Security Review Report
NM-0159 DOJO

NETHERMIND
SECURITY

(Feb 7, 2024)

Contents

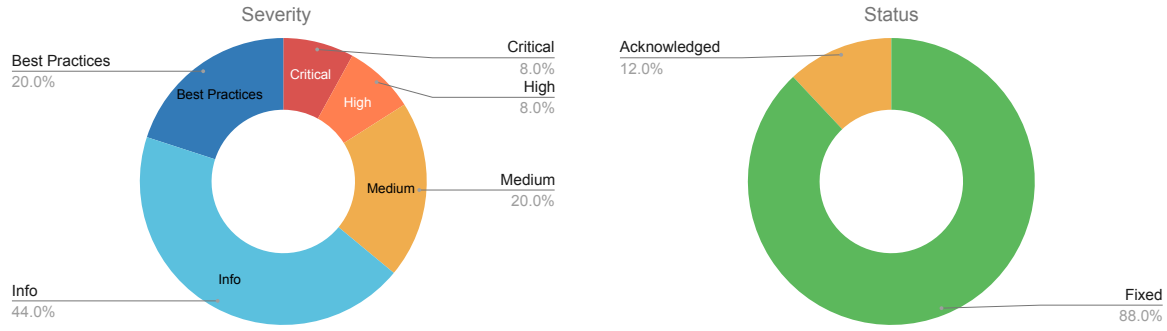
1 Executive Summary

This document outlines the security review conducted by Nethermind for the Dojo Game Engine. `Dojo` is a provable game engine and toolchain for building on-chain games and autonomous worlds. It is built around the concept of Entity Component System architecture. This model revolves around systems acting on entities, which are collections of pure data components. Systems efficiently determine which entities to process based on persistent queries over these components. The world contract controls all these components (systems and models). This contract enables the registration, creation, and deletion of entities. All data of these entities is stored in an on-chain database that supports indexing. The world contract also includes access control logic to authorize which addresses and systems can create, delete, or update entities.

The audited code comprises 945 lines of code in Cairo. The `Dojo` team has provided comprehensive documentation that explains Dojo's core architecture, and the detailed properties of each component, as well as how they interact with each other.

The audit was performed using: (a) manual analysis of the codebase and (b) simulation of the smart contracts. **Along this document, we report** 25 points of attention, where two are classified as `Critical`, two are classified as `High`, five are classified as `Medium`, and sixteen are classified as `Informational` or `Best Practice`. 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.



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (2), **High** (2), **Medium** (5), **Low** (0), **Undetermined** (0), **Informational** (11), **Best Practices** (5). **(b) Distribution of status: Fixed** (22), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan 25, 2024
Final Report	Feb 7, 2024
Methods	Manual Review
Repository	dojo-core
Commit Hash	36e5853877d011a5bb4b3bd77b9de676fb454b0c
Final Commit Hash	1a1f348536267773b37011ddbbb1aa820299a400
Documentation	Dojo Book
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/world.cairo	368	222	60.3%	82	672
2	src/components.cairo	1	0	0.0%	0	1
3	src/database.cairo	78	15	19.2%	10	103
4	src/base.cairo	24	8	33.3%	8	40
5	src/packing.cairo	128	12	9.4%	17	157
6	src/executor.cairo	26	15	57.7%	6	47
7	src/lib.cairo	14	7	50.0%	3	24
8	src/model.cairo	12	1	8.3%	1	14
9	src/database/utils.cairo	94	48	51.1%	24	166
10	src/database/storage.cairo	59	0	0.0%	12	71
11	src/database/index.cairo	101	24	23.8%	29	154
12	src/components/upgradeable.cairo	40	7	17.5%	9	56
	Total	**945**	**359**	**38.0%**	**201**	**1505**

3 Summary of Issues

	Finding	Severity	Update
1	Anyone can take ownership of non-exist models	Critical	Fixed
2	Setting metadata for resource can overwrite another resources metadata	Critical	Fixed
3	Executor contract can be upgraded to any class hash	High	Fixed
4	Function `delete(...)` does not update the new position for the last element	High	Fixed
5	Incorrect check in the `get_many()` function results in missing return data	Medium	Fixed
6	Incorrect check in the function `metadata_uri()` cause infinite loop	Medium	Fixed
7	Infinite loop in `query(...)` function when the condition is specified	Medium	Fixed
8	Packing lying on untrusted input	Medium	Acknowledged
9	The writer is able to upgrade a contract	Medium	Fixed
10	Function `get_by_key(...)` cannot be used because specific keys are not set anywhere	Info	Fixed
11	Missing check for duplicated elements in `ids` input params in the function `find_matching(...)`	Info	Fixed
12	No check if layout element matches max felt size	Info	Fixed
13	Open TODOs	Info	Fixed
14	Potential for front-running and DOS (Denial of Service) in `register_model(...)`	Info	Acknowledged
15	Potential overflow `offset` in the function `get_many(...)`	Info	Fixed
16	Power of a value missed edge case	Info	Fixed
17	The function `find_matching(...)` will fail for empty arrays	Info	Fixed
18	The index is not set in `set_entity(...)`	Info	Fixed
19	Upgrade checks not enough	Info	Fixed
20	`WhereCondition` is never being used in the function `query(...)`	Info	Fixed
21	Inconsistency of setting boolean values	Best Practices	Fixed
22	Incorrect documentation/comments	Best Practices	Fixed
23	Missing checks for a new executor contract	Best Practices	Fixed
24	Unnecessary check of address zero	Best Practices	Acknowledged
25	Unused variables	Best Practices	Fixed

4 System Overview

`Dojo` is a provable game engine for onchain games on Starknet and autonomous worlds. The game's owner deploys a world contract, which behaves as a database. It stores world models and systems. The world contract is a core contract that allows users to register their models, deploy system contracts, or set metadata for models.

The resource owners deploy system contracts that hold the world's logic, or it is possible to say game logic. The best way to understand this is an example: Imagine a system contract as a movement functionality for the world. It will hold functions that will update the player's position in the world.

