Initia

Smart Contract Security Assessment

Audit dates: Jan 07 — Jan 28, 2025

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Initia Move smart contract system. The audit took place from January 07 to January 28, 2025.

This audit was judged by LSDan.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Initia Move repository, and is composed of 5 smart contracts written in the Move and Golang programming languages.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

High Risk Findings (4)

[H-01] Domain pricing relies on pool price, which can be manipulated

Submitted by 0xluk3, also found by 0xcb90f054, den-sosnowsky, gss1, Heavyweight_hunters, and p4y4b13

https://github.com/code-423n4/2025-01-initia-move/blob/main/usernames-module/sources/name_service.move#L603

Finding description and impact

Payment for domains (registration, extensions) relies on direct spot price from the Dex module which is directly related to pool reserves. This can be manipulated with a flash loan or a large amount deposit, resulting in:

- buying a domain in a lower price
- making other users overpay for their domains

Calculating the price based directly on a liquidity pool reserves is a well known insecure pattern.

Proof of Concept

In `usernames` module, in function `get_cost_amount`, it calls dex module in line 603:

```
let spot_price = dex::get_spot_price(object::address_to_object<PairConfig>
(@pair), get_init_metadata());
```

Function `get_spot_price` in dex:

```
    #[view]
    /// Calculate spot price
    /// https://balancer.fi/whitepaper.pdf (2)
    public fun get_spot_price(
        pair: Object<Config>, base_coin: Object<Metadata>
    ): BigDecimal acquires Config, Pool, FlashSwapLock {
        let (coin_a_pool, coin_b_pool, coin_a_weight, coin_b_weight, _) =
            pool_info(pair, false);

        let pair_key = generate_pair_key(pair);
        let base_addr = object::object_address(&base_coin);
        assert!(
            base_addr == pair_key.coin_a || base_addr == pair_key.coin_b,
            error::invalid_argument(ECOIN_TYPE)
        );
        let is_base_a = base_addr == pair_key.coin_a;
        let (base_pool, quote_pool, base_weight, quote_weight) =
            if (is_base_a) {
                (coin_a_pool, coin_b_pool, coin_a_weight, coin_b_weight)
            } else {
                (coin_b_pool, coin_a_pool, coin_b_weight, coin_a_weight)
            };

        bigdecimal::div(
            bigdecimal::mul_by_u64(base_weight, quote_pool),
            bigdecimal::mul_by_u64(quote_weight, base_pool)
        )
    }
```

The function uses the pool reserves amounts to calculate the price. Please note, that even if that dex module would implement any lock during the loan, the funds used for manipulation might come from other source, e.g. direct deposit or another dex existing in the future, allowing flash loans.

Recommended mitigation steps

Use a TWAP price source instead, or use an oracle, e.g. Slinky to calculate the price.

andrew (Initia) confirmed and commented:

Flash loan price manipulation is prevented in `dex.move`, but attacks through swaps are still possible. While this makes attacks costly when there is sufficient liquidity, it can be an easy target in the early stages.

Therefore, we plan to hardcode the price as 1 at launch and update it later when slinky adds an initial price. Accordingly, we will modify the current code to hardcode the price as 1 and update it later using the slinky oracle.

[H-02] NFT Token ID contains forbidden character by design which prevents any domain from being issued at all

Submitted by 0xluk3

The `usernames` module allows for registering a domain. This happens in function `register_domain`. On registration, a NFT is minted to the buyer, with field `Token ID` in format `domain:timestamp`. However the `:` character is forbidden by underlying `nft.move` module which is also the reason why original unit tests fail.

Due to this, the protocol cannot be used in its current state, because no NFTs can be currently minted, thus, no domains can be claimed. Hence, this is equivalent to a permanent DoS.

Proof of Concept

In `register_domain`, the `name` string is being appended `.init:timestamp` in line [358-360](358-360).

Running original unit tests, e.g., `initiad move test --log_level trace --path ./usernames-module -f end_to_end`. a debug print was added in line 362: `std::debug::print(&name);`

```
BUILDING Usernames
Running Move unit tests
[debug] "abc.init:100"
[ FAIL    ]
0x42cd8467b1c86e59bf319e5664a09b6b5840bb3fac64f5ce690b5041c530565a::usern
ames::end_to_end

Test failures:

Failures in
0x42cd8467b1c86e59bf319e5664a09b6b5840bb3fac64f5ce690b5041c530565a::usern
ames:

┌── end_to_end ───────
│ error[E11001]: test failure
│    ┌─ /home/move/.move/https___github_com_initia-
labs_movevm_git_b6320db6def0aa9438abdb0e7f3f5bda711c8081/precompile/modul
es/initia_stdlib/sources/token/nft.move:94:37
│    │
│ 86 │     fun assert_token_id(token_id: &String) {
│    │         --------------- In this function in 0x1::nft
│    ·
│ 94 │             error::invalid_argument(EINVALID_TOKEN_ID)
│    │                                     ^^^^^^^^^^^^^^^^^ Test was
not expected to error, but it aborted with code 65545 originating in the
module
00000000000000000000000000000000000000000000000000000000000000001::nft
rooted here
│
│
│ stack trace
│       nft::create(/home/move/.move/https___github_com_initia-
labs_movevm_git_b6320db6def0aa9438abdb0e7f3f5bda711c8081/precompile/modul
es/initia_stdlib/sources/token/nft.move:112)
```

```
 │
initia_nft::mint_internal(/home/move/.move/https___github_com_initia-
labs_movevm_git_b6320db6def0aa9438abdb0e7f3f5bda711c8081/precompile/modul
es/initia_stdlib/sources/token/initia_nft.move:212-219)
 │
initia_nft::mint_nft_object(/home/move/.move/https___github_com_initia-
labs_movevm_git_b6320db6def0aa9438abdb0e7f3f5bda711c8081/precompile/modul
es/initia_stdlib/sources/token/initia_nft.move:189-196)
 │       usernames::register_domain(./sources/name_service.move:365-372)
 │       usernames::end_to_end(./sources/name_service.move:753)
 │
 └──────────────────────
```

Tracking back the error call stack:

- It fails in `mint_nft_object`
    - Then in `initia_nft.move#L189-L196`
    - Then in the same file `212-219`
    - Then in `nft.move#112` on call to `assert_token_id(&token_id);`.

