Security Review Report
NM-0135 STARKNET ID

NETHERMIND
SECURITY

(Oct 4, 2023)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the Starknet ID Auto Renew Contract. The reviewed contracts implement an automatic renewal feature for domain ownership, ensuring a seamless and continuous user experience.

The audited code consists of 245 lines of Cairo code. The audit was performed using (a) manual analysis of the codebase, (b) simulation of the smart contract, and (c) creation of test cases. Along this document, we report 10 points of attention, where one is classified as `High`, three are classified as `Low`, and six are classified as `Best Practices` or `Informational`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

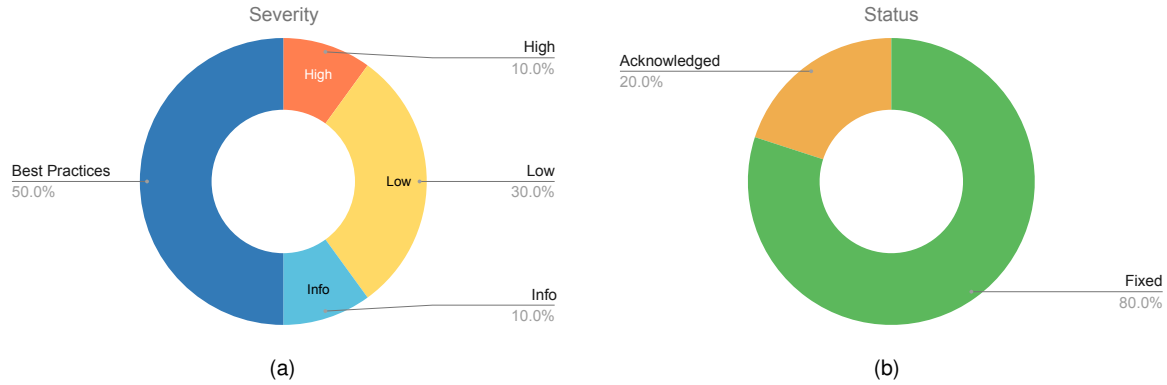

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (0), **Low** (3), **Undetermined** (0), **Informational** (1), **Best Practices** (5). **Distribution of status: Fixed** (8), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Aug. 24, 2023
Response from Client	Sep 30, 2023
Final Report	Oct 4, 2023
Methods	Manual Review, Automated Analysis
Repository	Starknet ID
Commit Hash (Audit)	71f58bca694ece9670a1148955e07d9cfdcf76c8
Commit Hash (Reaudit)	e1830508e0c310dc390684cc5cfc7517484bf2f4 5284970d3df79836c21debfe1cbd11ad3694bf52 8bc436b8a848354b21bab1785c06af1d83dc846d 1ffcf572894514e14c20f0e973ca6fe2b4d05d5a 9ef5b7cd44d67ae3799eba8989d571eea9ab0527
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/lib.cairo	2	1	50.0%	0	3
2	src/auto_renewal.cairo	243	99	40.7%	43	385
	Total	**245**	**100**	**40.8%**	**43**	**388**

3 Summary of Issues

	Finding	Severity	Update
1	Function `update_tax_contract(...)` is changing the contract owner	High	Fixed
2	Not using a two-step process for transferring ownership	Low	Fixed
3	Possible unnecessary spending of user funds	Low	Fixed
4	`tax_price` can be greater than `limit_price`	Low	Fixed
5	Parameter `metadata` can be changed by a whitelisted caller.	Info	Acknowledged
6	Generic error messages for pop operations	Best Practices	Fixed
7	Missing `tax_price` and `metadata` in the event `DomainRenewed`	Best Practices	Fixed
8	Missing events emission	Best Practices	Fixed
9	Storage variable `_is_renewing` can be more optimized.	Best Practices	Fixed
10	Storage variables can be constant	Best Practices	Acknowledged

