CAIRO SECURITY CLAN

BROTHER ID

SECURITY ASSESEMENT REPORT

DECEMBER 2024

Prepared for

STARKNET BROTHER

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 Brother ID.

Brother id is a name service and product of Starknet Brother $BROTHER. Starknet brother is a memecoin created with Unruggable Meme, a memecoin standard and deployment tool designed to ensure a maximum safety for memecoin traders.

The audit was performed using

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

9 points of attention, where 0 is classified as Critical, 0 is classified as High, 1 is classified as Medium,2 are classified as Low,1 is classified as Informational and 5 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.

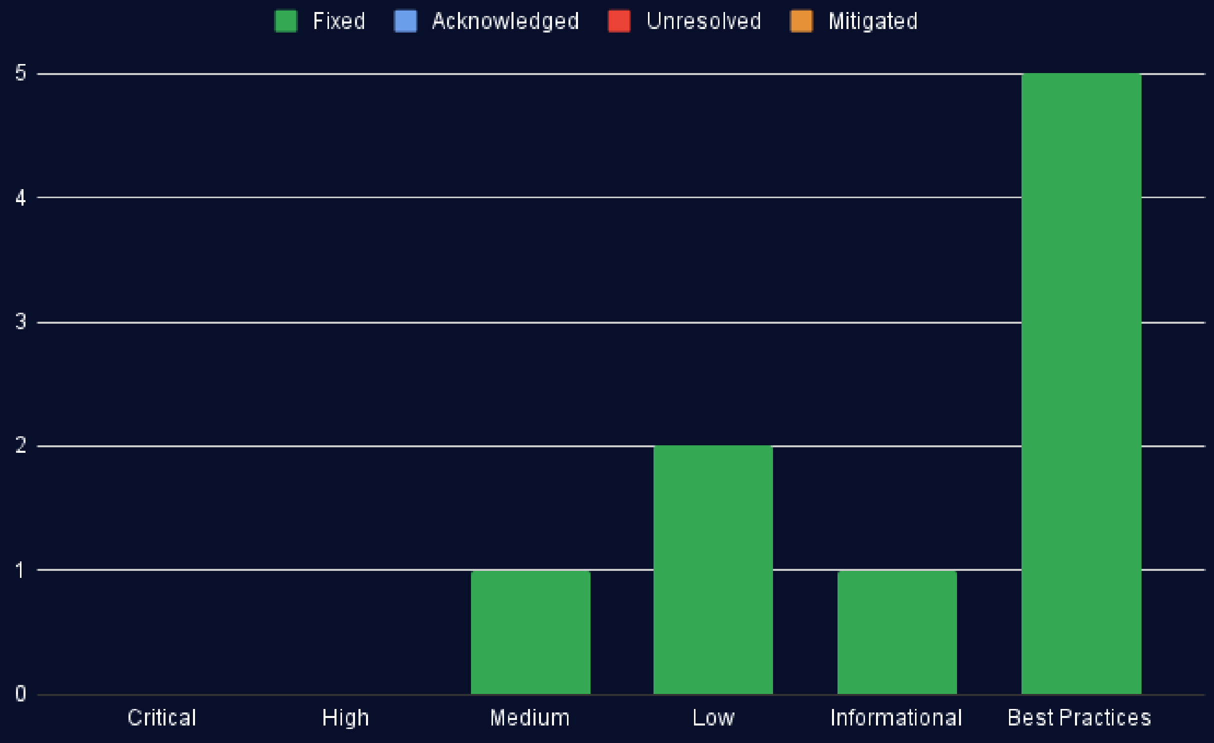

Fig 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (2), **Informational** (1), **Best Practices** (5).
Distribution of status: Fixed (9), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0).

4 Summary of Audit

Audit Type	Security Review
Cairo Version	2.2.0
Final Report	24/12/2024
Final Gist Link	gist
Documentation	Website documentation
Test Suite Assessment	NONE

4.1 Scoped Files

	Contracts
1	brother_id.cairo

4.2 Issues

	Findings	**Severity**	**Update**
1	Domain ownership ambiguity enabled by inconsistent management of tokenId when expired domains are repurchased	Medium	Fixed
2	Primary domain gets reset on transfer when user owns multiple domains	Low	Fixed
3	Domain renewal overwrites remaining registration period	Low	Fixed
4	Expired domains can be transferred	Informational	Fixed
5	Unused storage variable	Best Practices	Fixed
6	Unchecked return value of ERC20 `transferFrom()` in `register_domain()` & `renew_domain()`	Best Practices	Fixed
7	Unnecessary caller check	Best Practices	Fixed
8	Unused code	Best Practices	Fixed
9	Confusing function name `eth_contract()`	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;

c) **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 Medium

6.1.1 Domain Ownership Ambiguity Enabled by Inconsistent Management of TokenId When Expired Domains Are Repurchased

File(s): brother_id.cairo

Description: Once an expired domain is repurchased, it can still be transferred using its old `tokenId` due to the contract not updating or clearing the old `_token_id_to_domain` mapping. This allows the previous owner to transfer the domain to another address without the current owner's consent, essentially renewing the domain at the expense of the current owner's purchase.