This **function** is provided below:

```
    fun assert_token_id(token_id: &String) {
        let len = string::length(token_id);
        assert!(
            len <= MAX_NFT_TOKEN_ID_LENGTH,
            error::out_of_range(ENFT_TOKEN_ID_TOO_LONG)
        );
        assert!(
            string::index_of(token_id, &string::utf8(b":")) == len,
            error::invalid_argument(EINVALID_TOKEN_ID)
        );
    }
```

In the second assertion it requires `:` to be NOT present in the `token_id`. `index_of` will be equal to `len` only if the searched character will not be present in the string.

Since the name is passed as 4th argument to `mint_nft_object`, which is **defined** as:

```
    public fun mint_nft_object(
        creator: &signer,
        collection: String,
        description: String,
        token_id: String,
        uri: String,
        can_burn: bool
    )
```

Then, using a : by default causes the NFT to be not issued and function reverts.

Using a simple PoC:

```
#[test(chain = @0x1, source = @usernames, user1 = @0x2, user2 = @0x3,
lp_publisher = @0x3)]
fun my_poc_just_register(
    chain: signer,
    source: signer,
    user1: signer,
    user2: signer,
    lp_publisher: signer,
) acquires CoinCapsInit, ModuleStore {
    // Setup environment
    deploy_dex(&chain, &lp_publisher);
    let chain_addr = signer::address_of(&chain);
    let addr1 = signer::address_of(&user1);

    // Give tokens to users for registration
    init_mint_to(chain_addr, &user1, 100);
    init_mint_to(chain_addr, &user2, 100);

    // Initialize the name service with short durations for testing
    initialize(
        &source,
        100,  // price for 3 char
        50,   // price for 4 char
        10,   // default price
        1000, // min duration
        1000, // grace period
        string::utf8(b"https://test.com/"),
        string::utf8(b"https://test.com/"),
    );

    // Set initial block time
    std::block::set_block_info(100, 1000);

    // Step 1: Register all domains
    let domain_name1 = string::utf8(b"abc");
    let domain_name2 = string::utf8(b"def");
    let domain_name3 = string::utf8(b"ghi");
    std::debug::print(&string::utf8(b"\n=== Registering all domains
===")); 
    register_domain(&user1, domain_name1, 1000);
    register_domain(&user1, domain_name2, 1000);
    register_domain(&user1, domain_name3, 1000);
}
```

```
initiad move test --log_level trace --path ./usernames-module -f
my_poc_just_register
```

Line 359 is changed:

```
string::append_utf8(&mut name, b"-");//@audit was ":"
```

Result:

```
Running Move unit tests
[debug] "
=== Registering all domains ==="
[debug] "abc.init-1000"
[debug] "def.init-1000"
[debug] "ghi.init-1000"
[ PASS    ]
0x42cd8467b1c86e59bf319e5664a09b6b5840bb3fac64f5ce690b5041c530565a::usern
ames::my_poc_just_register
```

Recommended mitigation steps

Change the colon `:` to other separator.

andrew (Initia) confirmed and commented:

Fixed this in this commit.

[H-03] User can bypass `MAX_EXPIRATION` when extend expiration

Submitted by laksmana, also found by p4y4b13

https://github.com/code-423n4/2025-01-initia-move/blob/a96f5136c4808f6968564a4592fe2d6ac243a233/usernames-module/sources/name_service.move#L483

Finding Description and Impact

In the `extend_expiration` function, the validation for the duration is incorrect, allowing the user to bypass `MAX_EXPIRATION`:

```
 let expiration_date = metadata::get_expiration_date(token);
        let new_expiration_date = if (expiration_date > timestamp) {
            expiration_date + duration
        } else {
            timestamp + duration
        };

        assert!(
=>            new_expiration_date - expiration_date <= MAX_EXPIRATION,
            error::invalid_argument(EMIN_DURATION),
        );

        metadata::update_expiration_date(token, new_expiration_date);
```

The issue arises because the code subtracts `new_expiration_date - expiration_date` for validation.

Assume a user registers a domain and the `expiration_date` is equal to `MAX_EXPIRATION` + timestamp. Then, the user performs `extend_expiration` with a `duration` value equal to the `MAX_EXPIRATION`, the `new_expiration_date` becomes `expiration_date + duration`.

This leads to the following verification check passing:

```
assert!(
    new_expiration_date - expiration_date <= MAX_EXPIRATION,
    error::invalid_argument(EMIN_DURATION),
);
```

Since the `new_expiration_date` is calculated using `expiration_date` + `duration`, the subtraction `(new_expiration_date - expiration_date)` will always be less than to `MAX_EXPIRATION`.

As a result, the `update_expiration_date` function updates the expiration duration to a value far greater than `MAX_EXPIRATION`, effectively doubling it to `MAX_EXPIRATION * 2`.

```
metadata::update_expiration_date(token, new_expiration_date);
```

Proof of Concept

Add the code below into `name_service.move` and run the test:

```
 #[test(chain = @0x1, source = @usernames, user = @0x2, lp_publisher =
@0x3)]
    fun test_bypass_max_expiration(
        chain: signer,
        source: signer,
        user: signer,
        lp_publisher: signer,
    ) acquires CoinCapsInit, ModuleStore {
        deploy_dex(&chain, &lp_publisher);

        let addr = signer::address_of(&user);
        init_mint_to(signer::address_of(&chain), &user, 1000000000);

        initialize(
            &source,
            100,
            50,
            10,
            1000,
            1000,
            string::utf8(b"https://test.com/"),
            string::utf8(b"https://test.com/"),
        );

        std::block::set_block_info(100, 100);

        // before register
        assert!(get_name_from_address(addr) == option::none(), 0);
        assert!(get_address_from_name(string::utf8(b"abcd")) ==
option::none(), 1);
        assert!(get_valid_token(string::utf8(b"abcd")) == option::none(),
2);

        register_domain(&user, string::utf8(b"abcd"), MAX_EXPIRATION);
        let token =
*option::borrow(&get_valid_token(string::utf8(b"abcd")));
        let token_object = object::address_to_object<Metadata>(token);
        assert!(initia_std::nft::token_id(token_object) ==
string::utf8(b"abcd.init.100"), 3);
        set_name(&user, string::utf8(b"abcd"));
        assert!(get_name_from_address(addr) ==
option::some(string::utf8(b"abcd")), 4);
        assert!(get_address_from_name(string::utf8(b"abcd")) ==
option::some(addr), 5);

        // extend duration bypass a MAX_EXPIRATION
        extend_expiration(&user, string::utf8(b"abcd"), MAX_EXPIRATION);
        let token =
*option::borrow(&get_valid_token(string::utf8(b"abcd")));
        let expiration_date = metadata::get_expiration_date(token);
        assert!( expiration_date >= MAX_EXPIRATION * 2, 6);

    }
```

