code4rena

Namespace

Smart Contract Security Assessment

Version 2.0

Audit dates: Oct 10 — Oct 15, 2024

Audited by: Peakbolt
Spicymeatball

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Namespace

ENS Service Provider focused on growing ENS through Partnerships, Integrations, and making Subname minting implementation easy.

2.2 Scope

Repository	github.com/thenamespace/namespace-contracts-v2
Commit Hash	https://github.com/thenamespace/namespace-contracts-v2/commit/ad6bb55ed037a16c5cd37fbd5e731f319bf93f11
Mitigation Hash	https://github.com/thenamespace/namespace-contracts-v2/commit/6e8a9ed397f8444633d667d06cf794373f6f4dd6

2.3 Audit Timeline

DATE	EVENT
Oct 10, 2024	Kick-off call
Oct 10, 2024	Audit start
Oct 15, 2024	Audit end
Oct 17, 2024	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	1
Low Risk	4
Informational	0
Total Issues	7

3. Findings Summary

ID	DESCRIPTION	STATUS
H-1	Incorrect token expiry extension may lead to subdomain loss	Resolved
H-2	Missing verification of `resolverData` during `mint()`	Resolved
M-1	No mechanisms to transfer existing name token to new owner	Acknowledged
L-1	`balanceOf()` should not count expired subnames	Resolved
L-2	Users can transfer tokens and set approval for expired subnames	Resolved
L-3	Registry token owners should not be able to extend expiry for their subname	Resolved
L-4	Registry token owners are able to burn their token	Resolved

4. Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Incorrect token expiry extension may lead to subdomain loss

Severity: High	Status: Resolved

Context:

- EnsNameRegistry.sol#L200

Description: When a user extends the duration of their subdomain token the `_setExpiry` function is invoked:

```
    function _setExpiry(bytes32 node, uint256 expiry) internal {
>>      expiries[node] = block.timestamp + expiry;
        emitter.emitExpirySet(node, expiries[node]);
    }
```

The issues stem from how `expiry` is added to the current timestamp, which is incorrect when extending an existing `node`. For example, if a user mints a token at time `t1` with an expiry date `t2 = t1 + 365 days`, and at `t3 = t1 + 180 days` they decide to buy an additional 100 days, their expiry should extend to `t1 + 465 days`. Currently, `_setExpiry` logic, would calculate it as `t1 + 180 + 100 = t1 + 280 days`, which shortens the duration instead of extending it. In the worst case, a user could lose their subdomain token if the additional `expiry` period is too short.

Recommendation:

```
+ if (_isExpired(node) || expiries[node] == 0) {
    expiries[node] = block.timestamp + expiry
+} else {
+    expiries[node] = expiries[node] + expiry;
```

Namespace: Fixed with PR-19

Zenith: Verified.

[H-2] Missing verification of `resolverData` during `mint()`

Severity: High	Status: Resolved

Context:

- RegistryMinter.sol#L41

Description:

When minting subnames, the `mint()` caller can provide both `MintContext`and `resolverData` to mint the subname and then set the resolver data.

The protocol's offchain backend will sign the mint context, to ensure that the mint configuration is verified and correct.

However, the `resolverData` is not included in the signature unlike `MintContext`, which means that the `mint()` caller to provide any arbitrary value as it will not be verified. This can be exploited as the `mint()` caller can be anyone who has obtained the `mint()` signature.

An attack scenario could occur as follows:

1. Alice submit mint tx to chain.
2. Attacker observes the mempool and obtain the `mint()` signature for Alice's tx.
3. Attacker then frontrun Alice's mint tx, using the obtained signature and `MintContext`, but then set the `resolverData` to his favor. e.g. setting the subname resolver address to himself.
4. Alice's subname will be minted to her. But unknown to her, the subname resolves to the attacker's address. Any fund transfer to Alice's subname will then be sent to attacker's address.

```
    function _mint(
        MintContext memory context,
        bytes[] memory resolverData,
        bytes calldata extraData
    ) internal {
        address registryAddress = getRegistryResolver().nodeRegistries(
            context.parentNode
        );

        if (registryAddress == address(0)) {
            revert RegistryNotFound(context.parentNode);
        }

        bytes32 node;
```

```
        if (resolverData.length > 0) {
>>>         node = _mintWithData(context, registryAddress, resolverData);
        } else {
            node = _mintSimple(context, registryAddress);
        }
```

Recommendation:

This could be resolve by ensuring `resolverData` is in the signature so that it can be verified.

Alternatively, a simpler solution is to check that the `msg.sender` is a legitimate caller for the `mint()`.

Namespace: Fixed with PR-18

Zenith:

Fixed by adding a `verifiedMinter` that will be verified during `mint()` to prevent unauthorized update of the resolver data.

Additional note: The commit also fixed a known issue with the signature by adding expiration time, to prevent calling the functions after a long time using outdated parameters. Also, nonces are removed from the signatures as there are existing checks to prevent replay in `mint()` and `deploy()` or replay does not matter for `extendExpiry()`.

Verified.