Here is a proof of concept (PoC) explaining the vulnerability:

1. **Initial Registration**
   - User1 registers a domain (e.g., "tommy") for 2 years and becomes the owner of this domain with, let's say, `tokenId` 3, i.e., `_token_id_to_domain[3] -> tommy`.
   - New domain details for User1:

```
DomainDetails {
    handler: tommy,
    resolver: user1,
    token_id: 3,
    expiry_date: user1Expiry,
    last_transfer_time: now,
};
```

2. **Domain Expiry**
   - After the domain expires, without renewal, it becomes available for registration again.
3. **Re-registration by Another User**
   - User2 registers the same domain "tommy" for 1 year, with `tokenId` now being, let's say, 57. The mappings are updated to reflect User2's ownership with `tokenId` 57. However, the old mapping for `tokenId` 3 to "tommy" is not cleared or updated, leading to:
     - `_token_id_to_domain[3]` still pointing to "tommy".
     - `_token_id_to_domain[57]` now also pointing to "tommy".
   - New domain details for User2:

```
DomainDetails {
    handler: tommy,
    resolver: user2,
    token_id: 57,
    expiry_date: user2Expiry,
    last_transfer_time: now,
};
```

4. **Transfer**
   - User1, who still holds `tokenId` 3, can execute `transferFrom()` with this old `tokenId` because:
     (a) The function checks if the caller is the owner of the `tokenId` (which User1 is for `tokenId` 3).
     (b) This allows User1 to update the domain details for "tommy" to point to another address they control (let's say User1'), effectively transferring control of the domain without paying renewal fees.
   - New domain details after transfer:

```
DomainDetails {
    handler: tommy,
    resolver: user1',
    token_id: 3,
    expiry_date: user2Expiry,
    last_transfer_time: now,
};
```

   - Conversely, since User2 owns `tokenId` 57, they can also use a similar process to transfer the domain back to another secondary address, creating a never-ending ambiguity of domain ownership.

Recommendation(s): Consider implementing a mechanism to maintain unique `tokenIds` for each domain, even when repurchased after expiration.

Status: Fixed

Update from the client: Fixed in this revision.

6.2 Low

6.2.1 Primary Domain Gets Reset on Transfer When User Owns Multiple Domains

File(s): brother_id.cairo

Description: When a user registers a domain, it automatically becomes their primary domain. However, during the transfer of a domain, the primary domain is reset to zero. This behavior is useful when a user owns only one domain and intends to transfer it.

Issues arise when a user possesses more than one domain. For instance, consider two users, User1 and User2, where:

- User1 owns domain `x`, and User2 owns domain `y`, both being their respective primary domains.
- If User1 transfers `x` to User2, User2 now holds two domains: `y` and `x`, with `y` remaining as the primary domain.
- The issue occurs if User2 then decides to transfer domain `x` (which is not their primary since they did not register it) to another user. This action inadvertently resets User2's primary domain, `y`, to zero, despite `y` not being the domain transferred.

```
fn _transfer(
    ref self: ContractState, from: ContractAddress, to: ContractAddress, token_id: u256
) {
    .....
    // @audit >> Resets primary domain on transfer if user has multiple domains
    self._userPrimaryDomain.write(from, 0);
    .....
}
```

Although the user can manually correct this by calling the `setPrimary()` function to reassign their primary domain, this behavior disrupts the user experience.

Recommendation(s): Consider implementing a mechanism for uniquely managing the primary domain during transfers to prevent unintended resets.

Status: Fixed

Update from the client: Fixed in this revision.

6.2.2 Domain Renewal Overwrites Remaining Registration Period

File(s): brother_id.cairo

Description: The current implementation of the function `renew_domain()` calculates the new expiry date from the current block timestamp when a domain is renewed.

If a user renews a domain before its original expiry date, the remaining time on the current registration is not added to the new expiry date. This effectively ignores the time already paid for. Ideally, the remaining registration period should be considered during renewal.

```
fn renew_domain(ref self: ContractState, domain: felt252, years: u8) {
    .....
    // @audit >> Doesn't account for remaining registration period
    let expiry = now + (31556926 * years.into());
    let newDomainDetails = DomainDetails {
        handler: domainInfo.handler,
        resolver: domainInfo.resolver,
        token_id: domainInfo.token_id,
        expiry_date: expiry,
        last_transfer_time: domainInfo.last_transfer_time,
    };
    .....
}
```

Recommendation(s): Modify the logic so that, if renewal is performed before the expiry date, the remaining time is added to the new expiry period.

Status: Fixed

Update from the client: Fixed in this revision.

6.3 Informational

6.3.1 Expired Domains Can Be Transferred

File(s): brother_id.cairo

Description: The current system allows for the transfer of an expired domain using the `transferFrom()` or `safeTransferFrom()` functions. During the transfer, the `tokenId` associated with the domain is also transferred.

