Security Assessment for IKE

Report by Ulam Labs

IKE

Findings and Recommendations Report Presented to:

Water Cooler Studios Inc

October 21, 2024 Version: 1.0

Presented by:

Ulam Labs

Grabiszynska 163/502,

53-332 Wroclaw, POLAND

Executive Summary

Overview

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Ulam Labs Security Teams took to identify and validate each issue, and any applicable recommendations for remediation.

Scope

The audit has been conducted for the following WaterCoolerStudiosInc public GitHub repository:

WaterCoolerStudiosInc/ike-contracts:
2f824396d40aa37f56acfcbddbfd063c82de907d

Contracts in scope:

- vault
- share_token
- registry
- nomination_agent

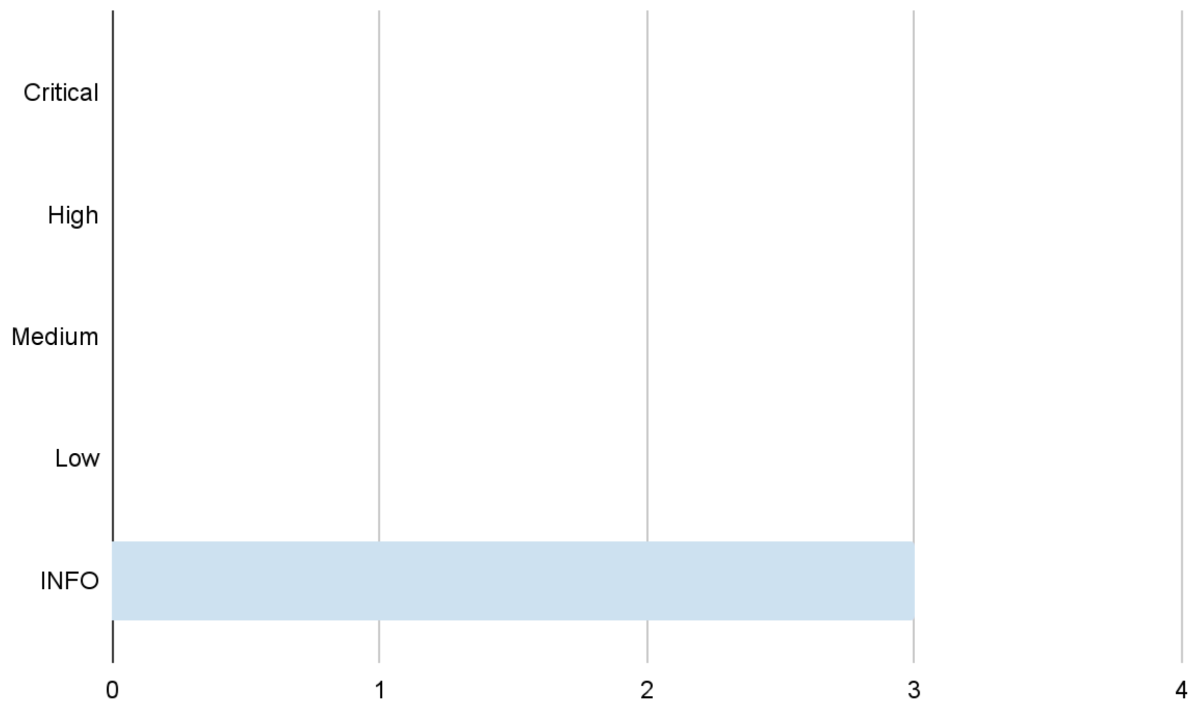

Chart 1: Findings by severity.

Key findings

During the Security Assessment, the following findings were discovered:

- 0 findings with a CRITICAL severity rating,
- 0 findings with a HIGH severity rating,
- 0 findings with a MEDIUM severity rating,
- 0 findings with a LOW severity rating.
- 3 findings with an INFO severity rating.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any potential consequences of the deployment or use of the contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding several components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims that impact a large quantity of funds.