Recommended Mitigation Steps

Update the validation logic to ensure the `new_expiration_date` itself does not exceed `MAX_EXPIRATION`. The code would look like this:

```
assert!(
    new_expiration_date <= MAX_EXPIRATION,
    error::invalid_argument(EMIN_DURATION),
);
```

andrew (Initia) confirmed and commented:

Actually, `MAX_EXPIRATION` means that you can register/extend to `current time + MAX_EXPIRATION`. And yes, current logic is not correct. So I updated those in this commit.

```
        assert!(
            new_expiration_date - timestamp <= MAX_EXPIRATION,
            error::invalid_argument(EMAX_EXPIRATION),
        );
```

[H-04] Extending a domain's expiration even after the grace period impacts domain buyers

Submitted by p4y4b13

https://github.com/code-423n4/2025-01-initia-move/blob/a96f5136c4808f6968564a4592fe2d6ac243a233/usernames-module/sources/name_service.move#L458

Finding description and impact

The `name_service.move` module allows users to register domain names. If anyone wants to register an already purchased domain, they can only do so once the `expiration_date +`

`grace_period` for that domain has passed. The `name_service.move` module allows anyone to call `extend_expiration` for any domain, which is a feature (according to sponsors).

The main issue is that the `extend_expiration()` function allows users to extend the expiration of a domain even after the grace period has ended, which is unintended behavior.

As a result, users, multi-sig owners of the actual domain name, or attackers can frontrun and attempt to call `extend_expiration()` after the grace period has ended, even if other users are trying to buy the same domain name using `register_domain()`.

This breaks a key invariant of the protocol, leading to genuine users being negatively impacted and experiencing a poor user experience.

Proof of Concept

Consider the following attack scenario:

1. `Alice` buys the domain name `vitalik` for a specific duration.
2. The expiration date and grace period for `Alice`'s domain are completed.
3. `Bob` observes that `Alice`'s domain `vitalik` has passed its `expiration_date + grace_period`. So, `Bob` calls `register_domain` to acquire the `vitalik` domain name.
4. At the same time, `Alice` calls `extend_expiration` for the domain name `vitalik`.
5. Since `Alice` pays a higher gas fee, her transaction is picked and executed first.
6. As a result, `Alice` successfully extends the expiration for her domain, and `Bob`'s transaction reverts.
7. However, `Alice`'s transaction should have been reverted since the grace period had already ended.

Recommended mitigation steps

To mitigate this issue, prevent the `extend_expiration()` function from being called for domains after the completion of `expiration_date + grace_period`.

```
public entry fun extend_expiration(
        account: &signer,
        domain_name: String,
        duration: u64,
    ) acquires ModuleStore {
        ....
        assert!(
            duration >= module_store.config.min_duration,
            error::invalid_argument(EMIN_DURATION),
        );

        let (_height, timestamp) = block::get_block_info();
        let expiration_date = metadata::get_expiration_date(token);
        // @audit Only allow extend_expiration before the grace period
+       assert!(
+           timestamp <= expiration_date +
module_store.config.grace_period ,
+           error::invalid_argument(ECANT_EXTEND_EXPIRATION),
+       );
        ....
    }
```

andrew (Initia) confirmed and commented:

Thank you for reporting. It is fixed on this commit.

Medium Risk Findings (4)

[M-01] Loss of funds due to address mappings are not cleaned up after domain expiry

Submitted by d4r3d3v1l

https://github.com/code-423n4/2025-01-initia-move/blob/main/usernames-module/sources/name_service.move#L350

https://github.com/code-423n4/2025-01-initia-move/blob/main/usernames-module/sources/name_service.move#L158

https://github.com/code-423n4/2025-01-initia-move/blob/main/usernames-module/sources/name_service.move#L175

Finding description and impact

The `register_domain` function doesn't properly clean up old mappings (`name_to_addr and addr_to_name`) when a new user registers an expired domain. While it removes the old `name_to_token` mapping, it leaves the previous user's address mappings.

When a new user re-registers an expired domain:

- The old `name_to_addr,addr_to_name` mappings still points to the previous user values.
- The `is_expired` check in getter functions(`get_address_from_name ,get_name_from_address`) is bypassed because the domain is now re-registered with a new expiration date results in returning previous user values.

Impact

Loss of Funds: `get_address_from_name` returns the address of the previous user.

While sending tokens, we can enter either domain or address. If we enter the domain name, it uses `get_address_from_name` function to retrieve the address. The funds will be transferred to the previous user, not to the one who owns the domain.

Request to fetch the address from the domain name:

```
https://rest.testnet.initia.xyz/initia/move/v1/accounts/0x42cd8467b1c86e5
9bf319e5664a09b6b5840bb3fac64f5ce690b5041c530565a/modules/usernames/view_
functions/get_address_from_name
```

Proof of Concept

Initial State:

- User1 registers domain "abc".
- User1 calls `set_name` for "abc".
- `name_to_addr["abc"]` points to User1's address.
- `addr_to_name[User1's address]` points to "abc".

After Domain Expires:

- User1's registration for "abc" expires.
- Old mappings still exist in the tables:
  - `name_to_addr["abc"]` → User1's address.
  - `addr_to_name[User1's address]` → "abc".

User2 Registers Expired Domain:

- User2 registers "abc".
- A new token is created and assigned to User2.
- Old mappings are not cleaned up:
  - `name_to_addr["abc"]` → Still points to User1's address.
  - `addr_to_name[User1's address]` → Still points to "abc".

Note : User2 didn't call the `set_name`.

Loss of funds scenario: User X want to transfer some tokens to User2 and type the `abc.init` in the address input,the function `get_address_from_name` fetch the User1 address instead and funds will transfer to User1 instead of User2.

Note: I verified the loss of funds scenario in the testnet with the help of [this site](#).