4 System Overview

The audit covers the `AutoRenewal` smart contract. This contract manages domain renewals for the `Starknet.id` system. The contract emits the following events.

```
#[event]
#[derive(Drop, starknet::Event)]
enum Event {
    EnabledRenewal: EnabledRenewal,
    DisabledRenewal: DisabledRenewal,
    DomainRenewed: DomainRenewed,
}

#[derive(Drop, starknet::Event)]
struct EnabledRenewal {
    #[key]
    domain: felt252,
    renewer: ContractAddress,
    limit_price: u256,
    meta_hash: felt252,
}

#[derive(Drop, starknet::Event)]
struct DisabledRenewal {
    #[key]
    domain: felt252,
    renewer: ContractAddress,
    limit_price: u256,
}

#[derive(Drop, starknet::Event)]
struct DomainRenewed {
    #[key]
    domain: felt252,
    renewer: ContractAddress,
    days: felt252,
    limit_price: u256,
    timestamp: u64,
}
```

The contract uses the following imports.

```
use starknet::ContractAddress;
use starknet::{get_caller_address, get_contract_address, get_block_timestamp};
use starknet::contract_address::ContractAddressZeroable;
use traits::{TryInto, Into};
use option::OptionTrait;
use array::ArrayTrait;
use integer::u64_try_from_felt252;
use debug::PrintTrait;
use openzeppelin::token::erc20::interface::{IERC20CamelDispatcher, IERC20CamelDispatcherTrait};
use naming::interface::naming::{INamingDispatcher, INamingDispatcherTrait};
```

And the following storage variables.

```
#[storage]
struct Storage {
    naming_contract: ContractAddress,
    erc20_contract: ContractAddress,
    tax_contract: ContractAddress,
    admin: ContractAddress,
    whitelisted_renewer: ContractAddress,
    can_renew: bool,
    // (renewer, domain, limit_price) -> 1 or 0
    _is_renewing: LegacyMap::<(ContractAddress, felt252, u256), bool>,
    // (renewer, domain) -> timestamp
    last_renewal: LegacyMap::<(ContractAddress, felt252), u64>,
}
```

The contract has the following getters.

```
fn is_renewing(
    self: @ContractState, domain: felt252, renewer: ContractAddress, limit_price: u256
) -> bool {
    self._is_renewing.read((renewer, domain, limit_price))
}
```

```
fn get_contracts(
    self: @ContractState
) -> (ContractAddress, ContractAddress, ContractAddress) {
    (self.naming_contract.read(), self.erc20_contract.read(), self.tax_contract.read())
}
```

The function `enable_renewals(...)` allows the `caller` to renew a given `domain` given a `limit_price`. The function is shown below.

```
fn enable_renewals(ref self: ContractState, domain: felt252, limit_price: u256, meta_hash: felt252)
{
    let caller = get_caller_address();
    self._is_renewing.write((caller, domain, limit_price), true);
    // we erase the previous renewal date
    self.last_renewal.write((caller, domain), 0);
    self.emit( Event::EnabledRenewal( EnabledRenewal { domain, renewer: caller, limit_price, meta_hash }))
}
```

The `caller` can also disable renewals for a given `domain` and `limit_price` by calling the function below.

```
fn disable_renewals(ref self: ContractState, domain: felt252, limit_price: u256) {
    let caller = get_caller_address();
    self._is_renewing.write((caller, domain, limit_price), false);

    self.emit( Event::DisabledRenewal( DisabledRenewal { domain, renewer: caller, limit_price, } ))
}
```