Technical analysis & findings

General observations

The audit started very smoothly, as build instructions were clear and everything complied successfully.
There are plenty of helpful comments in the code and a lot of tests.
Documentation was detailed and almost 100% aligned with the current state.
Smart contracts are using the old (v4.3) version of ink, but it is stable and supported, so it is not a problem.
During the audit process, the development team was perfectly supportive and responsive.
Despite many tries including intensive code review, access control stress testing, z3 proofs, and fuzzing tests, no issue requiring a fix was found.
The issues presented below are just informative because contracts are ready for deployment as is.

Delegate bonding promotes the worst-performing nominators

Finding ID: **IKE-1**
File: vault/data.rs
Severity: Info
Status: Acknowledged

Description

When user stakes, nominators with negative imbalances are prioritized. Such behavior is desired to align nominators' holding with their weight.

Impact

Distributing tokens in that way has some side effects. Nominators, whose performance is the worst, get the most tokens, making the algorithm not effective.

Recommended Mitigation Steps

Without an advanced strategy, it is not possible to solve both problems. That's why developers decided to delegate that problem off-chain. The operator of the contract (DAO) can then adjust weight based on profit. As there is no slashing on Aleph Zero, there is no risk any funds can be lost.

Update agent's message panic on integer overflow.

Finding ID: **IKE-2**
File: registry/lib.rs
Severity: Info
Status: Acknowledged

Description

In rust, overflow checks are turned off by default if the binary is built in release mode. It is very dangerous, as the result of some calculations can be unexpected.

An example of such a possibility is updated agent messages. As total weights have the same size as single agent weight, their sum can easily overflow.

Impact

The impact of such a scenario is still not so serious, as this endpoint is controlled by a trusted party, and after tests it turned out, that overflow checks are on for the contract ready to deploy, so the only problem here is not elegant error reporting.

Recommended Mitigation Steps

It is good practice to avoid such situations as overflow checks can be accidentally turned off after some updates and for better error reporting.

However, as no vulnerabilities have been identified, it is recommended not to make any changes.

Vault can be blocked if a fee is too high for a very long time

Finding ID: **IKE-3**
File: Royalties.sol
Severity: Info
Status: Acknowledged

Description

As mentioned above, overflow checks are on. Most of the time it solves some problems, but sometimes it has some negative effects. If something overflows permanently, such a path of execution can be considered blocked.

One example of such overflow is fee calculation. As fees can grow exponentially, especially if set to a high value, the contract can be bricked completely if upgradeability is not possible anymore.

Impact

It can sound serious, but the impact of this issue is heavily reduced by limiting the maximum fee (100%).

In the worst-case scenario, for big pools, it would take ~60 years to block funds on such pools. It gives a lot of time for users to react.

Recommended Mitigation Steps

To completely mitigate such an issue, it is recommended to use saturating add/multiply to avoid integer overflow.

As the issue is almost impossible, no changes in code are required.

