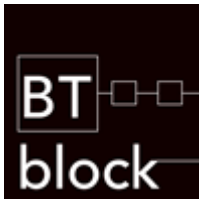

Smart Contract Security Assessment

EnergyWeb

December 2021
Version: 1.1

Presented by:
BTBlock LLC

Corporate Headquarters
BTBlock LLC
2720 Xavier Street
Denver, CO 80212

Security level: Public

Table of Contents

Table of Figures

Table of Tables

EXECUTIVE SUMMARY

Overview

EnergyWeb engaged BtBlock LLC to perform a Security Assessment for its Smart Contracts.

The assessment was conducted remotely by the BTBlock Security Team. Testing took place on November 07 - December 07, 2021, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks discovered within the environment during the engagement.
- To provide a professional opinion on the security measures' maturity, adequacy, and efficiency.
- To identify potential issues and include improvement recommendations based on the result of our tests.

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

Key Findings

The following are the major themes and issues identified during the testing period. Within the findings section, these, along with other items, should be prioritised for remediation to reduce the risk they pose.

- KS-ENERGYWEB-01 – Staking Pool - Owners can reinitialise pools
- KS-ENERGYWEB-02 – ClaimsManager.sol - Replay attack on registration
- KS-ENERGYWEB-03 – Staking Pool - Recorded staking time does not follow rewards' hourly subdivision
- KS-ENERGYWEB-04 – ClaimsManager - Any issuer can change the expiry date
- KS-ENERGYWEB-05 – ClaimManager - Address Cast to Abstract Contracts
- KS-ENERGYWEB-06 – ClaimManager - TTL of ENSRegistry is not checked

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was very supportive and open to discussing the design choices made

We can conclude that the reviewed code implements the documented functionality based on the formal verification.

Scope and Rules of Engagement

BTBlock performed a Security Assessment for EnergyWeb. The following table documents the targets in scope for the engagement. No other systems or resources were in scope for this assessment.

The source code was supplied through multiple private repositories at:

- https://gitlab.com/btblock-cybersec/energyweb/iam-contracts with the commit hash b2564e002d1e212116a0e88e9bba0a61da7ec582.
- https://gitlab.com/btblock-cybersec/energyweb/ethr-did-registry 1499a472ca56a42a3939e76652a06c4cd831dac0
- https://gitlab.com/btblock-cybersec/energyweb/resolvers 4a3d2097a8ab0da8c1e2c39480c908b1083c1f4e
- https://gitlab.com/btblock-cybersec/energyweb/staking-pool 83e1cabbad4897b8f4839f57125cdf202ac0e1bf

Files audited include the following, together with the same project dependencies:

```
.
├── ethr-did-registry/
│   └── contracts/
│       └── EthereumDIDRegistry.sol
├── iam-contracts/
│   └── contacts/
│       └── roles/
│           └── ClaimManager.sol
├── resolvers/
│   └── contacts/
│       └── PublicResolver.sol
└── staking-pool/
    └── contacts/
        └── StakingPool.sol
```

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the Security Assessment for EnergyWeb, we discovered:

- 2 findings with a MEDIUM severity rating.
- 2 findings with a LOW severity rating.
- 2 findings with an INFORMATIONAL severity rating.

The following chart displays the findings by severity.