4.2 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] No mechanisms to transfer existing name token to new owner

Severity: Medium	Status: Acknowledged

Context:

- RegistryFactory.sol#L27-L67
- EnsNameRegistry.sol#L204-L208

Description: ENS names are temporary and can expire. The Namespace deploys a registry and mints a name token based on the L1 proof of ENS name ownership:

```
    function _deploy(FactoryContext memory context) internal {
        bytes32 tdlHash = EnsUtils.namehash(bytes32(0), context.TLD);
        bytes32 nameNode = EnsUtils.namehash(tdlHash, context.label);
        INodeRegistryResolver registryResolver = getRegistryResolver();

>>      if (registryResolver.nodeRegistries(nameNode) != address(0)) {
            revert RegistryAlreadyExists(nameNode);
        }

        RegistryConfig memory config = RegistryConfig(
            context.parentControl,
            context.expirableType,
            context.tokenName,
            context.tokenSymbol,
            getRegistryURI(),
            context.owner,
            nameNode,
            address(getEmitter())
        );

>>      EnsNameRegistry registry = new EnsNameRegistry(config);
```

For example, if Alice deploys `acme.eth` on L2, she will still have full control of her Namespace token even after her ENS ownership expires:

```
    function _mintNameToken(bytes32 node, address owner) internal {
        uint256 tokenId = uint256(node);
```

```
        _mint(owner, tokenId);
>>      expiries[node] = type(uint256).max;
    }
```

However, if Bob becomes the new owner of `acme.eth` on L1, he won't be able to obtain the L2 token. The `deploy` function would revert, as the `acme.eth` node already exists, and minting isn't an option because `acme.eth` has an infinite expiry.

Recommendation: One possible solution would be to modify `_deploy` so that it burns the previous owner's (Alice's) name token and mints it for Bob if the registry already exists.

Namespace: We're aware of that and we'll communicate that on our front end. There is technically no trustless way of allowing a new L1 token owner to claim L2 token (without us (Namespace) acting as a mediator, but in that case, we would always have a technical possibility of transferring someone else's L2 token, which is something we don't want)

Zenith: Acknowledged

4.3 Low Risk

A total of 4 low risk findings were identified.

[L-1] `balanceOf()` should not count expired subnames

Severity: Low	Status: Resolved

Context:

- EnsNameRegistry.sol#L134-L137

Description:

For the subname tokens, `ownerOf()` only returns owner address if the subname has not expired.

However, EIP721 also specifies a `balanceOf(address _owner)` that returns the number of NFTs owned by `_owner`. Refer to EIP721 specs.

So in this case there is an inconsistency, as `balanceOf()` will include tokens of subnames that has expired. This is because OZ's ERC721 track the balance upon mint/burn/transfer only. Refer to OZ ERC721.

```
    /// @notice Count all NFTs assigned to an owner
    /// @dev NFTs assigned to the zero address are considered invalid,
and this
    ///  function throws for queries about the zero address.
    /// @param _owner An address for whom to query the balance
    /// @return The number of NFTs owned by `_owner`, possibly zero
    function balanceOf(address _owner) external view returns (uint256);
```

Recommendation: The straight forward solution is to iterate through all the owner's token and check the expiry to count the balance.

Namespace: Fixed in the following commit

Acknowledged that the issue of `balanceOf()` causing OOG error (as its unbounded) is a minor issue, based on the successful testing of 1500 tokens, which is sufficient for the protocol.

Zenith: Resolved according to the recommendation by returning the count of unexpired tokens for expirable sub-names. Verified.

[L-2] Users can transfer tokens and set approval for expired subnames

Severity: Low	Status: Resolved

Context:

- EnsNameRegistry.sol#L143-L158

Description: When the subname has expired, the owner is unable to change any configuration such as the resolver settings. However, the owner is still able to transfer the token to another address and also set the approval even when the subname has expired. This is an inconsistency and may cause confusion to the users and third-party integrations.

```
    function transferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public override {
        super.transferFrom(from, to, tokenId);

        if (from != address(0)) {
            emitter.emitNodeTransfer(
                bytes32(tokenId),
                registryNameNode(),
                from,
                to
            );
        }
    }
```

Recommendation: This can be resolved by disabling `transferFrom()` and `approve()` when the subname has expired.

Also, consider disabling both `tokenURI(tokenId)` and `getApproved(_tokenId)` too. Based on EIP721, these should revert when `_tokenId` is not a valid NFT, which is applicable to expired subnames, as they are no longer a valid NFT since they no longer have a current owner.

Namespace: Fixed with PR-20 & PR-24

Zenith: Verified.

[L-3] Registry token owners should not be able to extend expiry for their subname

Severity: Low	Status: Resolved

Context:

- EnsNameRegistry.sol#L96-L103

Description:

Upon deployment of the registry, the registry's owner token expiry is set to `typeof(uint256).max` to indicate that it is not expirable.

However, it is still possible for the token owner to change the expiry by calling `extendExpiry()` as there are no checks to prevent that in `setExpiry()`.

```
    function setExpiry(bytes32 node, uint256 expiry) external
onlyController {
        uint256 tokenId = uint256(node);
        if (ownerOf(tokenId) == address(0)) {
            revert NodeNotFound(node);
        }

        _setExpiry(node, expiry);
    }
```

Recommendation:

Add a check in `setExpiry()` to ensure that `node != registryNameNode()`.

Namespace: Fixed with PR-22

Zenith: Verified.

[L-4] Registry token owners are able to burn their token

Severity: Low	Status: Resolved

Context:

- EnsNameRegistry.sol#L110-L119

Description:

The owner of each registry are tracked by the registry's token owner.

However, there are no check in `burn()` that prevents registry's token owner from burning its own token. This will prevent functions from obtaining the current owner of the registry.

```
    function burn(bytes32 node) public registryTokenOwner {
>>>     if (_isControllable()) {
            _burn(uint256(node));
            delete expiries[node];

            emitter.emitNodeBurned(node, registryNameNode(),
_msgSender());
        } else {
            revert NodeNotControllable();
        }
    }
```

Recommendation: Add a check to prevent registry's token owners from burning their token, by checking that `node != registryNameNode()`.

Namespace: Fixed with PR-22

Zenith: Verified.