Coded proof of concept

I changed the `end_to_end` test in `name_service.move`:

```
    #[test(chain = @0x1, source = @usernames, user1 = @0x2, user2 = @0x3,
lp_publisher = @0x3)]
    fun end_to_end(
        chain: signer,
        source: signer,
        user1: signer,
        user2: signer,
        lp_publisher: signer,
    ) acquires CoinCapsInit, ModuleStore {
        deploy_dex(&chain, &lp_publisher);
        let chain_addr = signer::address_of(&chain);
        let addr1 = signer::address_of(&user1);
        let addr2 = signer::address_of(&user2);
        init_mint_to(chain_addr, &user1, 100);
        init_mint_to(chain_addr, &user2, 100);

        initialize(
            &source,
            100,
            50,
            10,
            1209600,
            1209600,
            string::utf8(b"https://test.com/"),
            string::utf8(b"https://test.com/"),
        );

std::block::set_block_info(100, 100);

        register_domain(&user1, string::utf8(b"abc"), 31557600);
        assert!(primary_fungible_store::balance(addr1,
get_init_metadata()) == 90, 0);

set_name(&user1, string::utf8(b"abc"));
        assert!(get_name_from_address(addr1) ==
option::some(string::utf8(b"abc")), 0);
        assert!(get_address_from_name(string::utf8(b"abc")) ==
option::some(addr1), 0);

        std::block::set_block_info(200, 100 + 31557600 + 1209600 + 1);
        register_domain(&user2, string::utf8(b"abc"), 31557600);// user2
registering the same domain after expiry

                assert!(get_name_from_address(addr1) ==
option::some(string::utf8(b"abc")), 0); // addr1 -> abc
```

```
        assert!(get_address_from_name(string::utf8(b"abc")) ==
option::some(addr1), 0); // abc -> addr1

}
```

Recommended mitigation steps

Remove the previous values after registering the domain in the `register_domain` function, instead of handling it in `set_name`.

andrew (Initia) confirmed and commented:

The username follows a consistent rule where it is recognized as my domain only if `set_name` is called.

For example, suppose User1 registers the domain `abc.init` and also sets `set_name`. Later, User2 makes a purchase offer, and User1 sells the username NFT. Even in this case, until User2 calls `set_name`, they merely "own" `abc.init`; the domain does not actually point to User2.

The same principle applies to the given scenario. Registering an expired username only means "ownership" and does not mean the domain points to the owner. Let's consider another case: suppose a user registers `abc.init` and calls `set_name`, but also registers `def.init` without calling `set_name`. If they ask another user to send them money, it would be very odd to tell them to send it to `def.init`. Naturally, they would direct them to `abc.init`. Therefore, the rule remains consistent—`set_name` must be set for the domain to point to the user, and an expired domain may not necessarily point to the user.

Nonetheless, to prevent any unintended issues arising from users misunderstanding the username logic, I believe it is reasonable to clear the existing records when registering an expired domain. Based on this reasoning, I have made the necessary changes and submitted this commit.

LSDan (judge) decreased severity to Medium and commented:

This doesn't fit as a high to me. There are several things that need to happen in the right order for this issue to add up to lost funds. That said, I also don't think it can be dismissed (reference here). As a user mistake, since there is no reason any modern tooling would inform the user that this situation exists. I'm open to reasoned arguments this should be lowered further during post-judging QA.

d4r3d3v1l (warden) commented:

As the judge mentioned, this issue requires specific steps to lead to a loss of funds. However, the issue still **breaks a protocol expected functionality** and introduces unintended behaviour which could make it as High.

Explanation:

When a domain expires, `get_address_from_name` correctly returns `None`. However, when another user re-registers the domain, it **unexpectedly returns the previous owner's address**, even though the domain now belongs to the new user. This means the previous owner can still use the expired domain until the new owner explicitly calls `set_name`

This is a unintended behaviour of the protocol. Re-registering a domain should **immediately transfer all rights to the new owner**, but the system retains stale mappings, which breaks the functionality of `get_address_from_name` function. This creates a time gap where the previous owner can still use the expired domain.

Here, "can still use the expired domain" means the previous owner retains access to the domain(for sometime) as if it were active, even after expiration.

Example scenario to describe the impact:

1. Alice buys `abc.init` domain and calls `set_name` (Alice gave this address to John for transactions).
2. John entered `abc.init` and sends a transaction to Alice.
3. Alice domain got expired; now when John tries to do (2), When John entered `abc.init` it will return `None` because it expired.
4. Now another user, Bob, already has `def.init` as primary also want to buy `abc.init` and calls `register_domain` with `abc.init`. There is no successive call to `set_name` after calling `register_domain`. Afterward, Bob continues to use `def.init` as his primary domain. Meanwhile, Bob's `register_domain` call extends the expiration of `abc.init`.

Note : Since Bob did not call `set_name` for `abc.init`, he would not share it with anyone for transactions. As of now, only John has `abc.init`, which was shared by Alice in step (1) .

5. Now, when John tries to send transaction to (Alice) `abc.init`, the system returns Alice's address instead of None in (3).

The issue was downgraded because loss of funds requires multiple steps to occur. However, this example scenario shows another significant impact that **unintended behaviour** of the protocol, which allows the previous owner to still use the **expired** domain for some time. This breaks the expected functionality of the protocol.

LSDan (judge) commented:

Severity stays at Medium.

[M-02] `get_cost_amount` allows unlimited free domain registrations

Submitted by SadBase, also found by danzero

The `get_cost_amount` function unintentionally sets the price for domain names of length greater than or equal to 7 to zero. This behavior is due to the following code logic:

```
let price_per_year =
    if (len == 3) {
        module_store.config.price_per_year_3char
    } else if (len == 4) {
        module_store.config.price_per_year_4char
    } else if (len < FREE_LENGTH) { // FREE_LENGTH = 7
        module_store.config.price_per_year_default
    } else {
        0
    };
```

Here, `FREE_LENGTH` is defined as `7`. When the length of the domain name is greater than or equal to `7`, the `else` branch is executed, setting the `price_per_year` to `0`. While this behavior may be intentional to make longer domain names free, it opens the system to abuse.

Impact

- **Abuse of free registrations:** Users can register unlimited long domain names at zero cost, leading to resource hoarding and potential abuse of the system.
- **Front-running attacks:** Malicious users could register desired long domain names before legitimate users, reducing availability.
- **Griefing:** Attackers can clog the domain name system by registering a large volume of free domains, creating disruptions for legitimate users.

The lack of cost for these registrations incentivizes malicious behavior, as attackers only incur transaction fees.

Proof of Concept

Exploitation Steps:

1. A user selects a domain name of 7 or more characters.
2. The `get_cost_amount` function assigns a cost of `0` due to the `else` branch in the logic.
3. The user registers the domain name for free.

Code Reference:

The issue originates from the following code snippet:

```
if (len < FREE_LENGTH) { // FREE_LENGTH = 7
    module_store.config.price_per_year_default
} else {
    0
}
```

As long as `len >= 7`, the `else` branch ensures the `price_per_year` is zero, enabling free registrations for long domain names.