Since an expired domain can be repurchased, this setup leads to scenarios where a single domain can be linked to multiple `tokenIds`, which can result in unexpected behavior.

```
fn _transfer(
    ref self: ContractState, from: ContractAddress, to: ContractAddress, token_id: u256
) {
    .....
    // @audit >> No check for expired domain
    let old_domain_details: DomainDetails = self.get_domain_by_tokenId(token_id);
    .....
}
```

Recommendation(s): Add checks to ensure that expired domains cannot be transferred.

Status: Fixed

Update from the client: Fixed in this revision.

6.4 Best Practices

6.4.1 Unused Storage Variable

File(s): brother_id.cairo

Description: The state variable `_is_domain_registered` remains unused. Although likely introduced for tracking registered domains, it duplicates the functionality already implemented by the function `get_is_domain_available()`.

```
#[storage]
struct Storage {
   ....
   _is_domain_registered: LegacyMap<felt252, bool>,
   ....
}
```

Recommendation(s): Consider removing this storage variable.

Status: Fixed

Update from the client: Fixed in this revision.

6.4.2 Unchecked Return Value of ERC20 `transferFrom()` in `register_domain()` & `renew_domain()`

File(s): brother_id.cairo

Description: The functions `register_domain()` and `renew_domain()` use the ERC20 function `transferFrom()`. This function returns a boolean value indicating whether the transfer was successful. However, the current implementation does not check this returned value, which could potentially lead to unexpected behavior.

```
fn register_domain(
    ref self: ContractState, domain: felt252, years: u8, resolver: ContractAddress
) {
    .....
    // @audit >> Unchecked return
    eth_contract().transferFrom(get_caller_address(), treasury, price);
    .....
}

fn renew_domain(ref self: ContractState, domain: felt252, years: u8) {
    .....
    // @audit >> Unchecked return
    eth_contract().transferFrom(get_caller_address(), treasury, price);
    .....
}
```

Recommendation(s): Consider implementing a check for the boolean return value to ensure the transfer was successful.

Status: Fixed

Update from the client: Fixed in this revision.

6.4.3 Unnecessary Caller Check

File(s): brother_id.cairo

Description: The functions burn() and assert_only_owner() include a check to ensure the caller is not the zero address. However, in a mainnet or testnet environment, the caller address can never be zero, making this check redundant.

```
fn assert_only_owner(self: @ContractState) {
    .....
    let caller: ContractAddress = get_caller_address();
    assert(!caller.is_zero(), 'Caller is the zero address');
    .....
}

fn burn(ref self: ContractState, tokenId: u256) {
    .....
    let caller = get_caller_address();
    assert(!caller.is_zero(), 'Caller is the zero address');
    .....
}
```

Recommendation(s): Consider removing the check for the zero address since get_caller_address() returns a non-zero address.

Status: Fixed

Update from the client: Fixed in this revision.

6.4.4 Unused Code

File(s): brother_id.cairo

Description: The contract implements certain code and logic which is not being used. These include:

1. The statement let now = get_block_timestamp(); in the function get_is_domain_available() is not used.

```
fn get_is_domain_available(self: @ContractState, domain: felt252) -> bool {
    .....
    let now = get_block_timestamp();
    .....
}
```

2. The internal function _safe_mint() remains unused.

```
fn _safe_mint(
    ref self: ContractState, to: ContractAddress, token_id: u256, data: Span<felt252>
) {
    self._mint(to, token_id);
    assert(
        _check_on_erc721_received(Zeroable::zero(), to, token_id, data),
        Errors::SAFE_MINT_FAILED
    );
}
```

3. In the function renew_domain(), an unnecessary operation updates the mapping from tokenId to domain, even though this relationship does not change during domain renewal.

```
fn renew_domain(ref self: ContractState, domain: felt252, years: u8) {
    .....
    self._token_id_to_domain.write(domainInfo.token_id, domain);
    .....
}
```

Recommendation(s): Consider removing these unused code/statements.

Status: Fixed

Update from the client: Fixed in this revision.

6.4.5 Confusing Function Name `eth_contract()`

File(s): brother_id.cairo

Description: In the `register_domain()` and `renew_domain()` functions, `eth_contract()` is called, which returns an `IERC20CamelDispatcher` pointing to the address `0x03b405a98c9e795d427fe82cdeeeed803f221b52471e3a757574a2b4180793ee`. Despite its name suggesting it's for ETH, this address is actually for the Starknet Brother token deployed on the Starknet mainnet, which can lead to confusion.

```
// @audit >> Confusing function name
fn eth_contract() -> IERC20CamelDispatcher {
    IERC20CamelDispatcher {
        contract_address: contract_address_const::<
            0x03b405a98c9e795d427fe82cdeeeed803f221b52471e3a757574a2b4180793ee
        >(),
    }
}
```

Recommendation(s): Consider renaming the function to better suit the actual token it references.

Status: Fixed

Update from the client: Fixed in this revision.

7 Compilation Output

```
scarb build
   Compiling lib(pyramidlp) pyramidlp v0.7.0 (/brotherid/Scarb.toml)
   Compiling starknet-contract(pyramidlp) pyramidlp v0.7.0 (/brotherid/Scarb.toml)
    Finished release target(s) in 24 seconds
```

8 Tests Output

```
    NO TESTS PROVIDED
```