Function **renew(...)**: Only the `whitelisted_renewer` can call this function for domain renewals. The function applies several checks to evaluate if the current contract is not disabled, the domain can be renewed, the domain is not renewed yet in the current year, and the domain expires within a month. The function also updates when the last renewal was applied to avoid charging more than once a year. Then, the flow capacity (limit price amount) is taken from the user account and transferred to the contract. The tax price is transferred to the tax contract, and the expiration date is renewed for the next year.

```
fn renew( ref self: ContractState, root_domain: felt252, renewer: ContractAddress,
    limit_price: u256, tax_price: u256, metadata: felt252,
    ) {
    assert(self.can_renew.read(), 'Contract is disabled');
    assert( get_caller_address() == self.whitelisted_renewer.read(), 'You are not whitelisted' );
    self._renew(root_domain, renewer, limit_price, tax_price, metadata);
}
```

```
fn _renew( ref self: ContractState, root_domain: felt252, renewer: ContractAddress,
    limit_price: u256, tax_price: u256, metadata: felt252,
) {
    let naming = self.naming_contract.read();
    let can_renew = self._is_renewing.read((renewer, root_domain, limit_price));

    assert(can_renew, 'Renewal not toggled for domain');

    // Check domain has not been renew yet this year
    let block_timestamp = get_block_timestamp();
    let last_renewed = self.last_renewal.read((renewer, root_domain));

    // 364 because we keep adding one day margin to the existing month,
    // if we take more than a day to renew, the margin will shrink.
    assert(block_timestamp - last_renewed > 86400_u64 * 364_u64, 'Domain already renewed');

    // Check domain is set to expire within a month
    let expiry: u64 = INamingDispatcher { contract_address: naming }
        .domain_to_data(array![root_domain].span())
        .expiry;
    assert(expiry <= block_timestamp + (86400_u64 * 30_u64), 'Domain not set to expire');
```

```
    // Renew domain
    // last_renewal is updated before external contract calls to prevent reentrancy attacks
    // if the naming contract was compromised
    self.last_renewal.write((renewer, root_domain), block_timestamp);
    // events are sent before calls to other contracts to prevent a reordering via reentrancy attack
    self.emit( Event::DomainRenewed( DomainRenewed { domain: root_domain, renewed, days: 365, limit_price, timestamp:
    ↪  block_timestamp } ) );
    let contract = get_contract_address();
    let erc20 = self.erc20_contract.read();
    let _tax_contract = self.tax_contract.read();

    // Transfer limit_price (including tax), will be canceled if the tx fails
    IERC20CamelDispatcher { contract_address: erc20 }.transferFrom(renewer, contract, limit_price);
    // transfer tax price to the tax contract address
    IERC20CamelDispatcher { contract_address: erc20 }.transfer(_tax_contract, tax_price);
    //Spend the remaining money to renew the domain
    //If something remains after this, it can be considered as lost by the user,
    //We keep the ability to claim it back but can't guarantee we will do it
    INamingDispatcher { contract_address: naming }.renew(root_domain, 365_u16, ContractAddressZeroable::zero(), 0,
    ↪  metadata);
}
```

The function **batch_renew(...)** : Only the `whitelisted_renewer` can call this function for batch domain renewals. The function applies the same checks described in `renew(...)`. The difference relies only on the number of domains to be renewed.

```
fn batch_renew( ref self: ContractState, domain: array::Span::<felt252>, renewer:
↪  array::Span::<starknet::ContractAddress>, limit_price: array::Span::<u256>, tax_price: array::Span::<u256>,
↪  metadata: array::Span::<felt252>,
) {
    assert(self.can_renew.read(), 'Contract is disabled');
    assert(
        get_caller_address() == self.whitelisted_renewer.read(), 'You are not whitelisted'
    );
    assert(domain.len() == renewer.len(), 'Domain & renewer mismatch len');
    assert(domain.len() == limit_price.len(), 'Domain & price mismatch len');

    let mut domain = domain;
    let mut renewer = renewer;
    let mut limit_price = limit_price;
    let mut tax_price = tax_price;
    let mut metadata = metadata;

    loop {
        if domain.len() == 0 {
            break;
        }
        let _domain = domain.pop_front().expect('pop_front error');
        let _renewer = renewer.pop_front().expect('pop_front error');
        let _limit_price = limit_price.pop_front().expect('pop_front error');
        let _tax_price = tax_price.pop_front().expect('pop_front error');
        let _metadata = metadata.pop_front().expect('pop_front error');
        self._renew(*_domain, *_renewer, *_limit_price, *_tax_price, *_metadata);
    }
}
```

The function **claim(...)** : Only the `admin` can call this function to withdraw funds deposited in the contract.

```
fn claim(ref self: ContractState, amount: u256) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    let erc20 = self.erc20_contract.read();
    IERC20CamelDispatcher { contract_address: erc20 }.transfer(get_caller_address(), amount);
}
```

The contract also has the following update functions.

```
// Admin function to update admin address and the tax contract address
fn update_admin(ref self: ContractState, new_admin: ContractAddress,) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    self.admin.write(new_admin);
}

fn update_tax_contract(ref self: ContractState, new_addr: ContractAddress,) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    self.admin.write(new_addr);
}

fn update_whitelisted_renewer(ref self: ContractState, whitelisted_renewer: starknet::ContractAddress) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    self.whitelisted_renewer.write(whitelisted_renewer);
}

fn toggle_off(ref self: ContractState) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    self.can_renew.write(false);
}
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. 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 the finding were to be exploited by an attacker. 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.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

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

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

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Function `update_tax_contract(...)` is changing the contract owner

File(s): auto_renewal.cairo

Description: The function `update_tax_contract(...)` is supposed to update the address of the tax contract. However, the function updates the admin address instead of the tax contract. Since the contract has no two-step process for transferring ownership, you effectively renounce or relinquish administrative control over the smart contract. If this happens in production, you remove the ability to execute administrative functions permanently and irreversibly. The function is reproduced below.