Other

Fuzzing tests are used to detect the issues and check the properties

To use tests in native rust (to improve performance), the source code was heavily modified to use global variables to simulate side effects.

```
#[ink::test]
fn whole_system() {
  let mut g = rand::rngs::StdRng::seed_from_u64(123456);
  let n = 10_000;
  for _ in 0..n
  {
    let n = g.next_u32() % 10_000_0;
    let era : u64 = (u32::MAX as u64) / 1_000_000;
    let vault_id = AccountId::from([0xf1; 32]);
    let token_id = AccountId::from([0xf2; 32]);
    let admin = AccountId::from([0xf5; 32]);

    set_callee::<DefaultEnvironment>(vault_id);
    set_caller2(admin);
    let mut vault = Vault::new(Hash::default(), Hash::default(), Hash::default(), era);

    let mut balances: [u128; 128] = [0; 128];
    let mut noms : Vec<u128> = vec![];
    let mut agents : Vec<AccountId> = vec![];
    let mut shares: [u128; 128] = [0; 128];
    let mut unlocks : Vec<Vec<(u64, u128)>> = vec![];
    let mut total_fees : u128 = 0;

    let mut total_balance : u128 = 0;
    for _ in 0..128 {
      unlocks.push(vec![]);
    }


    let timestamp0 : u64 = 1730122908;
    let mut timestamp  = timestamp0;

    for _ in 0..n {
      let user_id = (g.next_u32() % 128) as usize;
      let user = AccountId::from([user_id as u8; 32]);

      set_callee::<DefaultEnvironment>(vault_id);
      unsafe {
        CALLER = Some(user);
      }

      let amount = g.next_u64() as u128;
      set_caller2(admin);
      set_callee::<DefaultEnvironment>(AccountId::from([0xf3; 32]));
      set_value_transferred::<DefaultEnvironment>(amount);
      unsafe {
        match REGISTRY.as_mut().unwrap().add_agent(admin, admin) {
          Err(RegistryError::TooManyAgents) => (),
          Err(_) => panic!("'add_agent' unexpected error"),
          Ok(agent) => {
            noms.push(amount);
            agents.push(agent);
          },
        }
      }


      let action = g.next_u32() % 7_400;
      if action < 100 {
        total_fees += (timestamp - timestamp0) as u128 * total_balance / 50 / (365*24*60*60);
        timestamp += (g.next_u32() as u64) / 1_000_000;
        set_block_timestamp::<DefaultEnvironment>(timestamp);
      } else if action < 2_100 {
        set_caller2(user);
        balances[user_id] += amount;
        let new_shares0 = vault.get_shares_from_azero(amount);
        shares[user_id] += vault.get_shares_from_azero(amount);

        set_callee::<DefaultEnvironment>(vault_id);
        set_value_transferred::<DefaultEnvironment>(amount);
        match vault.stake() {
          Err(VaultError::MinimumStake) => if amount >= 1_000_000 {
            panic!("'stake' should have failed with MinimumStake")
          },
          Err(VaultError::ZeroTotalWeight) => (),
          Ok(new_shares) => {
            assert!(new_shares == new_shares0);
            total_balance += amount;
          },
          _ => panic!("unexpected error"),
        };
      } else if action < 4_100 {
        // let amount2 = amount %  (shares[user_id] + 1);
        let amount2 = shares[user_id];
        set_caller2(user);
        set_callee::<DefaultEnvironment>(vault_id);
        let azero = unsafe {
          nomination_agent::nomination_agent::UNBOND_NOW = Some(0);

          vault.get_azero_from_shares(amount2)
        };
        match vault.request_unlock(amount2) {
          Ok(_) => {
            shares[user_id] -= amount2;
            unlocks[user_id].push((timestamp + era * 14, azero));
          },
          Err(VaultError::ZeroUnbonding) => {
            shares[user_id] -= amount2;
          },
          _ => panic!("unexpected error"),
        }
      } else if action < 4_150 {
        set_caller2(admin);
        set_callee::<DefaultEnvironment>(AccountId::from([0xf3; 32]));
        set_value_transferred::<DefaultEnvironment>(amount);
        unsafe {
          match REGISTRY.as_mut().unwrap().add_agent(admin, admin) {
            Err(RegistryError::TooManyAgents) => (),
            Err(_) => panic!("'add_agent' unexpected error"),
            Ok(agent) => {
              noms.push(amount);
              agents.push(agent);
            },
          }
        }
      } else if action < 4_200 {
        set_caller2(admin);
        set_callee::<DefaultEnvironment>(AccountId::from([0xf3; 32]));
        let mut weights = vec![];
        for _ in agents.iter() {
          weights.push(g.next_u32() as u64);
        }
        unsafe {
          match REGISTRY.as_mut().unwrap().update_agents(agents.clone(), weights) {
            Err(_) => panic!("'update_agent' unexpected error"),
            Ok(_) => (),
          }
        }
      } else if action < 4_400 {
        set_caller2(admin);
        set_callee::<DefaultEnvironment>(vault_id);
        match vault.withdraw_fees() {
          Err(_) => panic!("'withdraw_fees' unexpected error"),
          Ok(_) => {
          },
        }
      } else if action < 6_400 {
        set_caller2(admin);
        set_callee::<DefaultEnvironment>(vault_id);
        if unlocks[user_id].len() != 0 {
          let unlock_id = g.next_u64() % unlocks[user_id].len() as u64;
          let azero = unsafe {
            let azero = unlocks[user_id][unlock_id as usize].1;
            nomination_agent::nomination_agent::UNBOND_NOW = Some(azero);
            azero
          };
          match vault.redeem_with_withdraw(user, unlock_id) {
            Err(VaultError::CooldownPeriod) => if timestamp >= unlocks[user_id][unlock_id as 
usize].0 {
              panic!("'redeem_with_withdraw' unexpected CooldownPeriod")
            },
            Err(_) => panic!("'redeem_with_withdraw' unexpected error"),
            Ok(_) => {
              unlocks[user_id].remove(unlock_id as usize);
              balances[user_id] += azero;
              total_balance -= azero;
            },
          }
        }
      } else if action < 7_400 {
        set_caller2(admin);
        set_callee::<DefaultEnvironment>(vault_id);
        let azero = unsafe {
          let azero = g.next_u64() as u128;
          nomination_agent::nomination_agent::UNBOND_NOW = Some(azero);
          azero
        };
        match vault.compound() {
          Err(VaultError::ZeroCompounding) => {
            unsafe {
              if nomination_agent::nomination_agent::UNBOND_NOW.is_some() {
                panic!("'compound' unexpected ZeroCompounding")
              }
            }
          },
          Err(_) => panic!("'compound' unexpected error"),
          Ok(amount) => {
            assert!(amount == azero, "compound  amount mismatch");
            total_balance += azero;
          },
        }
      }
    }

    for user_id in 0..128 {
      let user = AccountId::from([user_id as u8; 32]);
      set_caller2(user);
      set_callee::<DefaultEnvironment>(vault_id);

      let azero = vault.get_azero_from_shares(shares[user_id]);
      match vault.request_unlock(shares[user_id]) {
        Err(VaultError::ZeroUnbonding) => (),
        Ok(_) => {
          unlocks[user_id].push((timestamp + era * 14, azero));
          total_balance -= azero;
        },
        _ => panic!("final 'request_unlock' unexpected error"),
      }
    }
    timestamp += era * 14;
    set_block_timestamp::<DefaultEnvironment>(timestamp);


    for user_id in 0..128 {
      let user = AccountId::from([user_id as u8; 32]);
      set_caller2(user);
      set_callee::<DefaultEnvironment>(vault_id);

      for unlock_id in 0..unlocks[user_id].len() {
        set_caller2(admin);
        set_callee::<DefaultEnvironment>(vault_id);
        let azero = unsafe {
          let azero = unlocks[user_id][unlock_id].1;
          nomination_agent::nomination_agent::UNBOND_NOW = Some(azero);
          azero
        };
        match vault.redeem_with_withdraw(user, 0) {
          Err(e) => panic!("'redeem_with_withdraw' unexpected error: {:?}", e),
          Ok(_) => {
            total_balance -= azero;
          },
        }
      }
      }
    assert!(total_balance >= total_fees);
  }
}
```