See code reference from the relevant GitHub repository here.

Recommended mitigation steps

To prevent abuse and ensure proper pricing for all domain names, update the logic in the `get_cost_amount` function as follows:

```
let price_per_year =
    if (len == 3) {
        module_store.config.price_per_year_3char
    } else if (len == 4) {
        module_store.config.price_per_year_4char
    } else if (len < FREE_LENGTH) {
        module_store.config.price_per_year_default
    } else {
        module_store.config.price_per_year_default // Assign a default
cost for longer names
    };
```

andrew (Initia) confirmed and commented:

We decided to remove the free username and just keep 3 prices (3 char, 4 char and 5+ char). Here's the commit.

[M-03] The proposal expiration logic is incorrect

Submitted by Rhaydden

The `is_proposal_expired` function uses incorrect comparison logic that causes proposals to be marked as expired when they should still be active, and vice versa. This is as a result of the reversed comparison operator in the expiration check.

The impact of this bug is high because valid proposals are incorrectly marked as expired which prevents legitimate voting. Also the voting period enforcement is effectively reversed. This effectively creates a DoS because any multisig wallet created would be unable to execute proposals.

N/B: This issue is present in `multisig.move` files in both `initia_stdlib` and `minitia_stdlib`

Proof of Concept

In the `is_proposal_expired` function:

https://github.com/initia-labs/movevm/blob/7096b76ba9705d4d932808e9c80b72101eafc0a8/precompile/modules/minitia_stdlib/sources/multisig.move#L483-L500

```
File: multisig.move
483:     fun is_proposal_expired(
484:         max_period: &Period, proposal_height: u64,
proposal_timestamp: u64
485:     ): bool {
486:         let (height, timestamp) = get_block_info();
487:         let expired_height =
488:             if (option::is_some(&max_period.height)) {
489:                 let max_voting_period_height =
*option::borrow(&max_period.height);
490:  >>>>           (max_voting_period_height + proposal_height) >=
height
491:             } else { false };
492:
493:         let expired_timestamp =
494:             if (option::is_some(&max_period.timestamp)) {
495:                 let max_voting_period_timestamp =
*option::borrow(&max_period.timestamp);
496: >>>>            (max_voting_period_timestamp + proposal_timestamp)
>= timestamp
497:             } else { false };
498:
499:         expired_height || expired_timestamp
500:     }
```

Consider a scenario:

1. Proposal created at height 1000.
2. Voting period is 100 blocks.
3. At height 1050 (which should be valid for voting):
    - Expiration height = `1000 + 100 = 1100`.
    - Current comparison: `(1100 >= 1050)` = true.
    - This incorrectly marks the proposal as expired.
4. The proposal is marked as expired 50 blocks too early.

This can be verified by attaching the test below to `precompile/modules/minitia_stdlib/sources/multisig.move` and run test with `initiad move test --path ./precompile/modules/minitia_stdlib --filter test_proposal_expiration_bug --statistics`.

```
#[test(
    account1 = @0x101, account2 = @0x102, account3 = @0x103
)]
#[expected_failure(abort_code = 0x30005)] // Add this line to expect the
EPROPOSAL_EXPIRED error
fun test_proposal_expiration_bug(
    account1: signer,
    account2: signer,
    account3: signer
) acquires MultisigWallet {
    // create multisig wallet
    let addr1 = signer::address_of(&account1);
    let addr2 = signer::address_of(&account2);
    let addr3 = signer::address_of(&account3);

    // Create multisig with 100 block voting period
    create_multisig_account(
        &account1,
        string::utf8(b"multisig wallet"),
        vector[addr1, addr2, addr3],
        2,
        option::some(100), // max voting period of 100 blocks
        option::none()
    );
    let multisig_addr = object::create_object_address(&addr1, b"multisig
wallet");

    // Set initial block height to 1000
    set_block_info(1000, 0);

    // Create proposal at height 1000
    create_proposal(
        &account1,
        multisig_addr,
        @minitia_std,
        string::utf8(b"multisig"),
        string::utf8(b"update_config"),
        vector[],
        vector[
            std::bcs::to_bytes(&vector[addr1, addr2]),
            std::bcs::to_bytes(&2u64),
            std::bcs::to_bytes(&option::none<u64>()),
            std::bcs::to_bytes(&option::none<u64>())
        ]
    );
    // Move to height 1101 (proposal should be expired)
    // Since 1101 > (1000 + 100)
    set_block_info(1101, 0);

    // This should fail with EPROPOSAL_EXPIRED
    vote_proposal(&account1, multisig_addr, 1, true);
}
```

Console logs:

```
INCLUDING DEPENDENCY MoveNursery
INCLUDING DEPENDENCY MoveStdlib
BUILDING MinitiaStdlib
Running Move unit tests
[ FAIL    ] 0x1::multisig::test_proposal_expiration_bug

Test Statistics:

┌──────────────────────────────────────────────────────┬───────────────┬───
──────────────────────────────┐
│                    Test Name                         │    Time       │          Gas
Used            │
├──────────────────────────────────────────────────────┼───────────────┼───
──────────────────────────────┤
│ 0x1::multisig::test_proposal_expiration_bug │   0.047     │
61103             │
└──────────────────────────────────────────────────────┴───────────────┴───
──────────────────────────────┘

Test failures:

Failures in 0x1::multisig:

┌── test_proposal_expiration_bug ───────
│ Test did not error as expected
└──────────────────────

Test result: FAILED. Total tests: 1; passed: 0; failed: 1
```

After implementing the fix below:

```
INCLUDING DEPENDENCY MoveNursery
INCLUDING DEPENDENCY MoveStdlib
BUILDING MinitiaStdlib
```

```
Running Move unit tests
[ PASS    ] 0x1::multisig::test_proposal_expiration_bug

Test Statistics:

┌───────────────────────────────────────────────────────┬───────────────┬──
──────────────────────────────┐
│                    Test Name                          │     Time      │           Gas
Used            │
├───────────────────────────────────────────────────────┼───────────────┼──
──────────────────────────────┤
│ 0x1::multisig::test_proposal_expiration_bug │   0.054     │
59229             │
└───────────────────────────────────────────────────────┴───────────────┴──
──────────────────────────────┘

Test result: OK. Total tests: 1; passed: 1; failed: 0
```

Recommended mitigation steps

Fix the comparison logic in `is_proposal_expired`:

```
fun is_proposal_expired(
    max_period: &Period, proposal_height: u64, proposal_timestamp: u64
): bool {
    let (height, timestamp) = get_block_info();
    let expired_height =
        if (option::is_some(&max_period.height)) {
            let max_voting_period_height =
*option::borrow(&max_period.height);
-             (max_voting_period_height + proposal_height) >= height
+             height >= (max_voting_period_height + proposal_height)
        } else { false };

    let expired_timestamp =
        if (option::is_some(&max_period.timestamp)) {
            let max_voting_period_timestamp =
*option::borrow(&max_period.timestamp);
-             (max_voting_period_timestamp + proposal_timestamp) >=
timestamp
+             timestamp >= (max_voting_period_timestamp +
proposal_timestamp)
        } else { false };

    expired_height || expired_timestamp
```

```
  }
```

beer-1 (Initia) confirmed, but disagreed with severity and commented:

Good finding; but want to lower the severity to low. Expiration does not affect a lot on the logics and only affect to specific multis account.