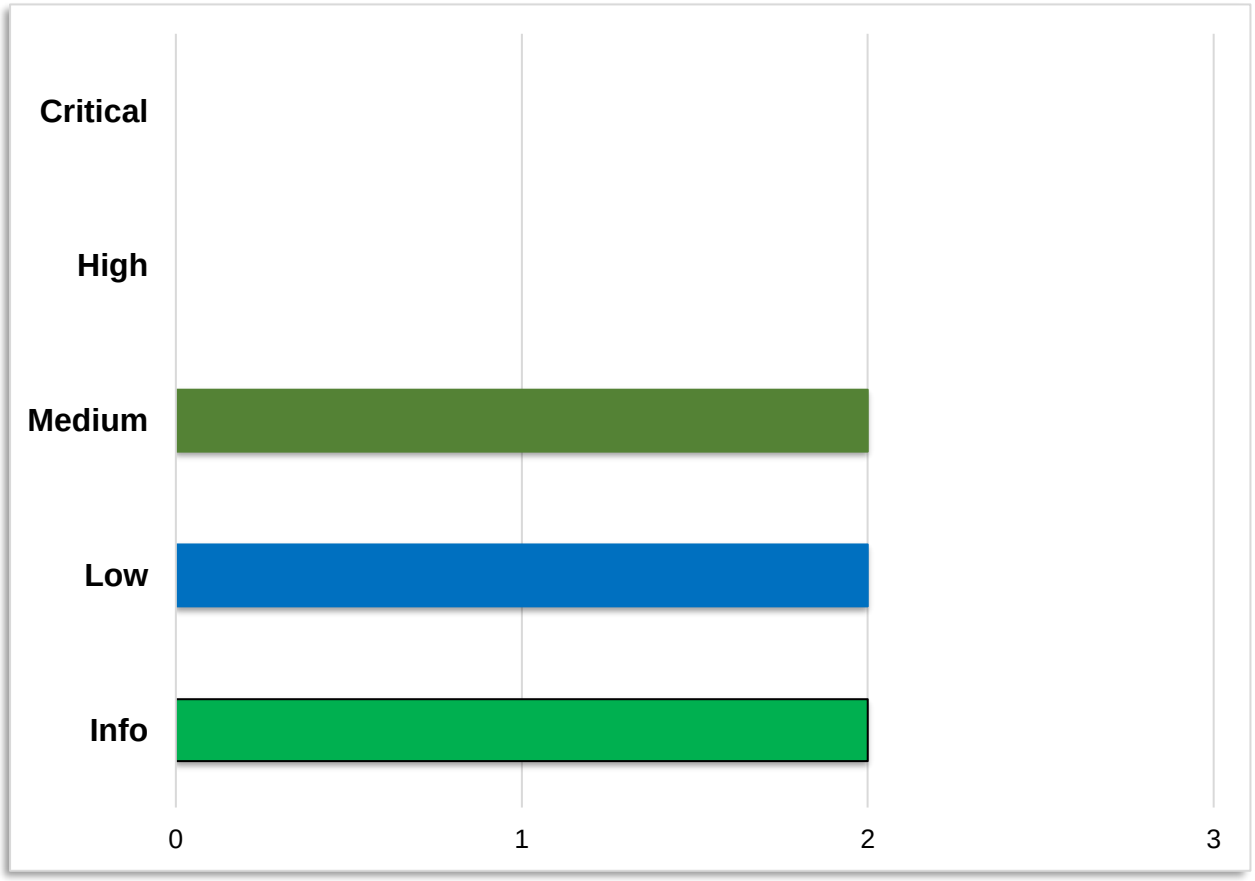

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each finding, including discovery methods, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-ENERGYWEB-01	**Medium**	Staking Pool - Onwers can reinitialize pools
KS-ENERGYWEB-02	**Medium**	ClaimsManager.sol - Replay attack on registration
KS-ENERGYWEB-03	**Low**	Staking Pool - Recorded staking time does not follow rewards' hourly subdivision
KS-ENERGYWEB-04	**Low**	ClaimsManager - Any issuer can change the expiry date
KS-ENERGYWEB-05	**Informational**	ClaimManager - Address Cast to Abstract Contracts
KS-ENERGYWEB-06	**Informational**	ClaimManager - TTL of ENSRegistry is not checked

Table 2: Findings Overview

Technical analysis

Based on the source code, the validity of the code was verified and confirmed that the intended functionality was implemented correctly and to the extent that the state of the repository allowed.

Based on formal verification, we conclude that the code implements the documented functionality to the extent of the code reviewed.

Technical Findings

General Observations

The code reviewed consisted of various Resolvers, a Registry, a ClaimManager and a Staking Pool contract. A total of 5 contracts and the same project dependencies were audited.

There was good communication with the development team, which quickly addressed noted issues.

Staking Pool - Owners can reinitialise pools

Finding ID: KS-ENERGYWEB-01
Severity: **Medium**
Status: **Remediated**

Description

According to documentation: Staking pool can be in one of these states during its lifetime: Deployed --> Initialised --> Open --> Expired --> Swept.

However, the initialisation function does not check whether the pool is already initialised. This can cause patrons to lose future (or current) rewards if, for example, the `end` time or `hourlyRatio` is shortened. Previously paid rewards are also discarded.