```
fn update_tax_contract(ref self: ContractState, new_addr: ContractAddress,) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    ////////////////////////////////////////////////////////////////////////
    // @audit Wrong storage variable. Should be "self.tax_contract.write(new_addr);"
    ////////////////////////////////////////////////////////////////////////
    self.admin.write(new_addr);
}
```

Recommendation(s): Update the function as shown below.

```
fn update_tax_contract(ref self: ContractState, new_addr: ContractAddress,) {
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
-    self.admin.write(new_addr);
+    self.tax_contract.write(new_addr);
}
```

Status: Fixed

Update from the client: We updated the code according to the recommendation. Fixed in 9ef5b7cd44d67ae3799eba8989d571eea9ab0527.

6.2 [Low] Not using a two-step process for transferring ownership

File(s): auto_renewal.cairo

Description: The `admin` can be updated using the `update_admin(...)` function. However, the code does not use a two-step process for transferring ownership. A two-step process for transferring ownership in smart contracts enhances security, reduces the risk of errors, and provides a mechanism for verification and reversibility. It is a best practice that adds an extra layer of protection when controlling a contract needs to change hands. Transferring ownership of a smart contract means giving control over its functions and funds to a new entity or address. A two-step process typically involves two transactions: one to initiate the ownership transfer and another to confirm it. This separation allows for verification and validation of the transfer, reducing the risk of accidental or malicious transfers.

The first step usually involves initiating the transfer and confirming the intent, while the second step finalizes the transfer. This adds an extra layer of security because it ensures that the owner has consciously and deliberately chosen to transfer ownership, preventing unauthorized or accidental transfers. A two-step process often includes a time delay between the initiation and confirmation steps. This delay can act as a safeguard against sudden or unexpected transfers. It allows the current owner or other stakeholders to react if the transfer was initiated without proper authorization or in error.

If a mistake is discovered during the confirmation step or if there are concerns about the legitimacy of the transfer, it can be halted before it becomes irreversible. This flexibility is especially valuable when the consequences of an ownership transfer can be significant. The function is reproduced below.

```
// Admin function to update the admin address and the tax contract address
fn update_admin(ref self: ContractState, new_admin: ContractAddress,) {
    ////////////////////////////////////////////////////////////////////
    // @audit Not using a two-step process for transferring ownership
    ////////////////////////////////////////////////////////////////////
    assert(get_caller_address() == self.admin.read(), 'Caller not admin');
    self.admin.write(new_admin);
}
```

Recommendation(s): Adopt a two-step process for transferring ownership.

Status: Fixed

Update from the client: We updated the code to adopt a two-step process via `start_admin_update` and `confirm_admin_update`. Fixed in: e1830508e0c310dc390684cc5cfc7517484bf2f4.

6.3 [Low] Possible unnecessary spending of user funds

File(s): auto_renewal.cairo

Description: The function `_renew` is called by external methods that can only be called by `whitelisted_renewer`, which is a centralized authority. This authority can decide tax prices. However, domain prices can also be changed by the authority on naming contracts if there is any case where that tax price or renewing cost is lowered from when users enable renewals. The user still pays the cost initially, but that amount is stuck in the contract and only claimable by the admin.