LSDan (judge) decreased severity to Medium and commented:

I disagree. This is guaranteed to happen and does impact the desired functionality of the protocol. As such, it fits as a medium. Please let me know if I'm misunderstanding anything.

[M-04] Incorrect request type in oracle currency pairs query whitelist breaks price discovery

Submitted by Rhaydden

In `keepers.go`, the `GetAllCurrencyPairs` query endpoint is misconfigured with an incorrect request type:

The root cause is that the request type is incorrectly set to `GetAllCurrencyPairsResponse` when it should be `GetAllCurrencyPairsRequest`. This mismatch between the expected and actual request types causes a protocol buffer deserialization error when clients attempt to query the supported currency pairs.

The `GetAllCurrencyPairs` endpoint is used during market data initialization and price oracle operations. Looking at the imports and genesis setup, we can see that this is part of the Skip Protocol's oracle system (from `github.com/skip-mev/connect/v2/x/oracle/types`).

A specific use case that would be broken by this bug is the initialization of market data during genesis, as seen in the `AddMarketData` function in `genesis.go`. The oracle genesis state is configured here, and the currency pairs are essential for setting up initial price feeds for trading pairs; configuring which currency pairs can be queried for prices and establishing the baseline for the auction module's pricing functionality.

This would prevent any client (including the app itself) from querying the list of supported currency pairs; which is critical for validators who need to know which pairs they should be providing price data for, trading interfaces that need to display available trading pairs and the auction module which needs to verify valid currency pairs for pricing assets.

This is particularly impactful because while individual price queries (via `/connect.oracle.v2.Query/GetPrice`) would still work, the ability to discover what pairs are actually supported is completely broken; making it impossible to programmatically determine which pairs are valid without hardcoding them.

Proof of concept

Issue is in the query whitelist configuration in [app/keepers/keepers.go]():

```
File: app/keepers/keepers.go
queryWhitelist.Stargate["/connect.oracle.v2.Query/GetAllCurrencyPairs"] = movetypes.ProtoSet{
                Request:  &oracletypes.GetAllCurrencyPairsResponse{}, <-------
                Response: &oracletypes.GetAllCurrencyPairsResponse{},
        }
```

The genesis state initialization in `app/genesis.go` that depends on this functionality:

```
File: app/genesis.go
func (genState GenesisState) AddMarketData(cdc codec.JSONCodec, ac address.Codec) GenesisState {
        var oracleGenState oracletypes.GenesisState
        cdc.MustUnmarshalJSON(genState[oracletypes.ModuleName], &oracleGenState)
      // ... market data initialization depending on currency pairs
```

When clients attempt to query `/connect.oracle.v2.Query/GetAllCurrencyPairs`, the request will fail because:

1. The client sends a `GetAllCurrencyPairsRequest` message.
2. The system attempts to deserialize this into a `GetAllCurrencyPairsResponse` type.
3. The deserialization fails due to message type mismatch.

Recommended mitigation steps

Correct the request type in the query whitelist configuration:

```diff
queryWhitelist.Stargate["/connect.oracle.v2.Query/GetAllCurrencyPairs"] = movetypes.ProtoSet{
-               Request:  &oracletypes.GetAllCurrencyPairsResponse{},
+       Request:  &oracletypes.GetAllCurrencyPairsRequest{},
                Response: &oracletypes.GetAllCurrencyPairsResponse{},
        }
```

[hoon (Initia) commented]():

Revision [here]().

beer-1 (Initia) confirmed, but disagreed with severity and commented:

Good to lower the severity to low.

LSDan (judge) commented:

@beer-1 - I view this as a a valid medium:

2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

This does present to me as an unavailable function of the protocol that is expected to work and important to the workflow of integrators. Please feel free to elaborate as to why you view it as low risk.

Low Risk and Non-Critical Issues

For this audit, 4 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Rhaydden** received the top score from the judge.

The following wardens also submitted reports: 0xluk3, Heavyweight_hunters, and SadBase.

Table of contents

ISSUE ID	DESCRIPTION
[10]	Unchecked withdrawal amount in vault module

Note: C4 excluded the invalid entries determined by the judge from this report.

[01] Unnecessary `clone()` in `ModuleBundle::singleton` causing performance overhead

The `ModuleBundle::singleton` method currently performs an unnecessary clone of the input vector. Since the `code` parameter is already owned (Vec), there's no need to clone it before creating a new Module. This creates an unnecessary memory allocation and copy operation, which can impact performance especially when dealing with large module bytecodes.

https://github.com/initia-labs/movevm/blob/7096b76ba9705d4d932808e9c80b72101eafc0a8/crates/types/src/module.rs#L59-L63

```
pub fn singleton(code: Vec<u8>) -> Self {
    Self {
        codes: vec![Module::new(code.clone())],
    }
}
```

The doesn't allign with other patterns used in other methods of the codebase (like `ModuleBundle::new`), which properly handle ownership without unnecessary cloning.