Another component of this protocol is the model. A model is a structure that is represented as a table. It holds keys and values, which are stored in the world contract. First, an owner of a model needs to declare a class of a specific model to be able to register it within the World contract.

The protocol contains two main contracts: **World** and **Executor**. Code is organized into separate files, which we call modules or libraries. The main libraries of **World** contract are:

a) **Database**

b) **Index**

c) **Storage**

d) **Packing**

Fig. 2 presents the interaction diagram of the contracts.

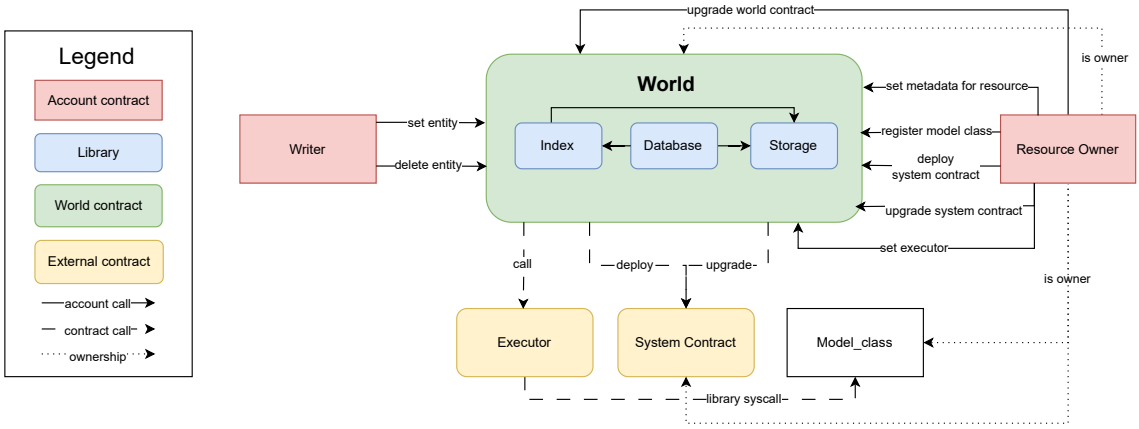


Fig. 2: Interaction Diagram of Contracts

The **Database** module contains the logic of a relational database. Users can create tables with keys and values, with each table representing an entity in the world contract.

The **Index** module handles creating and deleting indexes for the pair table-key.

The **Storage** module works as the storage of world contract. It offers low-level functionality for storing and fetching data from the internal storage of the contract.

The **Packing** module enables packing and unpacking functionality for custom data structures, ensuring efficient data handling within the system.

The modules mentioned above are part of the **World** contract, which implements an additional access control layer and external functionality for associating classes of models with the **World** contract and deploying system contracts. Furthermore, it allows functionality to write into a database and obtain needed data.

Another contract of the protocol is the **Executor**. **Executor** provides library system calls to deployed classes. In the current implementation, the **Executor** is called from **World** contract to obtain the name of the module class.

Two types of actors will interact with the **World** contract are:

a) **Resource owners:** Users who register models or deploy system contracts associated with the World contract become resource owners. They have the authority to manage these resources by inserting data and metadata related to them. Additionally, resource owners can assign ownership or writing privileges to other account contracts for their resources. It's important to note that the admin of the **World** contract is also part of this group.

b) **Writers:** Members of this group, likely system contracts, are responsible for setting data to the relational database or deleting data.

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 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.

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

6 Issues

6.1 [Critical] Anyone can take ownership of non-exist models

File(s): world.cairo

Description: The registering model process has no access control checks if the model name does not exist. However, owners of models are stored in the same mapping. A malicious actor can register their own model to take ownership of existing contracts or anything like the following assumption.

If there is a contract deployed by calling the `deploy_contract(...)` method, the owner will be the caller. However, a malicious actor can register the model with the name as the contract address of this newly deployed contract to take ownership.

Deploying a contract does not write models storage its contract address. So following if statement will grant ownership to the malicious actor;

```
fn register_model(ref self: ContractState, class_hash: ClassHash) {
    // ...
    let current_class_hash = self.models.read(name);
    if current_class_hash.is_non_zero() {
        assert(self.is_owner(caller, name), 'only owner can update');
        prev_class_hash = current_class_hash;
    } else {
        self.owners.write((name, caller), true);
    };

    // @audit-issue: Everyone can make themselves of owner of anything that doesn't exist in models.

    self.models.write(name, class_hash);
    EventEmitter::emit(ref self, ModelRegistered { name, class_hash, prev_class_hash });
}
```

As we see there is no model storage writing on the `deploy_contract(...)` method.

```
fn deploy_contract(
    ref self: ContractState, salt: felt252, class_hash: ClassHash
) -> ContractAddress {
    let (contract_address, _) = deploy_syscall(
        self.contract_base.read(), salt, array![].span(), false
    )
        .unwrap_syscall();
    let upgradeable_dispatcher = IUpgradeableDispatcher { contract_address };
    upgradeable_dispatcher.upgrade(class_hash);

    self.owners.write((contract_address.into(), get_caller_address()), true);

    EventEmitter::emit(
        ref self, ContractDeployed { salt, class_hash, address: contract_address }
    );

    contract_address
}
```

So initially, the model mapping will be zero for newly deployed contracts, and malicious actors can register the model to get ownership and then get access to access-controlled methods.

Recommendation(s): Consider changing the owner storage structure or registering the deployed contract address as a model.

Status: Fixed

Update from the client: In the commit bfd952b5112f3790621b1a5f68d47b25d9daa7d2 a new storage variable was added to avoid such conflict.

Update from Nethermind: It is still feasible to get an ownership of WORLD.

Update from the client: Fixed in 202d077 and missing test added in 0e1ea82. Updated in 1a1f3485 to ensure the resource metadata are also protected and only controlled by the world's creator.

6.2 [Critical] Setting metadata for resource can overwrite another resources metadata

File(s): world.cairo

Description: Resource metadata stored in one storage mapping like

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

Where the felt value is the name of the resource, and the first mapped value is the length of the metadata array. Length is used to read metadata like the following.

```
fn metadata_uri(self: @ContractState, resource: felt252) -> Span<felt252> {
    let mut uri = array![];

    // We add one here since we start i at 1;
    let len = self.metadata_uri.read(resource) + 1;

    let mut i = resource + 1;
    loop {
        if len == i {
            break;
        }

        uri.append(self.metadata_uri.read(i));
        i += 1;
    };

    uri.span()
}
```

However, writing can be done by only the owner of the resource;

```
fn set_metadata_uri(ref self: ContractState, resource: felt252, mut uri: Span<felt252>) {
    assert(self.is_owner(get_caller_address(), resource), 'not owner');
    // ...
}
```

But everyone can register their own model and be their own model's owner by calling `register_model(...)`. If a malicious actor registers a model with the name felt of `target_resource - 1`, then the actor can use this ownership to overwrite target resources metadata.

Recommendation(s): Consider changing metadata storing structure. Use Resource -> Span mapping.

Status: Fixed

Update from the client: That's a good catch. The situation here originates from the metadata uri being handles as a resource, but on more than one felt. Resources are to be considered as a one felt long identifier. To address this, we've adjusted the way metadata are handled by the world, and now it follows the regular ACL flow by using the same logic as models do. commit: f129edfc.

6.3 [High] Executor contract can be upgraded to any class hash

File(s): world.cairo

Description: Function `register_model(...)` can be used to upgrade existing executor contract implementation. In this function, there is an external call to the executor, where the executor library calls to target class hash. However, a malicious actor can pass any class hash. That class hash can contain `replace_class_syscall` to upgrade the executor contract to any other class hash.

```
fn register_model(ref self: ContractState, class_hash: ClassHash) {
    // ...
    let name = *class_call(@self, class_hash, NAME_ENTRYPOINT, calldata.span())[0];
    // @audit-issue: Any class hash can be used to library call at executor.
    // ...
}
```

```
impl Executor of IExecutor<ContractState> {
    fn call(
        self: @ContractState,
        class_hash: ClassHash,
        entrypoint: felt252,
        calldata: Span<felt252>
    ) -> Span<felt252> {
        starknet::syscalls::library_call_syscall(class_hash, entrypoint, calldata)
            .unwrap_syscall() // @audit-issue: There are no checks, call forwarded directly.
    }
}
```

Recommendation(s): Consider deploying the contract, then call to get the name.

Status: Fixed

Update from the client: To limit the API changes on current Dojo implementation, we propose in commits 479197e4f and 04f7ef756 to remove the executor. Indeed, the `library_call` syscall can't prevent the execution of syscalls, and in the context of Dojo, we do need model registering to be permissionless. The proposed solution introduces communication with models via regular `contract_call` syscall instead of library call, which prevents the world (or auxiliary contract like executor) from being modified.

6.4 [High] Function `delete(...)` does not update the new position for the last element

File(s): `index.cairo`

Description: In the `index.cairo` contract, there are 3 mappings used to store elements.

1. The first mapping stores the position of an element `(index, id) => position + 1`;
2. The second mapping stores the number of elements in this index `(index) => length`;
3. The third mapping stores the id of the element in each position `(index, position) => id`;

When an element is removed from an index, the function `delete(...)` does the following:

- Removes the position of the deleted element from the first mapping by setting its value to 0;
- Decreases the number of elements in the index in the second mapping;
- Updating the third mapping to point to the last element id;

However, since the last element is moved to a new position, the position of the last element is also changed. This means that we also need to update the first mapping for the last element. Currently, the function `delete(...)` does not handle this, which breaks the structure of the contract as the first mapping points to the wrong position.

```
fn delete(address_domain: u32, index: felt252, id: felt252) {
    if !exists(address_domain, index, id) {
        return ();
    }

    let index_len_key = build_index_len_key(index);
    let replace_item_idx = storage::get(address_domain, index_len_key) - 1;

    let index_item_key = build_index_item_key(index, id);
    let delete_item_idx = storage::get(address_domain, index_item_key) - 1;

    storage::set(address_domain, index_item_key, 0);
    storage::set(address_domain, index_len_key, replace_item_idx);

    // Replace the deleted element with the last element.
    // NOTE: We leave the last element set as to not produce an unncessary state diff.

    // @audit-issue Not update new position of replace item
    let replace_item_value = storage::get(address_domain, build_index_key(index, replace_item_idx));
    storage::set(address_domain, build_index_key(index, delete_item_idx), replace_item_value);
}
```

Recommendation(s): Consider updating the new position for the last element.

```
storage::set(address_domain, build_index_item_key(index, replace_item_value), delete_item_idx + 1);
```

Status: Fixed

Update from the client: Related to index removed code in e95153c46cfcb2e83a21f31ded201b41a6b0c2d6.

6.5 [Medium] Incorrect check in the `get_many()` function results in missing return data

File(s): `storage.cairo`

Description: The `get_many(...)` function uses `offset` to specify the starting position for retrieval from storage. It should retrieve storage slots from `offset` to `offset + length - 1`. However, the current code retrieves from `offset` to `length - 1` only. If `offset > length`, it causes an infinite loop. The issue is evident in the code snippet below, where the loop breaks when `offset == length`.

```
let mut offset = offset;
loop {
    // @audit-issue Incorrect check, should be (length + original_offset == offset.into())
    if length == offset.into() {
        break ();
    }
```

Recommendation(s): Consider changing the break condition to `length + original_offset == offset.into()` where the `original_offset` is the initial value of offset before the loop starts.

Status: Fixed

Update from the client: Yes that's correct. Currently only the offset 0 is enforced by the `dojo-lang` macro. This is the reason why it wasn't detected before.

We've adjusted the storage layer to always start at the offset 0, and offset is no longer exposed.

commit: f03cce87ade45848b5c5ab9b4a6cf72666343211.

And in a second commit, adjustement to be backward compatible without storing the length in the storage: 4b5c0f8023357a8d3c9c782ae89621b2978947ad.

We may update this to add the offset again to allow some further optimization on data retrieval. If added, it bounds checking will come along the offset.

6.6 [Medium] Incorrect check in the function `metadata_uri()` cause infinite loop

File(s): `world.cairo`

Description: In `world.cairo`, the metadata URI of a resource is stored as follows. First, at the `resource` position, it will store the length of the metadata URI. The following positions store the actual metadata URI.

```
fn metadata_uri(self: @ContractState, resource: felt252) -> Span<felt252> {
    let mut uri = array![];

    // We add one here since we start i at 1;
    let len = self.metadata_uri.read(resource) + 1;

    let mut i = resource + 1;
    loop {
        // @audit Incorrect check, it is assuming resource = 0
        // @audit The check should be `if resource + len == i`
        if len == i {
            break;
        }

        uri.append(self.metadata_uri.read(i));
        i += 1;
    };

    uri.span()
}
```

Consider the following scenario:

1. Alice stores the metadata URI for `resource = 256` with a length of `5`. In `set_metadata_uri(...)`, it will set `metadata_uri[256] = 5`, and the next 5 positions from `257` to `261` will store the metadata URI;
2. Now, when the function `metadata_uri(...)` is called to get the metadata URI for `resource = 256`, it will calculate:;

```
len = metadata_uri[256] + 1 = 6
i = resource + 1 = 257
```

As you can see, since the starting value of `i = 257` is already larger than `len = 6`, the break condition `len == i` will never be reached as the value of `i` will increase in each iteration.

Recommendation(s): Change the check to `resource + metadata_uri[resource] == i`.

Status: Fixed

Update from the client: Concern addressed in the previous one as this code is not longer present.

6.7 [Medium] Infinite loop in `query(...)` function when the condition is specified

File(s): `index.cairo`

Description: The `query(...)` function serves to return all elements from a table. This function allows the usage of the `where` parameter, which works similarly to the SQL `where` keyword, as it will return elements based on the condition. If this parameter is not specified, the function will return all values from the table.

However, only the elements that match the condition should be returned if the `where` parameter is used. This part of the code contains a loop that iterates through elements connected with a specific index. The index is created based on the `key` of the condition. The loop is where the issue arises, as the variable `idx` is not updated at the end of the iteration, which will cause an infinite loop, and the function will fail for an `out of gas` error.

```
Option::Some(clause) => {
    let mut serialized = ArrayTrait::new();
    table.serialize(ref serialized);
    clause.key.serialize(ref serialized);
    let index = poseidon_hash_span(serialized.span());

    let index_len_key = build_index_len_key(index);
    let index_len = storage::get(address_domain, index_len_key);
    let mut idx = 0;

    loop {
        if idx == index_len {
            break ();
        }
        let id = storage::get(address_domain, build_index_key(index, idx));
        res.append(id);
        // @audit-issue idx is not updated
    }
},
```

Recommendation(s): It is necessary to update the `idx` variable inside a loop at the end of the iteration.

Status: Fixed

Update from the client: Linked to `WhereClause`.

Code related to index was removed for now as it will be reworked in the future:

commit: 98ed1689eabad628331789c82ac01fdb3ff7e173

6.8 [Medium] Packing lying on untrusted input

File(s): packing.cairo

Description: Packing method trusting layout parameter for bitsizes of variables inside unpacked span. However, the layout parameter is not coming from a trusted source. It can be inputted as any inside `set_entity(...)` method.

```
fn pack(ref packed: Array<felt252>, ref unpacked: Span<felt252>, ref layout: Span<u8>) {
    assert(unpacked.len() == layout.len(), 'mismatched input lens');
    let mut packing: felt252 = 0x0;
    let mut offset: u8 = 0x0;
    loop {
        match unpacked.pop_front() {
            Option::Some(item) => {
                pack_inner(item, *layout.pop_front().unwrap(), ref packing, ref offset, ref packed);
            },
            Option::None(_) => {
                break;
            }
        };
    };
    packed.append(packing);
}
```

Recommendation(s): Consider store using native types. If the writer is trusted executable, consider packing off-chain.

Status: Acknowledged

Update from the client: The explicit layout is currently something very controlled by the `dojo-lang` compiler, where expansion of macro auto-generate the layout to be sent. As mentioned, sending a transaction explicitely can result in corrupted data, but since ACLs are covering who can modify the data, the input is always considered as trusted.

Finally, this is also a flexible way to allow the user to send arbitrarily packed data.

6.9 [Medium] The writer is able to upgrade a contract

File(s): world.cairo

Description: The protocol allows the contract deployment to be associated with the world contract by the function `deploy_contract(...)`. When the contract is deployed, it updates a mapping `owners` so that the caller of this function becomes an owner of the deployed contract.

The contract associated with the world can also be upgraded by invoking `upgrade_contract(...)` function. The following function checks the permissions of a caller by the following line:

```
assert_can_write(@self, address.into(), get_caller_address());
...
fn assert_can_write(self: @ContractState, resource: felt252, caller: ContractAddress) {
    assert(
        IWorld::is_writer(self, resource, caller)
            || IWorld::is_owner(self, get_tx_info().unbox().account_contract_address, resource)
            || IWorld::is_owner(self, get_tx_info().unbox().account_contract_address, WORLD),
        'not writer'
    );
}
```

The function `assert_can_write(...)` checks if the caller is the owner of `WORLD` or the owner of the resource(in this case, an owner of the contract) or has writing permissions.

However, that is where the issue arises as just an owner of the contract or of the `WORLD` should be able to upgrade a contract. The writer should not have a privilege for this action.

Recommendation(s): Consider checking access control for the `upgrade_contract(...)` function.

Status: Fixed

Update from the client: Fixed in 1d65123556f8b070ebd730647d4407befb1c92d8 where only ownership is check for upgrade.

6.10 [Info] Function `get_by_key(...)` cannot be used because specific keys are not set anywhere

File(s): `index.cairo`

Description: In the `index.cairo` contract, the `get_by_key(...)` function is designed to return all entries containing a specific key. The function utilizes `build_index_specific_key_len(...)` and `build_index_specific_key(...)` to create the specific keys and build the storage address. However, these keys are not set anywhere in the contract. As a result, the function will fail to return any results, rendering it unusable.

```
fn get_by_key(address_domain: u32, index: felt252, key: felt252) -> Array<felt252> {
    let mut res = ArrayTrait::new();
    let specific_len_key = build_index_specific_key_len(index, key); // @audit-issue Not set anywhere
    let index_len = storage::get(address_domain, specific_len_key);

    let mut idx = 0;

    loop {
        if idx == index_len {
            break ();
        }

        let specific_key = build_index_specific_key(index, key, idx); // @audit-issue Not set anywhere
        let id = storage::get(address_domain, specific_key);
        res.append(id);

        idx += 1;
    };

    res
}
```

Recommendation(s): Implement features to set specific keys for index.

Status: Fixed

Update from the client: Index related code was removed in e95153c46cfcb2e83a21f31ded201b41a6b0c2d6.

6.11 [Info] Missing check for duplicated elements in `ids` input params in the function `find_matching(...)`

File(s): `utils.cairo`

Description: The function `find_matching(...)` takes multiple entity lists as input parameters. It aims to find the set of entities present in all the provided lists. It does this by counting how many times each entity id occurs. If an id appears a number of times equal to the number of lists, it means the entity with this id exists in all lists. However, the function does not account for cases where a list contains duplicated entities. If any `entity_ids` list contains duplicate values, it would cause the count in `ids_match` to be incorrect, potentially returning an id that does not exist in all lists.

Recommendation(s): Consider checking for duplicate elements in each `entity_ids` list.

Status: Fixed

Update from the client: Index related code was removed in e95153c46cfcb2e83a21f31ded201b41a6b0c2d6.

6.12 [Info] No check if layout element matches max felt size

File(s): packing.cairo

Description: The `pack_inner(...)` function is packing a felt element. It takes several parameters. One of them is a `size` specifying the element's size. This element is obtained from the `layout` array parameter of the `pack(...)` function.

The element must fit to `felt252`; however, due to the following line and comments, the maximum size allowed is 251.

```
// Cannot use all 252 bits because some bit arrangements (eg. 11111...11111) are not valid felt252 values.
// Thus only 251 bits are used.                                ^-252 times-^
// One could optimize by some conditional alligment mechanism, but it would be an at most 1/252 space-wise improvement.
let remaining_bits: u8 = (251 - packing_offset).into();
```

If not, the line above will overflow, and the packing will revert. But If the last element of the layout (size) is larger than `251` and lower than `256` (to fit u8 space), the `pack_inner(...)` will not revert. There is no direct impact from this missing edge case. Still, to keep the consistency of the code, there should be an assertion that the size is not larger than `251`.

Recommendation(s): Consider additional checks that the size is not larger than the `251` value.

Status: Fixed

Update from the client: Was addressed in 29621f526957d5a330d2dc5c52af24fa68621c42 to ensure good behavior if layout is not correct.

6.13 [Info] Open TODOs

File(s): packing.cairo

Description: There is an open TODO in the function `unpack(...)`. According to the comment, the function should raise an error, but currently, it only exits the loop.

```
Option::None(_) => {
    // TODO: Raise error
    break;
}
```

Recommendation(s): Consider reviewing the intended behavior of this branch again.

Status: Fixed

Update from the client: Changed for a comment in b1e1aa82fa542dc516fa94d0915e0a4ab825a25f.

6.14 [Info] Potential for front-running and DOS (Denial of Service) in `register_model(...)`

File(s): `world.cairo`

Description: The `world` contract includes multiple models, each tracked by its name. This name is retrieved from the model class hash during the registration of the model.

```
fn register_model(ref self: ContractState, class_hash: ClassHash) {
    let caller = get_caller_address();
    let calldata = ArrayTrait::new();

    // @audit-issue Can front-run to register model with the same name
    let name = *class_call(@self, class_hash, NAME_ENTRYPOINT, calldata.span())[0];
    let mut prev_class_hash = starknet::class_hash::ClassHashZeroable::zero();

    // If model is already registered, validate permission to update.
    let current_class_hash = self.models.read(name);
    if current_class_hash.is_non_zero() {
        assert(self.is_owner(caller, name), 'only owner can update');
        prev_class_hash = current_class_hash;
    } else {
        self.owners.write((name, caller), true);
    };

    self.models.write(name, class_hash);
    EventEmitter::emit(ref self, ModelRegistered { name, class_hash, prev_class_hash });
}
```

However, since `register_model(...)` is a public function, an attacker could front-run and register a null or malicious model using the same `name` as the legitimate model registration transaction. Consequently, the legitimate model cannot be registered because its name already exists. The attacker could repeatedly do this to effectively block all model registrations.

Recommendation(s): Consider restricting the addresses that can register a new model or change the method of model tracking. For instance, use a combination of (name, caller_address) to mitigate the issue.

Status: Acknowledged

Update from the client: This is inherent to how the permissionless registration of the models works in the world. Due to that, front-running is indeed possible, but per se not an issue as then the permissions are still controlled by the owner.

6.15 [Info] Potential overflow `offset` in the function `get_many(...)`

File(s): storage.cairo

Description: The `get_many(...)` function uses `offset` to specify the start position to get from storage. The function retrieves storage slots from `offset` to `offset + length - 1`. However, `offset` uses `u8` while `length` uses `usize`. Therefore, if `length > max(u8)`, `offset` may overflow.

```
fn get_many(address_domain: u32, keys: Span<felt252>, offset: u8, length: usize, mut layout: Span<u8>) -> Span<felt252>
↪ {
  let base = starknet::storage_base_address_from_felt252(poseidon_hash_span(keys));
  let mut packed = ArrayTrait::new();

  let mut offset = offset;
  loop {
      // @audit-issue offset's datatype is smaller than `usize` and could overflow
      if length == offset.into() {
          break ();
      }

      packed
          .append(
              starknet::storage_read_syscall(
                  address_domain, starknet::storage_address_from_base_and_offset(base, offset)
              )
                  .unwrap_syscall()
          );

      offset += 1;
  };
  ...
}
```

Recommendation(s): Consider using the same data type for `offset` and `length`.

Status: Fixed

Update from the client: Solved in f03cce87ade45848b5c5ab9b4a6cf72666343211 where the offset is no longer exposed.

If we re-introduce the offset, bounds checks will be implemented as mentioned here.

6.16 [Info] Power of a value missed edge case

File(s): packing.cairo

Description: Method `fpow(...)` returns 1 if the second variable is zero. But there is no check for whether the first value is zero or not. In math, 0 power 0 should be undefined. In this case the method should return. But it returns 1 early because the second variable is zero.

```
fn fpow(x: u256, n: u8) -> u256 {
    let y = x;
    // @audit-issue: Should revert if x and n are zero.
    if n == 0 {
        return 1;
    }
    // ...
}
```

Recommendation(s): Consider checking if both parameters are zero or not.

Status: Fixed

Update from the client: Updated in 0c0a4ec462bff2ed5f8de3a8c712fc785621603d.

6.17 [Info] The function `find_matching(...)` will fail for empty arrays

File(s): `utils.cairo`

Description: The function `find_matching(...)` in `utils.cairo` serves to find entities that match the same IDs across all supplied entities.

As this function is not used anywhere in `world.cairo`, we guess that this function will be used for some off-chain interaction. Therefore, this function should handle all edge cases and probably not fail in different cases than assert statements checks.

However, the following function will fail if the supplied arrays are empty due to the following line.

```
// we want to keep the ordering from the first entity IDs
let mut ids1: Span<felt252> = *(ids.pop_front().unwrap()); //@audit-issue here it will fail for zero arrays
```

Recommendation(s): Consider handling a case when the supplied params are empty arrays.

Status: Fixed

Update from the client: Index related code was removed in e95153c46cfcb2e83a21f31ded201b41a6b0c2d6.

6.18 [Info] The index is not set in `set_entity(...)`

File(s): `world.cairo`

Description: The `world.cairo` exposes two functions: `set_entity(...)` and `delete_entity(...)`, for setting the model to the entity and deleting the model from the entity, respectively.

When the entity is set, the contract interacts with the storage to store data related to the model based on the keys.

All the values linked with keys are set to 0 during the entity's deletion. Additionally, when the entity is deleted, its index is also removed from the storage. However, no index is created with the entity during the flow of `set_entity(...)` function.

Recommendation(s): Consider creating an index when setting an entity.

Status: Fixed

Update from the client: Index related code was removed in e95153c46cfcb2e83a21f31ded201b41a6b0c2d6.

6.19 [Info] Upgrade checks not enough

File(s): `components/upgradeable.cairo`

Description: The upgrade method has a couple of checks, but it is not checking that the new class hash has the method world(...). If upgraded to a hash that this function doesn't exist, then all further upgrade trials will revert, and it can't be reversed.

```
fn upgrade(ref self: ComponentState<TContractState>, new_class_hash: ClassHash) {
    // @audit-issue: The assertion below will revert if there is no world entrypoint in the new class.
    assert(
        self.get_contract().world().contract_address.is_non_zero(),
        Errors::INVALID_WORLD_ADDRESS
    );
    assert(
        get_caller_address() == self.get_contract().world().contract_address,
        Errors::INVALID_CALLER
    );
    // ...
}
```

Recommendation(s): Consider checking methods used in assertions that also exist in the new implementation.

Status: Fixed

Update from the client: Attempt to fix with a `library_call_syscall`, not sure about the point of doing that right now. I would say we can wait interfaces to be supported and standardized (SNIP). A first attempt to mitigate this risk was done in commit: 8ff011.

6.20 [Info] `WhereCondition` is never being used in the function `query(...)`

File(s): database.cairo, index.cairo

Description: The function `query(...)` accommodates both conditional and non-conditional queries by allowing an input `where: Option<WhereCondition>`. However, this function is currently only used in the function `scan_ids(...)`, which always inputs `Option::None` as the `where` parameter. Consequently, `WhereCondition` is never used, and the logic written for it is redundant.

Consider the test case below for the function `scan(...)` that tests the case where it uses `WhereCondition`. This test fails and throws an `Index out of bounds` error.

```
#[test]
#[available_gas(10000000)]
fn test_database_scan2() {
    let even = array![2, 4].span();
    let odd = array![1, 3].span();
    let layout = array![251, 251].span();

    set_with_index('table', 'even', 0, even, layout);
    set_with_index('table', 'odd', 0, odd, layout);

        let condition : WhereCondition = WhereCondition {
        key: 'odd', // Replace with an appropriate key
        value: '1', // Replace with an appropriate value
    };

    let (keys, values) = scan('table', Option::Some((condition)), 2, layout);
    let res  = *keys.at(0);
    res.print();

}
```

Recommendation(s): Review the logic around `WhereCondition` in the function `query(...)`. If it's no longer being used, consider removing it.

Status: Fixed

Update from the client: The `WhereClause` was deprecated in 98ed1689eabad628331789c82ac01fdb3ff7e173.

6.21 [Best Practices] Inconsistency of setting boolean values

File(s): world.cairo

Description: The approach for setting a boolean value is differentiated across functions `revoke_owner(...)` and `revoke_write(...)`. The first function applies the following line:

```
self.owners.write((resource, address), bool::False(()));
```

While the second one uses simple `false`:

```
self.writers.write((model, system), false);
```

In practice, there is no distinction. However, adhering to consistent coding practices is advisable.

Recommendation(s): Standardize the approach for setting boolean values.

Status: Fixed

Update from the client: Updated in fdebe59d0997811b763d3059467fcb1d20b1a987.

6.22 [Best Practices] Incorrect documentation/comments

File(s): world.cairo

Description: Documentation for functions and contracts plays a crucial role in providing clarity on expected behaviour and the parameters required for utilizing specific functionality. Accurate comments (inline documentation) are essential for facilitating a clear understanding for developers, auditors, and users.

The inline documentation is wrongly written for the following functions:

- `set_entity(...);`
- `delete_entity(...);`
- `entity(...);`
- `entities(...);`
- `entity_ids(...);`
- `upgrade_contract(...);`
- `deploy_contract(...);`
- `set_metadata_uri(...);`
- `metadata_uri(...);`

Recommendation(s): Consider changing the documentation to describe the correct implementation of the functions.

Status: Fixed

Update from the client: Documentation updated in ef9279995afd379cb13f7dabb2b8c6912b9b8690 with error refactoring.

6.23 [Best Practices] Missing checks for a new executor contract

File(s): world.cairo

Description: The function `set_executor(...)` can change the executor's address but this function can be called just by the owner of the `WORLD` contract. The address of the new executor can be set to any arbitrary value.

However, a potential issue may arise if the executor's address is unintentionally set to zero or if a contract lacks the implementation of the `call(...)` function.

Recommendation(s): Consider checking if the new executor's address is not zero and if the `call(...)` function is available.

Status: Fixed

Update from the client: First update by verifying that the new address is not zero in c804b77e9b92a058829d4171c3a5c766832d0a50. TBD for the verification of `call`, should be done using interfaces instead of verification of entry point existency. But the executor may be removed anyway.

6.24 [Best Practices] Unnecessary check of address zero

File(s): components/upgradeable.cairo

Description: The upgrade method is already checking that the caller is a world address. However, there is no way that caller's address can be zero. So first assertion

```
assert(
    self.get_contract().world().contract_address.is_non_zero(),
    Errors::INVALID_WORLD_ADDRESS
);
```

is unnecessary in this case.

Recommendation(s): Consider removing this assertion.

Status: Acknowledged

Update from the client: If the contract has the world dispatcher uninitialized in the storage, the contract address will be 0. Then, even if there is a low chance to have the `get_caller_address()` to be zero, this is technically possible, and the second assertion alone is not enough to prevent this situation.

6.25 [Best Practices] Unused variables

File(s): `world.cairo`, `executor.cairo`

Description: The codebase contains several variables that are declared but never used.

- The function `delete_entity(...)` defines a `model_class_hash` variable that is not used;
- The function `entities(...)` has an `index` input parameter that is never used;
- The constant `EXECUTE_ENTRYPOINT` in `executor.cairo` is not used anywhere;

Recommendation(s): Consider deleting unused variables.

Status: Fixed

Update from the client: This was solved by several commits:
`delete_entity` in 9a94a397867928347955b61bd3df2bd4b3355fc3.
`entities()` removed as never used for now in e95153c46cfcb2e83a21f31ded201b41a6b0c2d6.
`EXECUTE_ENTRYPOINT` removed in 73d78093f4f1e0df95b4874c8b00af80b5034601.

Update from Nethermind: The changes in the code introduced unused storage variable `metadata_uri` in the `world.cairo`.

Update from the client: Fixed in c04ffa5c42c31b5cf90f6fca5bd1ea7b675e938b and 572fdc8a0d46d6a286dae8014d5824e6a34acd2d.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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.

Remarks about Dojo documentation

The `Dojo` team has provided documentation about their core game engine in the codebase comments and through their official documentation - Dojo Book. Moreover, the Dojo team was available to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Compilation Output

```
# NOTE: No output when successful compilation
> sozo build
```

8.2 Tests Output

```
> cargo run -r --bin sozo -- --manifest-path crates/dojo-core/Scarb.toml test
    Finished release [optimized] target(s) in 0.50s
     Running `target/release/sozo --manifest-path crates/dojo-core/Scarb.toml test`
running 60 tests
test dojo::packing_test::test_pack_unpack_felt252_u128 ... ok (gas usage est.: 1222852)
test dojo::packing_test::test_pack_unpack_single ... ok (gas usage est.: 672016)
test dojo::database::storage_test::test_storage_empty ... ok (gas usage est.: 2134184)
test dojo::packing_test::test_pack_unpack_max_felt252 ... ok (gas usage est.: 161316)
test dojo::packing_test::test_bit_fpow ... ok (gas usage est.: 455930)
test dojo::database::storage_test::test_storage_set_many ... ok (gas usage est.: 1035330)
test dojo::database_test::test_database_different_tables ... ok (gas usage est.: 1325496)
test dojo::database::introspect_test::test_generic_introspect ... ok (gas usage est.: 0)
[DEBUG] 0x2
[DEBUG] 0x2
test dojo::packing_test::test_inner_pack_unpack_u256_single ... ok (gas usage est.: 1236012)
test dojo::packing_test::test_pack_unpack_felt252_single ... ok (gas usage est.: 304042)
test dojo::database_test::test_database_different_keys ... ok (gas usage est.: 1325496)
test dojo::database_test::test_database_basic ... ok (gas usage est.: 669618)
test dojo::packing_test::test_bit_shift ... ok (gas usage est.: 1482020)
test dojo::packing_test::test_pack_unpack_u256_single ... ok (gas usage est.: 1327668)
test dojo::database::storage_test::test_storage ... ok (gas usage est.: 713178)
test dojo::packing_test::test_calculate_packed_size ... ok (gas usage est.: 244900)
test dojo::packing_test::test_pack_unpack_types ... ok (gas usage est.: 5289134)
test dojo::packing_test::test_pack_max_bits_value ... ok (gas usage est.: 56778)
test dojo::packing_test::test_pack_multiple ... ok (gas usage est.: 8077598)
test dojo::world_test::test_emit ... ok (gas usage est.: 996920)
test dojo::base_test::test_deploy_from_world_invalid_model_world ... ok (gas usage est.: 1292510)
test dojo::world_test::test_model ... ok (gas usage est.: 1323510)
[DEBUG] 0x3
test dojo::packing_test::test_pack_unpack_multiple ... ok (gas usage est.: 91414292)
test dojo::world_test::test_set_metadata_world ... ok (gas usage est.: 2461130)
test dojo::base_test::test_deploy_from_world_invalid_model ... ok (gas usage est.: 1936070)
test dojo::base_test::test_upgrade_direct ... ok (gas usage est.: 1651950)
test dojo::world_test::test_writer ... ok (gas usage est.: 1688730)
test dojo::base_test::test_upgrade_from_world_not_world_provider ... ok (gas usage est.: 1929200)
test dojo::world_test::test_set_metadata_same_model_rules ... ok (gas usage est.: 1233410)
test dojo::database::storage_test::test_storage_set_many_several_segments ... ok (gas usage est.: 200608996)
[DEBUG] 0x60c5e22020636861722072365742063616c6c
[DEBUG] 0x4d57073746f726167652073657420636d6e79
test dojo::base_test::test_upgrade_from_world ... ok (gas usage est.: 2033600)
[DEBUG] 0x508a673746f726167652067657420636d6e79
test dojo::benchmarks::bench_storage_many ... ok (gas usage est.: 12742230)
test dojo::world_test::test_delete ... ok (gas usage est.: 5188326)
test dojo::world_test::test_system ... ok (gas usage est.: 3338338)
test dojo::world_test::test_set_entity_admin ... ok (gas usage est.: 3324338)
test dojo::world_test::test_system_writer_access ... ok (gas usage est.: 2636374)
[DEBUG] 0x2d50202020206e61746976652070726570 ('-P    native prep')
[DEBUG] 0xaf04fa20636861722076574206d6163726f
test dojo::world_test::test_set_entity_unauthorized ... ok (gas usage est.: 1968280)
[DEBUG] 0x2020202062656e636820656d707479 ('    bench empty')
[DEBUG] 0x436c2020206e6174697665207772697465 ('Cl   native write')
test dojo::world_test::bench_execute_complex ... ok (gas usage est.: 32544620)
test dojo::benchmarks::bench_reference_offset ... ok (gas usage est.: 7622856)
test dojo::world_test::test_set_writer_fails_for_non_owner ... ok (gas usage est.: 1169140)
```

```
test dojo::world_test::test_metadata_update_owner_only ... ok (gas usage est.: 1314110)
[DEBUG] 0x45d8202020206e61746976652072656164
test dojo::world_test::test_set_owner_fails_for_non_owner ... ok (gas usage est.: 1079800)
test dojo::world_test::test_owner ... ok (gas usage est.: 2459820)
[DEBUG] 0x3022020202020202020666f6f20696e6974
[DEBUG] 0x8ef82020202073746f7261676520736574
[DEBUG] 0x2d50206e617469766520707265726f66 ('-P native prep of')
[DEBUG] 0x8ac2020666f6f2073657269616c697a65
test dojo::benchmarks::bench_native_storage ... ok (gas usage est.: 22519028)
[DEBUG] 0x436c206e617469766520777269742064206f66 ('Cl native writ of')
[DEBUG] 0x8ffc2020202073746f7261676520676574
[DEBUG] 0x2f8020202020202666f6f2076616c756573
[DEBUG] 0x45d8206e617469766520726561642064206f66
[DEBUG] 0x3022020202020636861727320696e6974
[DEBUG] 0x30220202020202063617365520696e6974
test dojo::benchmarks::bench_storage_single ... ok (gas usage est.: 15318892)
test dojo::world_test::test_upgradeable_world_from_non_owner ... ok (gas usage est.: 1000080)
[DEBUG] 0x16cbeae2020202020646220736574206172 72
[DEBUG] 0x46c8636861727320736572696e6974206 5
[DEBUG] 0x113a2063617365207365726e6974206 5
test dojo::benchmarks::bench_native_storage_offset ... ok (gas usage est.: 19367908)
test dojo::world_test::test_model_class_hash_getter ... ok (gas usage est.: 1468190)
test dojo::benchmarks::bench_simple_struct ... ok (gas usage est.: 23668928)
[DEBUG] 0x53162020206368617273207661 6c756573
[DEBUG] 0x1d9c202020206361736520766 16c756573
[DEBUG] 0x3022020202020202070 6f7320696e6974
test dojo::world_test::test_system_not_writer_fail ... ok (gas usage est.: 1975990)
[DEBUG] 0x5147ca20202063686172732064622 0736574
[DEBUG] 0x1cca2020706f732073657269616c697a65
[DEBUG] 0x62bf8202020206361736520646220736574
[DEBUG] 0x1e993fc2020202020646220676574 20617272
test dojo::world_test::test_array_model ... ok (gas usage est.: 2651310)
[DEBUG] 0xbcd2420202020636173652064622 0676574
[DEBUG] 0x18b02020202020706f732076616c756573
test dojo::world_test::test_upgradeable_world ... ok (gas usage est.: 1091880)
test dojo::world_test::test_upgradeable_world_with_class_hash_zero ... ok (gas usage est.: 993280)
[DEBUG] 0xb05f820202020207 06f7320646220736574
[DEBUG] 0xa1be1c2020206368617273206462206 76574
test dojo::benchmarks::bench_nested_struct ... ok (gas usage est.: 38948314)
[DEBUG] 0xd29c820202020207 06f7320646220676574
test dojo::benchmarks::bench_database_array ... ok (gas usage est.: 76435128)
test dojo::world_test::test_set_metadata_model_writer ... ok (gas usage est.: 4148584)
test dojo::benchmarks::test_struct_with_many_fields ... ok (gas usage est.: 41368318)
test dojo::benchmarks::bench_complex_struct ... ok (gas usage est.: 51954288)
[DEBUG] 0xcd1a8202020666f6f207365742063616c6c
[DEBUG] 0xfc75a2020666f6f20676574206d6163726f
test dojo::world_test::bench_execute ... ok (gas usage est.: 17394190)
test dojo::world_test::test_execute_multiple_worlds ... ok (gas usage est.: 6659796)
test result: ok. 60 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

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.