```
fn _renew(
    ref self: ContractState, root_domain: felt252, renewer: ContractAddress, limit_price: u256, tax_price: u256,
    ↪ metadata: felt252,
) {
   ...
    ////////////////////////////////////////////////////////////////
    // @audit-issue: The user always debited the amount of `limit_price`
    // but that amount can differ from time to time.
    ////////////////////////////////////////////////////////////////
    IERC20CamelDispatcher { contract_address: erc20 }
        .transferFrom(renewer, contract, limit_price);
    // transfer tax price to tax contract address
    IERC20CamelDispatcher { contract_address: erc20 }.transfer(_tax_contract, tax_price);
    ...
}
```

Recommendation(s): Consider calculating the current price of renewal (`renewal cost + tax price`) and then compare with `limit_price`. If it is lower or equal, then continue the flow and debit the user only the cost.

Status: Fixed

Update from the client: We followed the suggestion.

Fixed in: 1ffcf572894514e14c20f0e973ca6fe2b4d05d5a, 8bc436b8a848354b21bab1785c06af1d83dc846d.

6.4 [Low] `tax_price` can be greater than `limit_price`

File(s): auto_renewal.cairo

Description: The `whitelisted_renewer` charges the renewal for domains by calling the functions `renew(...)` and `batch_renew(...)`. These functions call the private function `_renew(...)` reproduced below.

```
fn _renew(
    ref self: ContractState,
    root_domain: felt252,
    renewer: ContractAddress,
    limit_price: u256,
    tax_price: u256,
    metadata: felt252,
) {
    let naming = self.naming_contract.read();
    let can_renew = self._is_renewing.read((renewer, root_domain, limit_price));
    assert(can_renew, 'Renewal not toggled for domain');
    ...
    IERC20CamelDispatcher { contract_address: erc20 }
        .transferFrom(renewer, contract, limit_price);
    ////////////////////////////////////////////////////////////////
    // @audit ensure that "limit_price" is greater than "tax_price"
    ////////////////////////////////////////////////////////////////
    IERC20CamelDispatcher { contract_address: erc20 }.transfer(_tax_contract, tax_price);
    ...
}
```

The function `_renew(...)` checks if the renewal for the specified `domain` and `limit_price` can be charged. Then, the `limit_price` amount is taken from the `renewer` account to the contract. After that, the `tax_price` is transferred to the `tax_contract`. As we can notice, the function does not check if the `tax_price` is less than or equal to the `limit_price`. When the `tax_price` exceeds the `limit_price`, funds deposited in the contract will be transferred to the `tax_contract`, and the user will not be charged the due amount.

Recommendation(s): Consider checking if `tax_price` is lesser or equal to the amount `limit_price`.

Status: Fixed

Update from the client: We followed the next suggestion, which resolves this one. Fixed in: 1ffcf572894514e14c20f0e973ca6fe2b4d05d5a, and 8bc436b8a848354b21bab1785c06af1d83dc846d.

6.5 [Info] Parameter `metadata` can be changed by a whitelisted caller.

File(s): auto_renewal.cairo

Description: Metadata passed to this function should be the same as users passed value to function `enable_renewals(...)`. However, in this case, the caller can change this value, which can cause a mismatch in sensitive data on the backend.

```
fn renew(
    ref self: ContractState,
    root_domain: felt252,
    renewer: ContractAddress,
    limit_price: u256,
    tax_price: u256,
    metadata: felt252,
) {
    ...
    self._renew(root_domain, renewer, limit_price, tax_price, metadata);
    ////////////////////////////////////////////////////////////////
    // @audit: Metadata is not the one that the user passes
    // while enabling renewal. As we know it is a hash of sensitive
    //  data, but in this case, it is passed to this call via
    //  centralized caller or oracle(whitelisted_caller)
    ////////////////////////////////////////////////////////////////
}
```

Recommendation(s): Consider getting this data directly from the users' parameter. It can be stored in storage and read back in this function. Also, whole data can be stored as a hash.

Status: Acknowledged

Update from the client: We use metadata for accounting. We could decide not to provide the metadata (a hash of private data) given by the user, but that would be a problem for us, not the user.

6.6 [Best Practices] Generic error messages for pop operations

File(s): auto_renewal.cairo

Description: The function `batch_renew(...)` receives arrays for batch renewals, and their length needs to be the same. However, the length checking is not applied on the arrays `tax_price` and `metadata`. The function is reproduced below.

