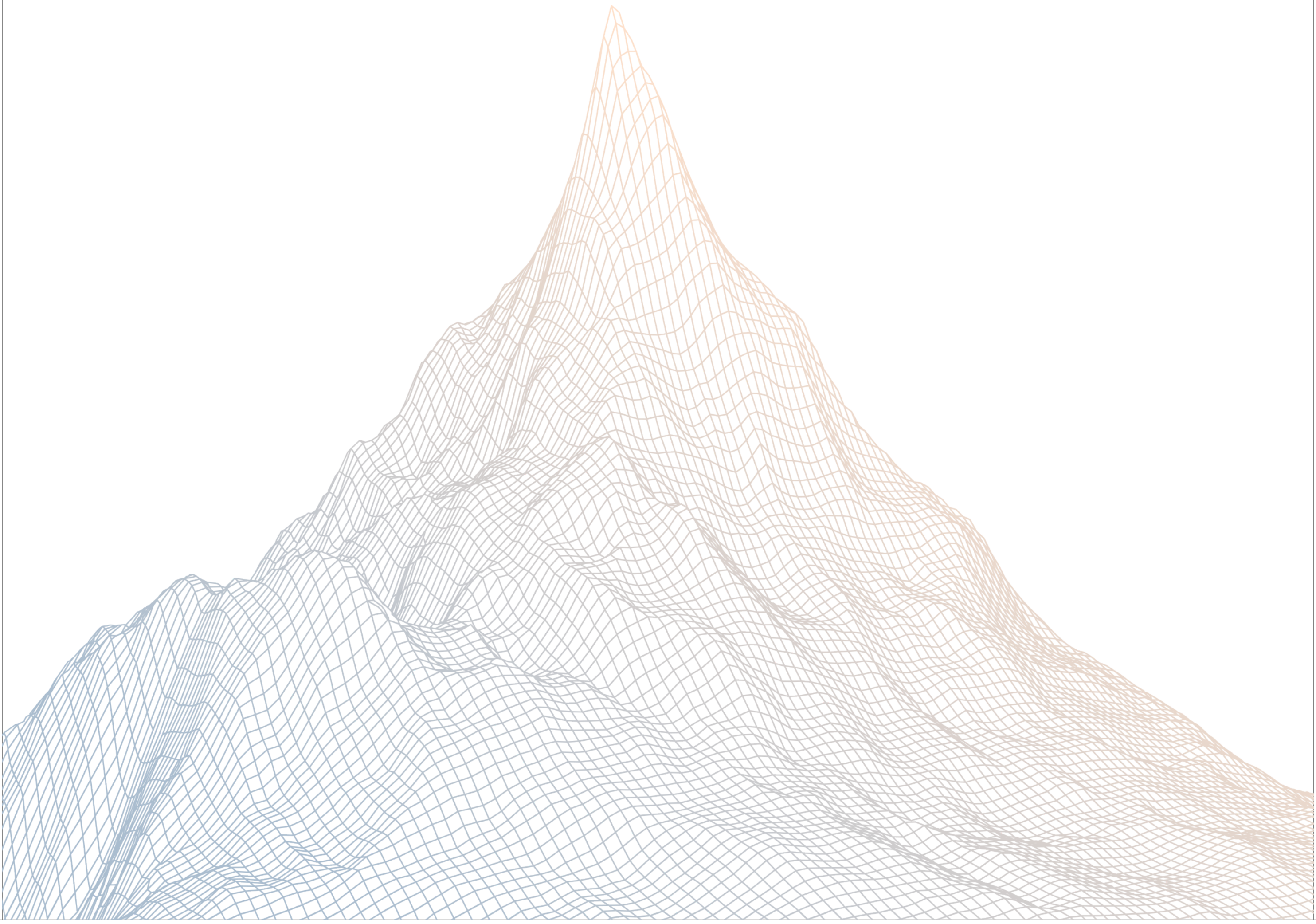
Zenith

Spectral

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES:	February 17th to February 26th, 2025
AUDITED BY:	Matte ether_sky sorryNotsorry

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 Spectral

At Spectral, we're pioneering the Onchain Agent Economy—a revolutionary paradigm where anyone can build agentic companies composed of multiple autonomous AI agents.

