeExecuted`. This event is supposed to be emitted when a pending upgrade is executed through the function `upgrade(...)`. However, it is currently missing in this function and is not used anywhere in the contract.

```
fn upgrade(ref self: ComponentState<TContractState>, calldata: Array<felt252>) {
    self.assert_only_owner();
    let pending_upgrade: PendingUpgrade = self.pending_upgrade.read();
    assert(pending_upgrade != Default::default(), 'upgrade/no-pending-upgrade');
    let PendingUpgrade { implementation, ready_at, calldata_hash } = pending_upgrade;

    assert(calldata_hash == poseidon_hash_span(calldata.span()), 'upgrade/invalid-calldata');

    let current_timestamp = get_block_timestamp();
    assert(current_timestamp >= ready_at, 'upgrade/too-early');
    assert(current_timestamp < ready_at + VALID_WINDOW_PERIOD, 'upgrade/upgrade-too-late');

    self.pending_upgrade.write(Default::default());
    ITimelockUpgradeCallbackLibraryDispatcher { class_hash: implementation }
        .perform_upgrade(implementation, calldata.span());
}
```

Recommendation(s): Consider adding the `UpgradeExecuted` event emission in the function `upgrade(...)`.

Status: Acknowledged

Update from client: The idea is that the event is emitted by the new implementation, this will allow us to even make changes to the even itself after the actual upgrade.

6.3 Best Practices

6.3.1 Improve checks on `execute_action(...)` function.

File(s): /src/contracts/escrow_library.cairo

Description: In the `execute_action(...)` function, there is an assertion that checks the selector is one of the supported selectors. However, argument lengths can also be checked. Since it is not vulnerable currently, it improves the checks.

```
fn execute_action(
    ref self: ContractState, this_class_hash: ClassHash, selector: felt252, args: Span<felt252>
) -> Span<felt252> {
    let is_whitelisted = selector == selector!("claim_external")
        || selector == selector!("claim_dust")
        || selector == selector!("cancel");
    assert(is_whitelisted, 'escr-lib/invalid-selector');
}
```

Recommendation(s): Consider adding a check that verifies argument lengths and also corrects them for each selector.

Status: Acknowledged

Update from client: The code generated by the compiler for the `claim_external`, `claim_dust`, and `cancel` (or any external function), already takes care of parsing the calldata and ensuring the length is correct.

6.3.2 Unnecessary assertion.

File(s): /src/contracts/gift_factory.cairo

Description: The variable `escrow_class_hash` passed as parameter to the `deposit(...)` function. However, this assertion reverts if this value does not equal the `escrow_class_hash_storage` value. Flow can continue directly with `escrow_class_hash_storage` instead of comparing with the parameter.

```
fn deposit(
    ref self: ContractState,
    escrow_class_hash: ClassHash,
    // ...
) {
    // ...
    let escrow_class_hash_storage = self.escrow_class_hash.read();
    assert(escrow_class_hash_storage == escrow_class_hash, 'gift-fac/invalid-class-hash');
    // ...
}
```

Recommendation(s): Consider removing assertion.

Status: Acknowledged

Update from client: Without this check, the frontend needs to wait for the transaction to be finalized to then query the `escrow_class_hash` hash from the events. We supply the `escrow_class_hash` on purpose so the flow can be: First query the factory the current classhash, then send the deposit transaction; no further action is needed. It will be guaranteed that the gift will be created as expected (expected address and class hashes, or the whole thing reverts). This might allow for a more optimistic UI where we let users share the link without waiting for the events. The transaction might still fail, but at least the sender still has his funds.

6.3.3 Access control for function `perform_upgrade(...)`.

File(s): /src/contracts/gift_factory.cairo

Description: Since the function `perform_upgrade(...)` in the current `gift_factory` will always revert, it is not an issue. However, we checked the mock `future_factory` and want to warn you about the risk of this function being called directly.

Since there is no access control and it is an external function, anyone can call it to replace the class hash with an arbitrary implementation.

```
#[abi(embed_v0)]
impl TimelockUpgradeCallbackImpl of ITimelockUpgradeCallback<ContractState> {
    fn perform_upgrade(ref self: ContractState, new_implementation: ClassHash, data: Span<felt252>) {
        starknet::syscalls::replace_class_syscall(new_implementation).unwrap();
    }
}
```

Recommendation(s): Consider adding a check to ensure this is a self-call to upgrade.

Status: Fixed

Update from client: Fixed in commit.