```
fn batch_renew(
    ref self: ContractState,
    domain: array::Span::<felt252>,
    renewer: array::Span::<starknet::ContractAddress>,
    limit_price: array::Span::<u256>,
    tax_price: array::Span::<u256>,
    metadata: array::Span::<felt252>,
) {
    ...
    assert(domain.len() == renewer.len(), 'Domain & renewer mismatch len');
    assert(domain.len() == limit_price.len(), 'Domain & price mismatch len');
    ...
    loop {
        if domain.len() == 0 {
            break;
        }
        let _domain = domain.pop_front().expect('pop_front error');
        let _renewer = renewer.pop_front().expect('pop_front error');
        let _limit_price = limit_price.pop_front().expect('pop_front error');
        ////////////////////////////////////////////////////
        // @audit Error messages for tax_price and metadata pop operations are too
        //        generic.
        ////////////////////////////////////////////////////
        let _tax_price = tax_price.pop_front().expect('pop_front error');
        let _metadata = metadata.pop_front().expect('pop_front error');
        self._renew(*_domain, *_renewer, *_limit_price, *_tax_price, *_metadata);
    }
}
```

If `tax_price` or `metadata` is lesser than the other arrays, the operation reverts with a generic error message.

Recommendation(s): Consider customizing the error messages to identify where the error has occurred easily.

Status: Fixed

Update from the client: We updated the code to check the missing arrays' lengths and replaced the .expect() by unwrap(). Fixed in: e1830508e0c310dc390684cc5cfc7517484bf2f4.

6.7 [Best Practices] Missing `tax_price` and `metadata` in the event `DomainRenewed`

File(s): `auto_renewal.cairo`

Description: The event `DomainRenewed` keeps track of the `domain`, `renewer`, `days`, `limit_price`, and `timestamp`, as shown below.

```
#[derive(Drop, starknet::Event)]
struct DomainRenewed {
    #[key]
    domain: felt252,
    renewer: ContractAddress,
    days: felt252,
    limit_price: u256,
    timestamp: u64,
}
```

This event is emitted in the function `fn _renew(...)`. This function receives the `root_domain`, `renewer`, `limit_price`, `tax_price`, and `metadata` as input parameters. The function header is reproduced below.

```
fn _renew(
    ref self: ContractState,
    root_domain: felt252,
    renewer: ContractAddress,
    limit_price: u256,
    tax_price: u256,
    metadata: felt252,
)
```

The code triggering the event is reproduced below.

```
self.emit(
    Event::DomainRenewed(
        DomainRenewed {
            domain: root_domain,
            renewer,
            days: 365,
            limit_price,
            timestamp: block_timestamp
        }
    )
);
```

We notice that the field `days` receives the constant `365`. Maybe this field can be omitted. On the other hand, the fields `tax_price` and `metadata` are not part of the event.

Recommendation(s): We suggest that the development team discuss the possibility of removing the field `days` and adding the `tax_price` and `metadata` fields.

Status: Fixed

Update from the client: We added the suggested relevant fields. Fixed in 5284970d3df79836c21debfe1cbd11ad3694bf52.

6.8 [Best Practices] Missing events emission

File(s): auto_renewal.cairo

Description: Some important state transitions are not generating events. Events provide a way to log and broadcast important state changes and transactions within a smart contract. These events are recorded on the blockchain and can be publicly accessed and examined by anyone. Events also enable off-chain systems and applications to monitor and analyze the behavior of smart contracts. Below, we list the functions that could benefit from emitting events.

```
- fn constructor(...) {}
- fn update_admin(...) {}
- fn update_tax_contract(...) {}
- fn update_whitelisted_renewer(...) {}
- fn toggle_off(...) {}
- fn claim(...) {}
```

Recommendation(s): We recommend that the development team carefully review the above functions and deliberate on which functions should emit events.

Status: Fixed

Update from the client: We added an event to all the listed functions so users can easily track what happens to the contract. Fixed in: e1830508e0c310dc390684cc5cfc7517484bf2f4.

6.9 [Best Practices] Storage variable `_is_renewing` can be more optimized.

File(s): auto_renewal.cairo

Description: Using `LegacyMap` in Cairo done hashes in the background. Storage variable `_is_renewing` is `LegacyMap` with keys of 3 elements tuple. However, `u256` type in Cairo is represented by two `u128`, so this mapping will have four elements tuple. To calculate the storage key of the `_is_renewing` variable, four Pedersen hashes and one keccak. Hash applications are one of the most expensive operations in Starknet.