Imagine a decentralized future where intelligent agents not only navigate the crypto landscape 24/7 but also collaborate towards a common goal, handling workflows like hiring, firing, performance management, deposits, and rewards distribution and providing real world utility to Web3 users.

Our mission is to advance and simplify onchain AI, breaking down technical barriers so that whether you're a normie or a degen, risk off or risk on, you can tap into the full power of onchain AI agents.

2.2 Scope

The engagement involved a review of the following targets:

Target	Spectral-ANS
Repository	https://github.com/Spectral-Finance/Spectral-ANS
Commit Hash	`935b348585d4373b0872f32bd5a2051b752d5207`

Files	`ANSRegistrar.sol`
	`ANSRegistry.sol`
	`ANSResolver.sol`
	`ANSReverseRegistrar.sol`

2.3 Audit Timeline

February 17, 2025	Audit start
February 26, 2025	Audit end
March 03, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	8
Low Risk	2
Informational	5
Total Issues	**16**

3 Findings Summary

ID	Description	Status
H-1	Domain renewals can be gamed to pay less	Resolved
M-1	Malicious users can potentially receive funds belonging to others after the company's domain is released	Acknowledged
M-2	When users call the registerANSAndApplyToJobOnBehalfOf function in the BatchANSAndJobApplicationDelegate, any excess funds will be lost	Resolved
M-3	The price calculation is incorrect in the renewDomain function	Resolved
M-4	The registerANSAndApplyToJobOnBehalfOf function of the BatchANSAndJobApplicationDelegate does not work properly	Resolved
M-5	The registerSubdomain function of the ANSRegistrar is incorrect	Resolved
M-6	Users cannot withdraw excess funds in ANSRegistrar	Resolved
M-7	Whitespace Characters Can Be Used to Bypass Domain Length Requirements and Mimic Registering Premium Domains at Lower Prices	Resolved
M-8	Domain Name Spoofing Using Right-to-Left Override Characters	Resolved
L-1	There is an incorrect parameter in the SubdomainRegistered event	Resolved
L-2	Last-Minute Grace Period Renewals Might Lead to Immediate Domain Expiration	Acknowledged
I-1	StringUtils.toAsciiString() Omits Standard 0x Prefix When Converting Addresses	Acknowledged
I-2	Redundant Grace Period Check in Domain Registration	Resolved
I-3	Redundant Expiration Check in releaseDomain Function	Resolved

ID	Description	Status
I-4	Inefficient Excess Payment Handling Mechanism	Acknowledged
I-5	NATSPEC / Implementation Clash - Pricing Tiers	Resolved

4 Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Domain renewals can be gamed to pay less

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

4.2 Target

- ANSRegistrar.sol

Description:

The `gracePeriod` mechanism can be exploited to register domains at half the intended cost. The exploit works as follows:

1. Register domain for minimum duration (30 days) using `registerTopLevelDomain`
2. Let domain expire and grace period pass (at day 60)
3. Re-register the same domain for 30 days after 1 second of expiry using `registerTopLevelDomain` again - tricky part but possible considering not all the domains are popular to be sniped.
4. Repeat this cycle

Path:

- t=0: Register for 30 days using `registerTopLevelDomain`
- t=60: Domain and grace period expired, NFT burned
- t=60+1s: Register again for 30 days using `registerTopLevelDomain`
- Repeat 6 times

This allows users to save almost 50% of the intended yearly registration cost

Recommendations:

Implement a progressive pricing system that makes short-term registrations more expensive:

```
File: ANSRegistrar.sol

    function calculatePrice(string memory label, uint256 duration) public pure returns (uint256 requiredPrice) {
        uint256 length = bytes(label).length;

        if (length >= 5) {
            requiredPrice = PRICE_FIVE_PLUS;
        } else if (length == 4) {
            requiredPrice = PRICE_FOUR;
        } else if (length == 3) {
            requiredPrice = PRICE_THREE;
        } else {
            revert("Label must be at least 3 characters long");
        }
        return requiredPrice * duration;
    }
```

So for above, there can be a progressive price increase, similar to below;

```
    if (duration < 90 days) {
        // Short-term premium: 50% more expensive
        requiredPrice = (requiredPrice * 150) / 100;
   } else if (duration < 180 days) {
         // Medium-term premium: 25% more expensive
        requiredPrice= (requiredPrice * 125) / 100;
  }
  // Long-term registrations (≥180 days) use requiredPrice
    return requiredPrice * duration;
```

This makes the short-term registration strategy more expensive, encouraging longer-term registrations.

Spectral: Resolved with @8e82b510d6...

Zenith: Verified.

4.3 Medium Risk

A total of 8 medium risk findings were identified.

[M-1] Malicious users can potentially receive funds belonging to others after the company's domain is released

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ANSRegistrar.sol

Description:

Imagine a `company` registers the `top-level domain` `hcompany` and several `subdomains` like `user1.hcompany`, `user2.hcompany`, etc. ANSRegistrar.sol#L228

```
function registerSubdomain(
    string calldata parentLabel,
    string calldata subLabel,
    address associatedAddress, // Address to associate with the subdomain
    address _resolver,
    uint64 ttl
) external {
    // Update the registry with the subdomain record (msg.sender is the
    subdomain owner)
    ansRegistry.setRecord(subNode, msg.sender, _resolver, ttl);

    // Set the parent-child relationship in the registry
    ansRegistry.setParent(subNode, parentNode);
}
```

Users can apply to the `company`'s job using `ANS labels` like `user1.hcompany` and `user2.hcompany` .

However, there's no guarantee that the `company` will retain this `domain` indefinitely. After the `grace period`, any malicious user can re-register this `top-level domain` and become the `owner`.

Since this `domain` is still the `parent domain` for its `subdomains` like `user1.hcompany`, the malicious user gains control over these `subdomains` as well. So, they could change the `owner` of these `subdomains` to themselves. ANSRegistry.sol#L144

```
function setRecord(
    bytes32 node,
    address newOwner,
    address newResolver,
    uint64 newTtl
@→ ) external onlyDomainManagers(node) {

    _records[node] = Record({
        owner: newOwner,
        resolver: resolverToSet,
        ttl: newTtl
    });
    emit RecordChanged(node, newOwner, resolverToSet, newTtl);
}
```

And consequently update the addresses for these `subdomains` as they wish. ANSResolver.sol#L57-L62

```
function setAddr(bytes32 node, address domainAddr) external override {
    (address owner, , ) = ansRegistry.getRecord(node);
    require(owner == msg.sender, "Caller is not the domain owner");
    _addresses[node] = domainAddr;
    emit AddressSet(node, domainAddr); // Emit event
}
```

When `hiring bonuses` or `trading bonuses` are distributed, all these funds would be sent to the new malicious receivers. OctoDistributor.sol#L331

```
function transferHiringDistributions(
    HiringDistribution[] calldata distributions,
    address agentToken,
    uint256 totalSpec,
    uint256 totalAgentToken,
    uint256 totalUsdc
) external nonReentrant onlyAgenticCompany() {
    for (uint256 i = 0; i < distributions.length; i++) {
        UserBalances storage user = userBalances[
@->             systemAd-
    dresses.ansResolver.addr(distributions[i].recipientAnsNode)
```

```
          ];
          user.spectral += distributions[i].specAmount;
          accSpecAmount += distributions[i].specAmount;
          _addOrGetAgentToken(user, agentToken);
          user.agent_tokens_list[agentToken]
      += distributions[i].agentTokenAmount;
          accAgentTokenAmount += distributions[i].agentTokenAmount;
          user.usdc += distributions[i].usdcAmount;
          accUsdcAmount += distributions[i].usdcAmount;
      }
  }
```

Recommendations:

- Track all `subdomains` for a `parent domain` in the `registerSubdomain` function and reset the `parent domain` for all `subdomains` when renewing the `parent domain`. To achieve this, a function to reset the `parent domain` should also be added to the `ANSRegistry`.
- A simpler solution is to not allow the same domain to be used twice in the `registerTopLevelDomain` function.

Spectral: We acknowledge this issue, its the responsibility of the top level domain owner to not let their domain expire and be overtaken by somebody new

Zenith: Acknowledged

[M-2] When users call the registerANSAndApplyToJobOnBehalfOf function in the BatchANSAndJobApplicationDelegate, any excess funds will be lost

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ANSRegistrar.sol

Description:

When users call the `registerANSAndApplyToJobOnBehalfOf` function in the `batch contract`, it internally calls the `registerTopLevelDomain` function to register the `top level` on behalf of the `caller`. BatchANSAndJobApplicationDelegate.sol#L77

```
function registerANSAndApplyToJobOnBehalfOf(
    string calldata _ansLabel,
    uint256 _ansDuration,
    address _company,
    bytes32 _jobId
)
    external
    payable
{
    BatchDelegateStorage storage $ = _getStorage();

    // Register the ANS name
    bytes32 ansNode = $.ansRegistrar.registerTopLevelDomain{value:
    msg.value}(
        _ansLabel,
        _ansDuration,
        address($.ansResolver)
    );
}
```

The issue is that from the `ANSRegistrar`'s perspective, `msg.sender` is the `batch contract`. ANSRegistrar.sol#L187-L191

```
function registerTopLevelDomain(
    string calldata label,
    uint256 duration,
    address _resolver
) external payable nonReentrant returns (bytes32 labelHash) {
    // Handle excess payment using the withdrawal pattern
    uint256 excess = msg.value - requiredPrice;
    if (excess > 0) {
        excessPayments[msg.sender] += excess;
        totalExcessPayments += excess;
        emit ExcessPaymentStored(msg.sender, excess);
    }

    return labelHash;
}
```

As a result, any excess funds are saved to the `batch contract`'s balance, meaning the `original callers` can't receive these excess funds. Additionally, there's no mechanism in place for the `batch contract` to `withdraw` these excess funds.

Recommendations:

Add a `refund address` to the `registerTopLevelDomain` function.

```
function registerTopLevelDomain(
    string calldata label,
    uint256 duration,
    address _resolver,
+ address _refundTo
) external payable nonReentrant returns (bytes32 labelHash) {
    // Handle excess payment using the withdrawal pattern
    uint256 excess = msg.value - requiredPrice;
    if (excess > 0) {
-        excessPayments[msg.sender] += excess;
+^^I^^I excessPayments[_refundTo] += excess;
        totalExcessPayments += excess;
-        emit ExcessPaymentStored(msg.sender, excess);
+        emit ExcessPaymentStored(_refundTo, excess);
    }

    return labelHash;
}
```

Then, in the `registerANSAndApplyToJobOnBehalfOf` function, set the `caller` as the `refund` address.

```
function registerANSAndApplyToJobOnBehalfOf(
    string calldata _ansLabel,
    uint256 _ansDuration,
    address _company,
    bytes32 _jobId
)
    external
    payable
{
    BatchDelegateStorage storage $ = _getStorage();

    // Register the ANS name
    bytes32 ansNode = $.ansRegistrar.registerTopLevelDomain{value:
    msg.value}(
        _ansLabel,
        _ansDuration,
        address($.ansResolver),
+^^I^^I msg.sender
    );
}
```

Spectral: Resolved with @5ed4165db1...

Zenith: Verified.