Add this simple test case to the [`module.rs`]() file:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_singleton_unnecessary_clone() {
        // Create a large vector to make the clone operation more
noticeable
        let large_code = vec![0u8; 1_000_000];
        let initial_capacity = large_code.capacity();

        // Get the memory address of the original vector
        let original_ptr = large_code.as_ptr();

        // Create ModuleBundle using singleton
        let bundle = ModuleBundle::singleton(large_code);

        // Extract the code from the bundle
        let extracted_code = bundle.into_inner().pop().unwrap();

        // Get the memory address of the extracted vector
        let final_ptr = extracted_code.as_ptr();

        // Due to the unnecessary clone(), these pointers will be
different
        // If we remove the clone(), they should be the same
(optimization for move)
        assert_ne!(
            original_ptr,
            final_ptr,
            "Vector was cloned unnecessarily - addresses differ"
        );

        // The capacity should be the same as we're dealing with same-
sized vectors
        assert_eq!(
            initial_capacity,
            extracted_code.capacity(),
            "Capacity should remain the same"
        );
    }

    #[test]
    fn test_singleton_fixed() {
        // This shows how it should work without the clone()
        let large_code = vec![0u8; 1_000_000];
        let original_ptr = large_code.as_ptr();

        // Create a new implementation without clone() for testing
        let bundle = ModuleBundle {
            codes: vec![Module::new(large_code)],
        };

        let extracted_code = bundle.into_inner().pop().unwrap();
        let final_ptr = extracted_code.as_ptr();

        // These pointers should be the same as no clone occurred
        assert_eq!(
            original_ptr,
            final_ptr,
            "Vector was moved without cloning - addresses match"
        );
    }
}
```

Recommended mitigation steps

```
pub fn singleton(code: Vec<u8>) -> Self {
    Self {
-       codes: vec![Module::new(code.clone())],
+       codes: vec![Module::new(code)],
    }
}
```

[02] Wrong telemetry labels for JSON operations in move message server

The Move message server's JSON operation handlers (`ExecuteJSON` and potentially `ScriptJSON`) are using incorrect telemetry labels that conflict with their non-JSON counterparts. This makes it impossible to distinguish between JSON and non-JSON operations in telemetry metrics which is bad for debugging.

https://github.com/initia-labs/initia/blob/10ff76b8394c901e3f5d41350aa9822244c1030b/x/move/keeper/msg_server.go#L101

```
func (ms MsgServer) ExecuteJSON(context context.Context, req
*types.MsgExecuteJSON) (*types.MsgExecuteJSONResponse, error) {
        defer telemetry.MeasureSince(time.Now(), "move", "msg",
"execute")
        ctx := sdk.UnwrapSDKContext(context)
        if err := req.Validate(ms.ac); err != nil {
                return nil, err
        }
```

Recommended mitigation steps

Update the telemetry labels in JSON operation handlers to use distinct labels:

```
func (ms MsgServer) ExecuteJSON(context context.Context, req
*types.MsgExecuteJSON) (*types.MsgExecuteJSONResponse, error) {
-       defer telemetry.MeasureSince(time.Now(), "move", "msg",
"execute")
+       defer telemetry.MeasureSince(time.Now(), "move", "msg",
"execute_json")
        ctx := sdk.UnwrapSDKContext(context)
        if err := req.Validate(ms.ac); err != nil {
                return nil, err
        }
```

[03] `bigdecimal::rev()` lacks explicit division by zero check

The `rev()` function in `bigdecimal.move` doesn't explicitly check for division by zero when calculating the reciprocal of a `BigDecimal`:

Albeit, all other division operations have zero checks:

- `div` checks with `assert!(!biguint::is_zero(num2.scaled),...)`
- `div_by_u64` checks with `assert!(num2 != 0,...)`
- `div_by_u128` checks with `assert!(num2 != 0,...)`
- `div_by_u256` checks with `assert!(num2 != 0,...)`

https://github.com/initia-labs/movevm/blob/7096b76ba9705d4d932808e9c80b72101eafc0a8/precompile/modules/initia_stdlib/sources/bigdecimal.move#L119-L124

```
public fun rev(num: BigDecimal): BigDecimal {
    let fractional = f();
    BigDecimal {
        scaled: biguint::div(biguint::mul(fractional, fractional),
num.scaled)
    }
}
```

Add this test case to `bigdecimal.move`:

```
#[test]
#[expected_failure(abort_code = 0x10065, location = 0x1::biguint)]
fun test_bigdecimal_rev_zero() {
    let num = zero();
    rev(num);
}
```

Test outputs this:

```
INCLUDING DEPENDENCY MoveNursery
INCLUDING DEPENDENCY MoveStdlib
BUILDING InitiaStdlib
Running Move unit tests
[ PASS    ] 0x1::bigdecimal::test_bigdecimal_rev_zero

Test Statistics:

┌──────────────────────────────────────────────────────┬───────────────┬─────
─────────────────────────────┐
│                  Test Name                 │    Time     │          Gas
Used             │
├──────────────────────────────────────────────────────┼───────────────┼─────
─────────────────────────────┤
│ 0x1::bigdecimal::test_bigdecimal_rev_zero │   0.018     │
799              │
└──────────────────────────────────────────────────────┴───────────────┴─────
─────────────────────────────┘

Test result: OK. Total tests: 1; passed: 1; failed: 0
```

`deposit_reward_for_chain` and `unbonding_share_amount_ratio` that call this function have safeguards in place.

Recommended mitigation steps

While it's already protected by the underlying biguint module, it'd still be safe to add explicit checks.

```
public fun rev(num: BigDecimal): BigDecimal {
+    assert!(
+        !biguint::is_zero(num.scaled),
+        error::invalid_argument(EDIVISION_BY_ZERO)
+    );
    let fractional = f();
    BigDecimal {
        scaled: biguint::div(biguint::mul(fractional, fractional),
num.scaled)
    }
}
```

[04] Import of protobuf

As per the docs;

The SDK has migrated from gogo/protobuf (which is currently unmaintained), to our own maintained fork, cosmos/gogoproto.

This means you should replace all imports of github.com/gogo/protobuf to github.com/cosmos/gogoproto. This allows you to remove the `>replace directive replace github.com/gogo/protobuf => github.com/regen-network/protobuf v1.3.3-alpha.regen.1` from your `go.mod` file.

While `go.mod` files for minimove imports protobuf as:

```
    github.com/cosmos/gogoproto v1.7.0
```

And still has the replace directive:

```
    github.com/gogo/protobuf => github.com/regen-network/protobuf
v1.3.3-alpha.regen.1
```

The code is actually already using the new recommended import `github.com/cosmos/gogoproto`, which is good. However, the replace directive for `github.com/gogo/protobuf` is still present and should be removed since you're already using the new import path.

Recommended mitigation steps

1. Remove the replace directive since you're already using the correct import.
2. Verify there are no remaining direct imports of `github.com/gogo/protobuf` in the codebase.
3. Keep using `github.com/cosmos/gogoproto` as you currently are.

[05] Method call syntax in `native_test_only_set_block_info` is not correct

`set_block_info` in `native_test_only_set_block_info` function is incorrectly called as a static method using `NativeBlockContext::set_block_info()` syntax when it's defined as an instance method. This causes a compilation error when the "testing" feature is enabled since the method expects `&mut self` as its first parameter.

https://github.com/initia-labs/movevm/blob/7096b76ba9705d4d932808e9c80b72101eafc0a8/crates/natives/src/block.rs#L63

Recommended mitigation steps