```
#[storage]
struct Storage {
    ...
    _is_renewing: LegacyMap::<(ContractAddress, felt252, u256), bool>,
    ////////////////////////////////////////////////////////////////////
    // @audit-issue: Calculation of key consumes following applications.
    // p(p(p(p(keccak(_is_renewing),u256.high), u256.low,)
    // felt252,) ContractAddress,) p: Pedersen Hash
    ////////////////////////////////////////////////////////////////////
    ...
}
```

It is better to optimize the storage variable, while this variable is just used to register users' correct variables.

Recommendation(s): Consider using the following approach.

```
#[storage]
struct Storage {
    ...
    _is_renewing: LegacyMap::<(ContractAddress, felt252), u256>,
    ////////////////////////////////////////////////////////////////
    // @audit-issue: Similar usage but instead of using u256 as key,
    // it is used as a value.
    ////////////////////////////////////////////////////////////////
}
```

Status: Fixed

Update from the client: We adopted the suggested approach. Fixed in e1830508e0c310dc390684cc5cfc7517484bf2f4.

6.10 [Best Practices] Storage variables can be constant

File(s): auto_renewal.cairo

Description: Storage variables with names: `naming_contract` and `erc20_contract` can be constants instead of stored in contract storage. These variables are assigned in the constructor and never changed again. Reading from storage causes unnecessary syscalls.

```
#[storage]
struct Storage {
    naming_contract: ContractAddress,
    erc20_contract: ContractAddress,
    ...
}
```

Recommendation(s): Consider assigning these variables to contract-level constants.

Status: Acknowledged

Update from the client: We understand the suggestion but prefer to keep on storage variables because it is not that expensive and allows us to test the contract more easily and use the same class hash on both testnet and mainnet.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

The `Starknet.ID` team has provided two sources of documentation. A README explaining all the implemented features and restrictions in the Auto renew contract. Business logic documentation can be found in https://docs.starknet.id/.

8 Test Suite Evaluation

8.1 Contracts Compilation

```
$ scarb --release build
    Updating git repository https://github.com/starknet-id/naming
    Updating git repository https://github.com/openzeppelin/cairo-contracts
    Updating git repository https://github.com/starknet-id/identity
   Compiling auto_renew_contract v0.1.0 (/home/NM-135-StarknetID/auto_renew_contract/Scarb.toml)
    Finished release target(s) in 5 seconds
```

8.2 Tests Output

```
$ scarb test
     Running cairo-test auto_renew_contract
testing auto_renew_contract ...
running 15 tests
test auto_renew_contract::tests::test_renewals::test_toggle_off_contract_fail ... ok (gas usage est.: 1351970)
test auto_renew_contract::tests::test_renewals::test_claim_fail ... ok (gas usage est.: 1404310)
test auto_renew_contract::tests::test_renewals::test_update_whitelisted_renewer_fail ... ok (gas usage est.: 1357380)
test auto_renew_contract::tests::test_renewals::test_update_tax_addr_fail ... ok (gas usage est.: 1357380)
test auto_renew_contract::tests::test_renewals::test_claim ... ok (gas usage est.: 2374660)
test auto_renew_contract::tests::test_renewals::test_renew_fail_not_toggled ... ok (gas usage est.: 4330320)
test auto_renew_contract::tests::test_renewals::test_renew_disabled_contract_fails ... ok (gas usage est.: 4861820)
test auto_renew_contract::tests::test_renewals::test_toggle_renewal ... ok (gas usage est.: 4509170)
test auto_renew_contract::tests::test_renewals::test_renew_fail_expiry ... ok (gas usage est.: 4742220)
test auto_renew_contract::tests::test_renewals::test_renew_expired_domain ... ok (gas usage est.: 8010860)
test auto_renew_contract::tests::test_renewals::test_renew_domain ... ok (gas usage est.: 8007310)
test auto_renew_contract::tests::test_renewals::test_renew_with_metadata ... ok (gas usage est.: 7526450)
test auto_renew_contract::tests::test_renewals::test_renew_fail_wrong_limit_price ... ok (gas usage est.: 7309300)
test auto_renew_contract::tests::test_renewals::test_renew_with_updated_whitelisted_renewer ... ok (gas usage est.:
↪  8147280)
test auto_renew_contract::tests::test_renewals::test_renew_domains ... ok (gas usage est.: 14278620)
test result: ok. 15 passed; 0 failed; 0 ignored; 0 filtered out;
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer