Security Audit and Analysis

Starknet.Id

This report details a security analysis of the naming and identity contracts within the Starknet.Id protocol, conducted in April-May 2024 by **Subsix (Anubhav Kumar)**. The analysis aims to identify and assess potential security vulnerabilities within the protocol contracts.

Blockchain network: **Starknet**
Smart contract language: **Cairo**
Testing suite: **starknet-foundry**

Subsix_
anubhav11697@gmail.com
sub_six

Table of Contents

Disclaimer 2
Protocol Overview 3
Risk Classification 4
Audit Summary 5
Security Findings 6

Disclaimer

This report is provided for informational purposes only. It does not constitute investment advice, financial guidance, or an endorsement of the protocol. The analysis, conducted as a **shadow smart contract audit**, aims to identify potential vulnerabilities and security issues within the reviewed smart contract code. However, it's crucial to understand that:

- **Smart contracts are immutable by design.** Once deployed on a blockchain, their code cannot be altered. This immutability is a core strength for trust and transparency, but it also means that any vulnerabilities discovered after deployment are permanent.
- **No audit can guarantee absolute security or functionality.** Due to the immutable nature of smart contracts and the ever-evolving landscape of security threats, there's always a possibility of undiscovered vulnerabilities or unforeseen interactions with other parts of the code.
- **This analysis is not a comprehensive assurance of the code's integrity.** It is important to acknowledge that even a comprehensive examination cannot definitively guarantee the absence of all potential vulnerabilities within the code. The inherent complexity of smart contracts, coupled with the ever-evolving threat landscape, necessitates acknowledging this limitation.

Smart contract exploits can sometimes originate from internal actors with authorized access. This analysis primarily focuses on identifying vulnerabilities exploitable by unauthorized external actors. Users of the Starknet.Id protocol are solely responsible for conducting their own due diligence and security assessments. This includes evaluating the identified vulnerabilities (if any) within the context of their specific use case. **Subsix (Anubhav Kumar)** accepts no liability for any losses or damages arising from the use of this report or the Starknet.ID smart contracts.

Protocol Overview

Starknet.ID is an identity management protocol built on the Starknet. It enables users to create portable, on-chain identities that can be used across various Starknet applications. These identities can store user-controlled data and link to human-readable ".stark" domains, simplifying authentication and fostering a more user-friendly experience within the Starknet ecosystem. Smart contracts in the scope are **upgradeable** by the admin of the contract. The current design employs a **single administrator**, which grants this admin access to a specific set of **11** functions within the smart contracts.

Protocol Name	Starknet.Id
Language	Cairo
Codebase	https://github.com/starknet-id/identity https://github.com/starknet-id/naming
Commit(s)	Initial Commits: https://github.com/starknet-id/identity/commit/183cb4f52e62ddac49d1dc5e5c3f228794c93741 https://github.com/starknet-id/naming/commit/df8c9c83f4882418be7511a8b4e5dd8c755b51e6 Final Commits: https://github.com/starknet-id/identity/commit/c8734d2ac2893e162ad9a2e506a0ce61ac36d3f9 https://github.com/starknet-id/naming/commit/8abe7e0e76947b88f45e61016d5d416c94366233

Risk Classification

Severity Level	Impact : High	Impact : Medium	Impact: Low
Likelihood : High	Critical	High	Medium
Likelihood : Medium	High	Medium	Low
Likelihood : Low	Medium	Low	Low

Impact

High : Leads to a loss of a significant portion of assets in the protocol, or significant harm to a majority of users, loss of ownership of the contract.

Medium : Global loss is not significant or losses to only a subset of users, core functionality of the protocol is affected.

Low : Issues which can cause little to no damage, bugs that are easily recoverable, leads to any kind of unexpected behavior with some of the protocol functionalities that are not critical.

Likelihood

High : Almost certain to happen, easy to perform, or not as easy but highly incentivized.

Medium : Only conditionally possible or incentivized, but still relatively likely.

Low : Requires stars to align, or little-to-no incentive

To address issues that do not fit High/Medium/Low severity, I have used two more finding severities: **Informational** and **Best Practices**.

Informational : Findings do not pose any risk to the application, but they carry some information that should be known.

Best Practice : Findings are used when some piece of code does not conform with smart contract development best practices.

Audit Summary

This report details the findings from a manual security review of Starknet.ID done by me. There are a total **11 findings**, 1 low, 3 informational and 7 best practices.