Proof of issue

File name: staking-pool/contracts/StakingPool.sol
Line number: 85

```
  function init(
        uint256 _start,
        uint256 _end,
        uint256 _hourlyRatio,
        uint256 _hardCap,
        uint256 _contributionLimit,
        bytes32[] memory _patronRoles
    ) external payable onlyOwner {
        require(
            _start >= block.timestamp,
            "Start date should be at least current block timestamp"
        );
        // check if stake pool time is at least 1 day
        require(_end - _start >= 1 days, "Duration should be at least 1
day");

        uint256 maxFutureRewards = compound(
            _hourlyRatio,
            _hardCap,
            _start,
            _end
        ) - _hardCap;

        require(msg.value >= maxFutureRewards, "Rewards lower than
expected");

        start = _start;
        end = _end;
        hourlyRatio = _hourlyRatio;
        hardCap = _hardCap;
        contributionLimit = _contributionLimit;
        patronRoles = _patronRoles;

        remainingRewards = msg.value;

        emit StakingPoolInitialized(msg.value, block.timestamp);
    }
```

Overwriting `remainingRewards` essentially causes the loss of rewards already paid to the contract. Although unlikely, this could be exploited by owners. Stealing ETH is prevented because most `remainingRewards` can be unstaked (due to arithmetics safety) or swept.

Severity and Impact summary

The pool's owners can reinitialise it, affecting its lifetime, patron's interests, and other owners' previously paid ETH.

Recommendation

Require `!initialized()` as a modifier or manually in the initialization function. Existing interfaces also exist. See OpenZeppelin's Initialisable implementation

References

- OpenZeppelin's Initialisable implementation

ClaimsManager.sol - Replay attack on registration

Finding ID: KS-ENERGYWEB-02
Severity: **Medium**
Status: **Remediated**

Description

To register a new record in ClaimsManager, the subject's and issuer's signatures on a combination of input data is checked. No nonce is included in the data. This means that the registration could be replayed, resetting an altered record to previous states.

Examples of attacks could be: - After an expiration date update, an attacker could replay the initial registration, denying the update. - If the expiration of a record is for some reason reduced, the subject could replay the initial registration to set it back to its original value.

Proof of issue

File name: contracts/roles/ClaimManager.sol
Line number: 98

```
    bytes32 agreementHash = ECDSA.toEthSignedMessageHash(
      _hashTypedDataV4(keccak256(abi.encode(
      AGREEMENT_TYPEHASH,
      subject,
      role,
      version
    ))));

    bytes32 proofHash = ECDSA.toEthSignedMessageHash(
      _hashTypedDataV4(keccak256(abi.encode(
      PROOF_TYPEHASH,
      subject,
      role,
      version,
      expiry,
      issuer
    ))));
```

Severity and Impact summary

Registration replay could allow attackers or subjects to set update records to previous states.

Recommendation

Add an increasing nonce to the `struct Record`, whose value should be included in the signed data, so that each registration is unique.

Staking Pool - Recorded staking time does not follow rewards' hourly subdivision

Finding ID: KS-ENERGYWEB-03
Severity: **Low**
Status: **Remediated**

Description

Reward calculation is performed according to the hours passed before a patron's last (un)stake occurred. The timestamp is then updated to evaluate future rewards correctly. The last timestamp will be recorded if multiple (un)stakes occur within the same hour. So staking can continue indefinitely without any interest being accumulated.

The issue is marked as Low and unlikely to happen in practice.

Proof of issue

Filename: staking-pool.sol

```
// Line 227
    function updateStake(uint256 deposit, uint256 compounded) private {
        stakes[msg.sender].deposit = deposit;
        stakes[msg.sender].compounded = compounded;
        stakes[msg.sender].time = block.timestamp;
    }
// Line 253
    function compound(
        uint256 ratio,
        uint256 principal,
        uint256 compoundStart,
        uint256 compoundEnd
    ) public view returns (uint256) {
        uint256 n = (compoundEnd - compoundStart) / 1 hours;
```

`stakes[msg.sender].time` is always updated to the last timestamp. Frequent stakes (e.g. within less than one hour) will

Severity and Impact summary

Frequent (un)staking can result in unexpected (fewer) rewards.

Recommendation

Update `stakes[msg.sender].time` only if `stakes[msg.sender].time - block.timestamp >= 1 hour`.

ClaimsManager - Any issuer can change the expiry date

Finding ID: KS-ENERGYWEB-04
Severity: **Low**
Status: **Not an Issue**

Description

To register a new record in ClaimsManager, the subject's and issuer's signatures on a combination of input data is checked. However, the subject's data is missing the expiry date, which would allow any issuer to select it at their leisure.