```
fn native_test_only_set_block_info(
     let height = safely_pop_arg!(arguments, u64);

     let block_context = context.extensions_mut().get_mut::
<NativeBlockContext>();
-    NativeBlockContext::set_block_info(block_context, height,
timestamp);
+    block_context.set_block_info(height, timestamp);

     Ok(smallvec![])
 }
```

[06] Inconsistent upgrade name constants may lead to upgrade handler failures

The codebase contains two different constants for the upgrade name:

1. In [`upgrade.go`]: `const upgradeName = "0.6.5"`
2. In [`const.go`]: `const UpgradeName = "0.0.0"`

If any part of the protocol relies on `UpgradeName` for version checks or upgrade logic, it will use an incorrect version ("0.0.0") The duplicate constants violate the DRY (Don't Repeat Yourself) principle and make maintenance error-prone. There could easily be an error because of this.

Recommended mitigation steps

Consider consolidating the upgrade name constants into a single location and use a single source of truth.

[07] Ed25519 public key validation uses size constant (32) instead of error code (1) for invalid length assertion

The `public_key_from_bytes` function incorrectly uses a size constant (`PUBLIC_KEY_SIZE = 32`) as an error code instead of the dedicated error constant `E_WRONG_PUBKEY_SIZE = 1`.

https://github.com/initia-labs/movevm/blob/7096b76ba9705d4d932808e9c80b72101eafc0a8/precompile/modules/initia_stdlib/sources/crypto/ed25519.move#L47

```
// Error code defined but not used
const E_WRONG_PUBKEY_SIZE: u64 = 1;
```

```
// Size constant incorrectly used as error code
const PUBLIC_KEY_SIZE: u64 = 32;

public fun public_key_from_bytes(bytes: vector<u8>): PublicKey {
    assert!(
        std::vector::length(&bytes) == PUBLIC_KEY_SIZE,
        std::error::invalid_argument(PUBLIC_KEY_SIZE) // Incorrectly uses
32 as error code
    );
    PublicKey { bytes }
}
```

Recommended mitigation steps

```
public fun public_key_from_bytes(bytes: vector<u8>): PublicKey {
    assert!(
        std::vector::length(&bytes) == PUBLIC_KEY_SIZE,
+         std::error::invalid_argument(E_WRONG_PUBKEY_SIZE)
-         std::error::invalid_argument(PUBLIC_KEY_SIZE)
    );
    PublicKey { bytes }
}
```

[08] Contradictory `base_path` existence check in clean command

While not directly causing failures, it indicates a flaw in the logic flow and could mask other issues.

Proof of concept

https://github.com/initia-labs/movevm/blob/7096b76ba9705d4d932808e9c80b72101eafc0a8/crates/compiler/src/clean.rs#L100-L103

```
if !base_path.exists() || !build_path.exists() {
    return Ok(());
}
```

In the `validate_manifest` function of the Move VM's clean command, there is a contradiction where it checks for the non-existence of the base path (`!base_path.exists()`) after already having used this path to verify and read the `Move.toml` file. We've already used base_path to find and read the Move.toml file earlier in the function. If base_path doesn't exist, the function would have already failed at the manifest check.

This condition would never be true for base_path since we've already confirmed its existence by reading the manifest from it.

Recommended mitigation steps

Remove the redundant base path check and only verify the build path existence:

```
// Replace this:
if !base_path.exists() || !build_path.exists() {
    return Ok(());
}

// With this:
if !build_path.exists() {
    return Ok(());
}
```

[09] Wrong error code is used for maximum name length validation

In the check_name function of the name service module, when validating the maximum length of a domain name, the wrong error code is being used. The function uses EMIN_NAME_LENGTH for both minimum and maximum length validations which is wrong.

https://github.com/code-423n4/2025-01-initia-move/blob/a96f5136c4808f6968564a4592fe2d6ac243a233/usernames-module/sources/name_service.move#L556

```
fun check_name(name: String) {
    let bytes = string::bytes(&name);
    let len = vector::length(bytes);
    assert!(len >= 3, error::invalid_argument(EMIN_NAME_LENGTH));
    assert!(len <= MAX_LENGTH,
error::invalid_argument(EMIN_NAME_LENGTH)); // Wrong error code
    ...snip
}
```

When a user attempts to register a domain name longer than MAX_LENGTH (64 characters), they will receive an error message suggesting the name is too short (EMIN_NAME_LENGTH: "name length must be more bigger than or equal to 3") rather than the correct error message indicating the name is too long.

Recommended mitigation steps

```
fun check_name(name: String) {
    let bytes = string::bytes(&name);
    let len = vector::length(bytes);
    assert!(len >= 3, error::invalid_argument(EMIN_NAME_LENGTH));
-     assert!(len <= MAX_LENGTH,
error::invalid_argument(EMIN_NAME_LENGTH));
+     assert!(len <= MAX_LENGTH,
error::invalid_argument(EMAX_NAME_LENGTH));
    ...snip
}
```

[10] Unchecked withdrawal amount in vault module

The function is already restricted with `friend` access and the underlying `primary_fungible_store::withdraw` includes balance checks.

Proof of concept

The `withdraw` function in `vault.move` allows withdrawing funds without checking if the requested amount is available in the vault:

https://github.com/code-423n4/2025-01-initia-move/blob/a96f5136c4808f6968564a4592fe2d6ac243a233/vip-module/sources/vault.move#L53-L63

```
public(friend) fun withdraw(amount: u64): FungibleAsset acquires
ModuleStore {
    let module_store = borrow_global_mut<ModuleStore>(@vip);
    assert!(
        module_store.reward_per_stage > 0,
        error::invalid_state(EINVALID_REWARD_PER_STAGE),
    );
    let vault_signer =
        object::generate_signer_for_extending(&module_store.extend_ref);
    primary_fungible_store::withdraw(&vault_signer, reward_metadata(),
amount)
}
```

While this might be safe because `primary_fungible_store::withdraw` in `vip.move` includes internal balance checks, it's still best practice to validate the withdrawal amount against the available balance at the vault level for explicit safety guarantees.

Recommended mitigation steps

Add a balance check before performing the withdrawal.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.