ID	Title	Severity
L-01	Missing “id” check in buy() can lead to domain getting stolen or misuse of naming contract	Low
I-O1	Potential front-running in buy() & altcoin_buy()	Informational
I-02	Unchecked return value of ERC20 transfer() & transferFrom() in pay_domain() & claim_balance()	Informational
I-03	Renewal functionality allows renewal of domains even if they haven't been purchased yet	Informational
BP-01	Unused storage variable	Best Practices
BP-02	Missing zero check for admin address update	Best Practices
BP-03	Should early revert in renew() & atlcoin_renew()	Best Practices
BP-04	Redundant authorization checks	Best Practices
BP-05	Unnecessary type casting to felt252	Best Practices
BP-06	Caching get_caller_address to reduce redundancy	Best Practices
BP-07	Unused Imports	Best Practices

Security Findings

[L-01] Missing “id” check in buy() can lead to domain getting stolen or misuse of naming contract

File(s) : naming/main.cairo

Description : The buy() function accepts an “id” parameter. However, they lack validation to ensure a Starknet identity has been minted for this specific “id”. If a user attempts to purchase a domain using an unminted “id”, the purchase will be successful, processing the payment and assigning the domain to the “id”. Since the “id” doesn't exist yet, a malicious actor could exploit this vulnerability by discovering the “id” and calling the identity contract’s mint() function with that “id”. This would grant the attacker ownership of the domain, essentially stealing it from the intended user without any cost.

```
fn buy(
    ref self: ContractState,
    id: u128, // @audit >>> [ Low >> Missing validation check ]
    domain: felt252,
    days: u16,
    resolver: ContractAddress,
    sponsor: ContractAddress,
    discount_id: felt252,
    metadata: felt252,
) {
```

Potential for misuse :

While the direct exploit for this misuse is highly unlikely, this vulnerability does create a potential avenue for a more elaborate scam. If a domain is purchased with an unminted ID, it initially resolves to address zero. Although the buyer wouldn't truly own the domain and wouldn't be able to transfer it, a malicious actor could potentially leverage this to misuse the naming contract through an independent setup, (outside the StarknetID UI). This setup could involve tricking users into believing they own the domain by:

1. Allowing the user to buy the domain with an unminted ID.
2. The attacker then mints an identity for the above ID.
3. The attacker calls **Identity.set_user_data(id, 'starknet', 'user_address')** to ensure the domain resolves to the user's address."

By doing this, users are deceived into thinking they've bought a domain, but they never truly own it—the attacker maintains ownership.

Recommendation(s) : Consider implementing a check within the buy() function to verify that the "id" being used is minted or that a Starknet identity exists for the given "id". Identity contract ERC721 internal *_exists()* function can be used for this purpose.

[I-01] Potential front-running in buy() & altcoin_buy()

File(s) : naming/main.cairo

Description : In the current system, acquiring a domain involves a two-step process: first, the user mints an identity (a number passed as input to *mint()*). Then, as the second step, users buy the domain using this identity. Once a domain is bought, it resolves to the owner of the identity. However, this two-step process could potentially create a front-running case where an attacker could exploit this vulnerability to steal or buy domains for free from users.

Currently, front-running is not possible on Starknet because transaction execution follows a FIFO sequential order. However, as Starknet plans to decentralize the sequencer and there's a possibility of a fees market being created, the sequencer will sort its mempool by profitability and construct blocks accordingly.

In this scenario, an attacker can observe the mempool and identify which ID is being used for the user's mint(ID) call. Subsequently, the attacker can formulate their own mint() call with the same ID as the user's and price it higher for execution before the user's mint() can be executed. When the user's buy() function is subsequently executed after minting, it assigns ownership of the domain to the specified ID. By effectively placing the attacker's mint call before the user's mint() call, the attacker is able to steal the domain for free.

Recommendation(s) : Consider adding front-running protection. This vulnerability poses a risk, particularly in light of future developments in the Starknet ecosystem, and should be addressed to ensure the security of the protocol.

[I-02] Unchecked return value of ERC20 transfer() & transferFrom() in pay_domain() & claim_balance()

File(s) : naming/main.cairo, naming/internal.cairo

Description : pay_domain() and claim_balance() uses ERC20 transferFrom() and transfer() functions respectively. These functions return a boolean value indicating whether the transfer was successful. However, the current implementation doesn't check the returned value, potentially resulting in unexpected behavior, particularly for tokens that do not revert on failure.

```
// @audit >> [ Info >> Unchecked return value ]
IERC20CamelDispatcher { contract_address: erc20 }
    .transferFrom(get_caller_address(), get_contract_address(), discounted_price);
```

```
// @audit >> [ Info >> Unchecked return value ]
IERC20CamelDispatcher { contract_address: erc20 }
    .transfer(get_caller_address(), balance);
```

Recommendation(s) : Consider implementing a check for boolean return value of above functions.