Access control verification

Vault contract

```
#[ink::test]
fn test_access_control() {
  let era = 1;
  let admin = AccountId::from([0x1; 32]);
  let someone = AccountId::from([0x2; 32]);
  let admin_adj_fee = AccountId::from([0x3; 32]);
  let admin_fee_to = AccountId::from([0x4; 32]);
  let vault_id = AccountId::from([0x5; 32]);
  let admin_mig = AccountId::from([0x6; 32]);
  let callers = [admin, someone, admin_adj_fee, admin_fee_to, admin_mig];

  set_callee::<DefaultEnvironment>(vault_id);
  set_caller::<DefaultEnvironment>(admin);
  let mut vault = Vault::new(Hash::default(), Hash::default(), Hash::default(), era);

  set_caller::<DefaultEnvironment>(admin);
  vault.transfer_role_adjust_fee(admin_adj_fee).unwrap();
  vault.transfer_role_fee_to(admin_fee_to).unwrap();
  vault.transfer_role_set_code(admin_mig).unwrap();

  for caller in callers {
    if caller != admin_adj_fee {
      set_caller::<DefaultEnvironment>(caller);
      match vault.transfer_role_adjust_fee(someone) {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'transfer_role_adjust_fee' should have failed with InvalidPermissions"),
      };
      match vault.adjust_fee(0) {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'adjust_fee' should have failed with InvalidPermissions"),
      };
    }
  }

  for caller in callers {
    if caller != admin_fee_to {
      set_caller::<DefaultEnvironment>(caller);
      match vault.transfer_role_fee_to(someone) {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'transfer_role_fee_to' should have failed with InvalidPermissions"),
      };
      match vault.withdraw_fees() {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'withdraw_fees' should have failed with InvalidPermissions"),
      };
    }
  }
  for caller in callers {
    if caller != admin_mig {
      set_caller::<DefaultEnvironment>(caller);
      match vault.transfer_role_set_code(someone) {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'transfer_role_set_code' should have failed with InvalidPermissions"),
      };
      match vault.set_code([0;32]) {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'set_code' should have failed with InvalidPermissions"),
      };
      match vault.disable_set_code() {
        Err(VaultError::InvalidPermissions) => (),
        _ => panic!("'disable_set_code' should have failed with InvalidPermissions"),
      };
    }
  }
}
```