7 Test Evaluation

7.1 Compilation Output

```
Run scarb build
    Updating git repository github.com/keep-starknet-strange/alexandria
    Updating git repository github.com/openzeppelin/cairo-contracts
    Updating git repository github.com/foundry-rs/starknet-foundry
   Compiling argent_gifting v0.1.0 (/013-ARGENT-GIFTING/013-ARGENT-GIFTING/contracts/Scarb.toml)
    Finished release target(s) in 17 seconds
```

7.2 Tests Output

7.2.1 Cairo Tests

```
Run scarb test
     Running test argent_gifting (snforge test)
Warning:  Package snforge_std version does not meet the recommended version requirement =0.25.0, it might result
     in unexpected behaviour
   Compiling argent_gifting v0.1.0 (/013-ARGENT-GIFTING/013-ARGENT-GIFTING/contracts/Scarb.toml)
    Finished release target(s) in 14 seconds

Collected 2 test(s) from argent_gifting package
Running 0 test(s) from src/
Running 2 test(s) from tests/
[PASS] tests::test_claim_hash::precalculated_hash_sepolia (gas: ~3)
[PASS] tests::test_claim_hash::precalculated_hash_mainnet (gas: ~3)
Tests: 2 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

7.2.2 Typescript Tests

```
scarb run test-ts
   Compiling argent_gifting v0.1.0 (\contracts\Scarb.toml)
    Finished release target(s) in 19 seconds

yarn run v1.22.18
warning ..\..\package.json: No license field
Done in 2.12s.

yarn run v1.22.18
warning ..\..\package.json: No license field

Provider: 127.0.0.1:5050
Deployer: 0x64b48806902a367c8598f4f95c305e8c1a1acba5f082d294a43793113115691

  Escrow Account
       EscrowAccount declared
       EscrowLibrary declared
       GiftFactory declared
    [PASS] Test only protocol can call validate (25930ms)
    [PASS] Test only protocol can call execute
    [PASS] Test escrow can only do whitelisted lib calls
    [PASS] Test escrow contract cant call another contract
    [PASS] Test escrow contract can only call 'escrow_internal'
    [PASS] Test escrow contract cant perform a multicall
    [PASS] Test cannot call 'claim_internal' twice
  Cancel Gift
    [PASS] fee_token == gift_token
       MockERC20 declared
    [PASS] fee_token != gift_token (17965ms)
    [PASS] wrong sender
    [PASS] owner reclaim dust (gift_token == fee_token)
    [PASS] escr-lib/already-claimed (gift_token != fee_token)
```

```
  Claim External
    [PASS] gift_token == fee_token flow using txV3: false (no dust receiver)
    [PASS] gift_token == fee_token flow using txV3: false  (w/ dust receiver)
    [PASS] gift_token == fee_token flow using txV3: true (no dust receiver)
    [PASS] gift_token == fee_token flow using txV3: true  (w/ dust receiver)
    [PASS] gift_token != fee_token (w/ dust receiver)
    [PASS] gift_token != fee_token (no dust receiver)
    [PASS] Zero Receiver
    [PASS] Cannot call claim external twice
    [PASS] Invalid Signature
    [PASS] Claim gift cancelled
  Claim Internal
    [PASS] gift token == fee token using txV3: false
    [PASS] Cant claim if no fee amount deposited (fee token == gift token) using txV3: false
    [PASS] Test max fee too high using txV3: false
    [PASS] gift token == fee token using txV3: true
    [PASS] Cant claim if no fee amount deposited (fee token == gift token) using txV3: true
    [PASS] Test max fee too high using txV3: true
    [PASS] Cant call gift internal twice
  Deposit
    [PASS] Double deposit
    [PASS] Deposit works using txV3: false (gift token == gift token)
    [PASS] Deposit works using txV3: false with 0 fee amount set (gift token == gift token)
    [PASS] Deposit works using txV3: false with 0 fee amount set (gift token != gift token)
    [PASS] Deposit works using: false (gift token != gift token)
    [PASS] Max fee too high gift_amount > fee_amount (gift token == fee token)
    [PASS] Deposit works using txV3: true (gift token == gift token)
    [PASS] Deposit works using txV3: true with 0 fee amount set (gift token == gift token)
    [PASS] Deposit works using txV3: true with 0 fee amount set (gift token != gift token)
    [PASS] Deposit works using: true (gift token != gift token)
    [PASS] Max fee too high gift_amount > fee_amount (gift token == fee token)
    [PASS] Deposit fails class hash passed != class hash in factory storage
        BrokenERC20 declared
    [PASS] Deposit fails if erc reverts (12637ms)
  All events are emitted
    [PASS] Deposit
    [PASS] Cancelled
    [PASS] Claim Internal (2501ms)
    [PASS] Claim External (2566ms)
  Test Core Factory Functions
    [PASS] Calculate escrow address
    [PASS] claim_dust: false
    [PASS] claim_dust: true
    [PASS] Pausable (2598ms)
    Ownable
      [PASS] Pause
      [PASS] Unpause
      [PASS] Ownable: Get Dust
  Test Factory Upgrade
       FutureFactory declared
    [PASS] Upgrade (3210ms)
    [PASS] cannot downgrade (10276ms)
    [PASS] only-owner
    [PASS] Invalid Calldata
    [PASS] Too Early
    [PASS] Too Late
    Propose Upgrade
      [PASS] implementation-null
      [PASS] only-owner
      [PASS] replace pending implementation /w events
    Cancel Upgrade
      [PASS] Normal flow /w events
      [PASS] No new implementation
      [PASS] Only Owner

  65 passing (2m)
Done in 137.46s.
```