[I-03] Renewal functionality allows renewal of domains even if they haven't been purchased yet

File(s) : naming/main.cairo

Description : The renew() and altcoin_renew() functions are responsible for renewing domains, but these functions don't check whether the given domain is purchased or not and the renew logic gets executed. This oversight could lead to a waste of funds if someone attempts to renew a domain not possessed by anyone. Additionally, it emits an event that the offchain might monitor, potentially leading to inaccurate data interpretation.

```
fn renew(
    ref self: ContractState,
    domain: felt252, // @audit >>> [ Info >> Missing check for domain existence ]
    days: u16,
    sponsor: ContractAddress,
    discount_id: felt252,
    metadata: felt252,
) {
```

```
fn altcoin_renew(
    ref self: ContractState,
    domain: felt252, // @audit >>> [ Info >> Missing check for domain existence ]
    days: u16,
    sponsor: ContractAddress,
    discount_id: felt252,
    metadata: felt252,
    altcoin_addr: ContractAddress,
    quote: Wad,
    max_validity: u64,
    sig: (felt252, felt252),
) {
```

Recommendation(s) : Consider implementing a check to make sure only purchased domains can be renewed. Can use domain-hash for this purpose.

[BP-01] Unused storage variable

File(s) : identity/main.cairo

Description : The state variable Proxy_admin remains unused. Although likely introduced for ownership management, it duplicates the functionality already provided by the OpenZeppelin Ownable component utilized by the identity contract. The value stored in this variable is identical to the value stored within the Ownable component's owner variable.

```
Proxy_admin: felt252, // @audit >>> [ BP >> Unused storage variable ]
```

Recommendation(s) : Consider removing this storage variable.

[BP-02] Missing zero check for admin address update

File(s) : naming/main.cairo

Description : The set_admin() function updates the admin of the naming contract. However, it fails to check if the new_admin address is set to zero. This oversight could lead to a catastrophic issue if new_admin is accidentally set to the zero address. The same oversight also applies when setting the _admin_address in the naming contract constructor.

```
// @audit >>> [ BP >> Missing zero check for  new_admin ]
fn set_admin(ref self: ContractState, new_admin: ContractAddress) {
    assert(get_caller_address() == self._admin_address.read(), 'you are not admin');
    self._admin_address.write(new_admin);
}
```

Recommendation(s) : Consider implementing a check for the zero address. Or, Just as the identity contract utilizes the OpenZeppelin Ownable component for ownership management, the naming contract can adopt a similar approach.

[BP-03] <u>Should early revert in renew() & atlcoin_renew()</u>

File(s) : naming/main.cairo

Description : In the renew() and altcoin_renew() functions, the expiry calculation is determined based on given days, and subsequently asserted to be within a specified range. However, these checks are currently positioned after several significant internal calls. If all the internal calls passed and the assertion failed due to the specified days and expiry not falling within the expected range, gas could be expended until the point where the transaction failed.

```
// @audit >>> [ BP >> Can be moved up :: Early Revert ]
// find new domain expiry
let new_expiry = if domain_data.expiry <= now {
    now + 86400 * days.into()
} else {
    domain_data.expiry + 86400 * days.into()
};
// 25*365 = 9125
assert(new_expiry <= now + 86400 * 9125, 'purchase too long');
assert(days >= 6 * 30, 'purchase too short');
```

Recommendation(s) : It is recommended to relocate these checks and assertions to occur earlier in the function after the domain_data is obtained. This adjustment would optimize gas usage, particularly in cases where reverts occur in the subsequent internal calls.

[BP-04] Redundant authorization checks

File(s) : naming/main.cairo

Description : The naming contract exhibits code redundancy in the form of repeated authorization checks. Specifically, the following line is found in 10 different functions :

```
// @audit >>> [ BP >> Redundant code ]
assert(get_caller_address() == self._admin_address.read(), 'you are not admin');
```

Recommendation(s) : Consider refactoring the code by implementing a common function to handle authorization checks or integrating the OpenZeppelin Ownable component for streamlined access control management.