Share token contract

```
#[ink::test]
fn test_access_control() {
  let vault = AccountId::from([0x1; 32]);
  let token_id = AccountId::from([0x2; 32]);
  let someone = AccountId::from([0x3; 32]);
  let someone2 = AccountId::from([0x4; 32]);
  let callers = [vault, someone];

  set_caller::<DefaultEnvironment>(vault);
  set_callee::<DefaultEnvironment>(token_id);
  let mut token = Token::new(None, None);

  for caller in callers {
    set_caller::<DefaultEnvironment>(caller);
    if caller != vault {
      match token.mint(someone, 100) {
        Err(PSP22Error::Custom(_)) => (),
        _ => panic!("'mint' should have failed with Unauthorized")
      }
      match token.burn(100) {
        Err(PSP22Error::Custom(_)) => (),
        _ => panic!("'burn' should have failed with Unauthorized")
      }
    }
  }
  for caller in callers {
    set_caller::<DefaultEnvironment>(caller);
    match token.transfer(someone2, 1, vec![]) {
      Err(PSP22Error::InsufficientBalance) => (),
      _ => panic!("'transfer' should have failed with Insufficient funds")
    }
  }
  set_caller::<DefaultEnvironment>(vault);
  token.mint(someone2, 1).unwrap();

  for caller in callers {
    if caller != vault {
      set_caller::<DefaultEnvironment>(caller);
      match token.transfer_from(someone2, someone, 1, vec![]) {
        Err(PSP22Error::InsufficientAllowance) => (),
        _ => panic!("'transfer_from' should have failed with Insufficient allowance")
      }
    }
  }
  for caller in callers {
    set_caller::<DefaultEnvironment>(caller);
    token.approve(someone2, 1).unwrap();
    token.increase_allowance(someone2, 1).unwrap();
    token.decrease_allowance(someone2, 1).unwrap();
  }
}
```