Proof of issue

File name: contracts/roles/ClaimManager.sol
Line number: 98

```
    bytes32 agreementHash = ECDSA.toEthSignedMessageHash(
      _hashTypedDataV4(keccak256(abi.encode(
      AGREEMENT_TYPEHASH,
      subject,
      role,
      version
    ))));

    bytes32 proofHash = ECDSA.toEthSignedMessageHash(
      _hashTypedDataV4(keccak256(abi.encode(
      PROOF_TYPEHASH,
      subject,
      role,
      version,
      expiry,
      issuer
    ))));
```

Severity and Impact summary

Any issuer can change the expiry date at their leisure without subjects approving the change.

Recommendation

Include expiry in the subject's signed data. If the issuer is known to the subject, it could also be included in the hash, requiring only one call to `ECDSA.toEthSignedMessageHash`, thus saving gas.

ClaimManager - Address Cast to Abstract Contracts

Finding ID: KS-ENERGYWEB-05

Severity: **Informational**

Status: **Open**

Description

During verification, resolver addresses are obtained from the registry and cast to the appropriate contracts. Those contracts inherit from `ResolverBase`, whose authorisations methods are abstract. The resolvers' authorisation methods should ensure authenticity and correct verification. Currently, verification depends on trusting the registry's record.

Proof of issue

File name: contracts/roles/ClaimManager.sol

```
// Line 143
function verifyPreconditions(address subject, bytes32 role) internal view {
    address resolver = ENSRegistry(ensRegistry).resolver(role);
    (bytes32[] memory requiredRoles, bool mustHaveAll) =
EnrolmentPrerequisiteRolesResolver(resolver).prerequisiteRoles(role);

// Line 163
  function  verifyIssuer(address issuer, bytes32 role) internal view {
    address resolver = ENSRegistry(ensRegistry).resolver(role);
    (address[] memory dids, bytes32 issuer_role) =
IssuersResolver(resolver).issuers(role);
```

`EnrolmentPrerequisiteRolesResolver` and `IssuersResolver` are abstract classes whose `authorised` modifier is not implemented.

Recommendation

Use resolver contracts whose authorisation methods are implemented in place.

ClaimManager - TTL of ENSRegistry is not checked

Finding ID: KS-ENERGYWEB-06
Severity: **Informational**
Status: **Open**

Description

ENS records, stored in (ENSRegsitry)[https://github.com/ensdomains/ens-contracts/blob/master/contracts/registry/ENSRegistry.sol], contain a Time to Live field, which is not checked during verification. This issue is purely informational since it might be implemented as intended.

Proof of issue

File name: iam-contracts.sol

```
  function verifyPreconditions(address subject, bytes32 role) internal view
{
    address resolver = ENSRegistry(ensRegistry).resolver(role); // HERE no
ttl

    (bytes32[] memory requiredRoles, bool mustHaveAll) =
EnrolmentPrerequisiteRolesResolver(resolver).prerequisiteRoles(role); //
HERE cast to abstract class
    if (requiredRoles.length == 0) {
      return;
    }
    uint numberOfRequiredRoles = mustHaveAll ? requiredRoles.length : 1;
    uint numberOfRoles = 0;
    for (uint i = 0; i < requiredRoles.length && numberOfRoles <
numberOfRequiredRoles; i++) {
      if (this.hasRole(subject, requiredRoles[i], 0)) {
        numberOfRoles++;
      }
    }
    require(
      numberOfRoles == numberOfRequiredRoles,
      "ClaimManager: Enrollment prerequisites are not met"
    );
  }

  function  (address issuer, bytes32 role) internal view {
    address resolver = ENSRegistry(ensRegistry).resolver(role);
    (address[] memory dids, bytes32 issuer_role) =
IssuersResolver(resolver).issuers(role);
    if (dids.length > 0) {
      EthereumDIDRegistry registry = EthereumDIDRegistry(didRegistry);
      for (uint i = 0; i < dids.length; i++) {
        // here owner doesn't matter?
        if (dids[i] == issuer || registry.validDelegate(dids[i],
ASSERTION_DELEGATE_TYPE, issuer)) {
          return;
        }
      }
      revert("ClaimManager: Issuer is not listed in role issuers list");
    } else if (issuer_role != "") {
      require(hasRole(issuer, issuer_role, 0), "ClaimManager: Issuer does
not has required role");
    } else {
      revert("ClaimManager: Role issuers are not specified");
```

```
        }
    }
```

Recommendation

If set, use the ENSRegistry's record TTL in verification.

References

- ENSRegsitry Contract