[BP-05] Unnecessary type casting to felt252

File(s) : identity/main.cairo

Description : The function get_extended_verifier_data includes a basic inefficiency. It accepts a parameter "length" of type felt252. However, when calling the internal function get_extended, the "length" parameter is explicitly cast to felt252 using the into() method. This type casting is unnecessary because "length" is already of the expected type felt252.

```
fn get_extended_verifier_data(
    self: @ContractState,
    id: u128,
    field: felt252,
    length: felt252,
    verifier: ContractAddress,
    domain: u32
) -> Span<felt252> {
    self
        .get_extended(
            VERIFIER_DATA_ADDR,
            array![id.into(), field, verifier.into()].span(),
            length
                .into(), // @audit >>> [ BP >> Unneccesary type casting to felt252 ]
            domain,
        )
}
```

Recommendation(s) : Consider removing the unnecessary type casting by simply passing “length” directly to the get_extended function call.

[BP-06] Caching get_caller_address to reduce redundancy

File(s) : identity/main.cairo

Description : The reset_main_id function calls get_caller_address twice. This is unnecessary as the caller's address stays the same within the function.

```
// @audit >>> [ BP >> Can cache get_caller_address ]
fn reset_main_id(ref self: ContractState) {
    let id = self.main_id_by_addr.read(get_caller_address());
    self.main_id_by_addr.write(get_caller_address(), 0);
    self
        .emit(
            Event::MainIdUpdate(MainIdUpdate { id, owner: ContractAddressZeroable::zero() })
        );
}
```

Recommendation(s) : Consider caching get_caller_address() in a local variable upon retrieval and use that variable throughout the function.

[BP-07] <u>Unused Imports</u>

Identity Module

File : identity/main.cairo

```
// @audit >>> [ BP >> Unused Imports ]
use integer::{u256_safe_divmod, u256_as_non_zero};
use core::pedersen;
use starknet::get_contract_address;
```

Naming Module

File : naming/asserts.cairo

```
// @audit >>> [ BP >> Unused imports ]
use integer::{u256_safe_divmod, u256_as_non_zero};
use naming::{
    interface::{
        naming::{INaming, INamingDispatcher, INamingDispatcherTrait},
        resolver::{IResolver, IResolverDispatcher, IResolverDispatcherTrait},
        pricing::{IPricing, IPricingDispatcher, IPricingDispatcherTrait},
        referral::{IReferral, IReferralDispatcher, IReferralDispatcherTrait},
    }
};
```

File : naming/internal.cairo

```
use naming::{
    interface::{
        naming::{INaming, INamingDispatcher, INamingDispatcherTrait}, // @audit >>> [ BP >> Unused imports ]
        resolver::{IResolver, IResolverDispatcher, IResolverDispatcherTrait},
        pricing::{IPricing, IPricingDispatcher, IPricingDispatcherTrait}, // @audit >>> [ BP >> Unused imports ]
        referral::{IReferral, IReferralDispatcher, IReferralDispatcherTrait},
    },
};
use starknet::{
    contract_address::ContractAddressZeroable, ContractAddress, get_caller_address,
    get_contract_address,
    get_block_timestamp // @audit >>> [ BP >> Unused imports >> get_block_timestamp ]
};
```

File : naming/utils.cairo

```
use wadray::{Wad, WAD_SCALE}; // @audit >>> [ BP >> Unused imports >> WAD_SCALE ]
```

File : naming/main.cairo

```
use integer::{u256_safe_divmod, u256_as_non_zero}; // @audit >>> [ BP >> Unused imports ]
use naming::{
    naming::{asserts::AssertionsTrait, internal::InternalTrait, utils::UtilsTrait},
    interface::{
        naming::{INaming, INamingDispatcher, INamingDispatcherTrait},
        // @audit >>> [ BP >> Unused import ]
        resolver::{
            IResolver, IResolverDispatcher, IResolverDispatcherTrait
        },
        pricing::{IPricing, IPricingDispatcher, IPricingDispatcherTrait},
        // @audit >>> [ BP >> Unused import ]
        referral::{
            IReferral, IReferralDispatcher, IReferralDispatcherTrait
        },
    }
};
use clone::Clone; // @audit >>> [ BP >> Unused imports ]
use array::ArrayTCloneImpl; // @audit >>> [ BP >> Unused imports ]
```

File : interface/resolver.cairo

```
use starknet::ContractAddress; // @audit >>> [ BP >> Unused imports ]
```

Recommendation(s) : Consider removing these imports.