Registry contract

```
#[ink::test]
fn test_access_control() {
  let admin_add = AccountId::from([0x1; 32]);
  let op_add = AccountId::from([0x2; 32]);
  let admin_up = AccountId::from([0x3; 32]);
  let op_up = AccountId::from([0x4; 32]);
  let admin_rm = AccountId::from([0x5; 32]);
  let op_rm = AccountId::from([0x6; 32]);
  let admin_mig = AccountId::from([0x7; 32]);
  let op_mig = AccountId::from([0x8; 32]);
  let vault = AccountId::from([0x9; 32]);
  let registry_id = AccountId::from([0x10; 32]);
  let someone = AccountId::from([0x11; 32]);
  let callers = [admin_add, op_add, admin_up, op_up, admin_rm, op_rm, admin_mig, op_mig, vault,
someone];

  set_callee::<DefaultEnvironment>(registry_id);
  set_caller::<DefaultEnvironment>(vault);
  let mut registry = Registry::new(admin_add, admin_up, admin_rm, admin_mig, Hash::default());

  set_caller::<DefaultEnvironment>(admin_add);
  registry.transfer_role(RoleType::AddAgent, op_add).unwrap();

  set_caller::<DefaultEnvironment>(admin_up);
  registry.transfer_role(RoleType::UpdateAgents, op_up).unwrap();

  set_caller::<DefaultEnvironment>(admin_rm);
  registry.transfer_role(RoleType::RemoveAgent, op_rm).unwrap();

  set_caller::<DefaultEnvironment>(admin_mig);
  registry.transfer_role(RoleType::SetCodeHash, op_mig).unwrap();

  for caller in callers {
    set_caller::<DefaultEnvironment>(caller);
    if caller != admin_add {
      match registry.transfer_role(RoleType::AddAgent, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(AddAgent)' should have failed with InvalidPermissions"),
      };
      match registry.transfer_role_admin(RoleType::AddAgent, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role_admin(AddAgent)' should have failed with
InvalidPermissions"),
      };
    }
    if caller != admin_up {
      match registry.transfer_role(RoleType::UpdateAgents, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(UpdateAgents)' should have failed with
InvalidPermissions"),
      };
      match registry.transfer_role_admin(RoleType::UpdateAgents, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(UpdateAgents)' should have failed with
InvalidPermissions"),
      };
    }
    if caller != admin_rm {
      match registry.transfer_role(RoleType::RemoveAgent, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(RemoveAgent)' should have failed with InvalidPermissions"),
      };
      match registry.transfer_role_admin(RoleType::RemoveAgent, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(RemoveAgent)' should have failed with InvalidPermissions"),
      };
    }
    if caller != admin_mig {
      match registry.transfer_role(RoleType::SetCodeHash, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(SetCodeHash)' should have failed with InvalidPermissions"),
      };
      match registry.transfer_role_admin(RoleType::SetCodeHash, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'transfer_role(SetCodeHash)' should have failed with InvalidPermissions"),
      };
    }
    if caller != op_add {
      match registry.add_agent(someone, someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'add_agent' should have failed with InvalidPermissions"),
      };
    }
    if caller != op_up {
      match registry.update_agents(vec![], vec![]) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'update_agents' should have failed with InvalidPermissions"),
      };
    }
    if caller != op_rm {
      match registry.remove_agent(someone) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'remove_agent' should have failed with InvalidPermissions"),
      };
    }
    if caller != op_mig {
      match registry.set_code([0; 32]) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'set_code' should have failed with InvalidPermissions"),
      };
      match registry.set_agent_code([0; 32]) {
        Err(RegistryError::InvalidPermissions) => (),
        _ => panic!("'set_agent_code' should have failed with InvalidPermissions"),
      };
    }
  }
}
```

```
#[ink::test]
  fn test_access_control() {
      let vault = AccountId::from([0x1; 32]);
      let admin = AccountId::from([0x2; 32]);
      let validator = AccountId::from([0x3; 32]);
      let nominator_id = AccountId::from([0x4; 32]);
      let someone = AccountId::from([0x5; 32]);
      let registry = AccountId::from([0x6; 32]);
      let callers = [vault, admin, validator, someone, registry];

      set_caller::<DefaultEnvironment>(registry);
      set_callee::<DefaultEnvironment>(nominator_id);

      let mut nominator = NominationAgent::new(vault, admin, validator);

      for caller in callers {
        set_caller::<DefaultEnvironment>(caller);
        if caller != vault {
          match nominator.deposit() {
              Err(RuntimeError::Unauthorized) => (),
              _ => panic!("'deposit' should have failed with Unauthorized")
          };
          match nominator.start_unbond(0) {
              Err(RuntimeError::Unauthorized) => (),
              _ => panic!("'start_unbound' should have failed with Unauthorized")
          }
          match nominator.withdraw_unbonded() {
              Err(RuntimeError::Unauthorized) => (),
              _ => panic!("'withdraw_unbonded' should have failed with Unauthorized")
          }
          match nominator.compound() {
              Err(RuntimeError::Unauthorized) => (),
              _ => panic!("'compound' should have failed with Unauthorized")
          }
        }
      }

      for caller in callers {
        set_caller::<DefaultEnvironment>(caller);
        if caller != registry {
          match nominator.destroy() {
              Err(RuntimeError::Unauthorized) => (),
              _ => panic!("'destroy' should have failed with Unauthorized")
          }
        }
      }

      for caller in callers {
        set_caller::<DefaultEnvironment>(caller);
        if caller != admin {
          match nominator.admin_withdraw_bond(caller) {
              Err(RuntimeError::Unauthorized) => (),
              _ => panic!("'admin_withdraw_bond' should have failed with Unauthorized")
          }
        }
      }
  }
```

Severity classification

We have adopted a severity classification inspired by the Immunefi Vulnerability Severity Classification System - v2. It can be found here.