[M-3] The price calculation is incorrect in the renewDomain function

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ANSRegistrar.sol

Description:

The `price` is calculated based on the length of the `label`. In the `registerTopLevelDomain` function, `.ethAgent` is not involved.
(ANSRegistrar.sol#L166)[https://github.com/Spectral-Finance/Spectral-ANS/blob/935b348585d4373b0872f32bd5a2051b752d5207/src/contracts/ANSRegistrar.sol#L16

```
function registerTopLevelDomain(
    string calldata label,
    uint256 duration,
    address _resolver
) external payable nonReentrant returns (bytes32 labelHash) {
    uint256 requiredPrice = calculatePrice(label, duration);
    require(msg.value >= requiredPrice, "Insufficient payment for domain");

    uint256 expiration = block.timestamp + duration;

    domainData[labelHash] = DomainInfo({
        name: string(abi.encodePacked(label, ".ethAgent")),
        expiration: expiration
    });
}
```

However, in the `renewDomain` function, `.ethAgent` is included in the `price` calculation.

```
function renewDomain(bytes32 labelHash, uint256 duration)
    external payable nonReentrant {
    uint256 requiredPrice = calculatePrice(info.name, duration);
    require(msg.value >= requiredPrice, "Insufficient payment for renewal");
}
```

Recommendations:

One possible solution would be:

```
- function calculatePrice(string memory label, uint256 duration)
    public pure returns (uint256 requiredPrice) {
+ function calculatePrice(string memory label, uint256 duration,
    bool includesSuffix) public pure returns (uint256 requiredPrice) {

-     uint256 length = bytes(label).length;
+ ^^I uint256 length = includesSuffix ? bytes(label).length - 9 :
    bytes(label).length;

    if (length >= 5) {
        requiredPrice = PRICE_FIVE_PLUS;
    } else if (length == 4) {
        requiredPrice = PRICE_FOUR;
    } else if (length == 3) {
        requiredPrice = PRICE_THREE;
    } else {
        revert("Label must be at least 3 characters long");
    }
    return requiredPrice * duration;
}

function renewDomain(bytes32 labelHash, uint256 duration)
    external payable nonReentrant {
-     uint256 requiredPrice = calculatePrice(info.name, duration);
+^^I uint256 requiredPrice = calculatePrice(info.name, duration, true);
    require(msg.value >= requiredPrice, "Insufficient payment for renewal");
}
```

Spectral: Resolved with @8e82b510d6c...

Zenith: Verified.

[M-4] The registerANSAndApplyToJobOnBehalfOf function of the BatchANSAndJobApplicationDelegate does not work properly

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BatchANSAndJobApplicationDelegate.sol

Description:

Users can register their `ANS name` and apply for a job using the `registerANSAndApplyToJobOnBehalfOf` function.
BatchANSAndJobApplicationDelegate.sol#L77

```
function registerANSAndApplyToJobOnBehalfOf(
    string calldata _ansLabel,
    uint256 _ansDuration,
    address _company,
    bytes32 _jobId
)
    external
    payable
{
    BatchDelegateStorage storage $ = _getStorage();

    // Register the ANS name
    bytes32 ansNode = $.ansRegistrar.registerTopLevelDomain{value:
    msg.value}(
        _ansLabel,
        _ansDuration,
        address($.ansResolver)
    );
}
```

It is typical for users to have an `ANS name` like 'username.companyname' as this directly indicates who is applying and to which company. However, this is currently impossible.

In the `registerTopLevelDomain` function, the `parentHash` is calculated at `line 158`, resulting in `keccak(keccak('companyname'))`. However, the `BatchANSAndJobApplicationDelegate` is not the `owner` of that node, causing the transaction to revert at `line 162`.

```
function registerTopLevelDomain(
    string calldata label,
    uint256 duration,
    address _resolver
) external payable nonReentrant returns (bytes32 labelHash) {

158:    bytes32 parentHash = getParentDomainHash(label);

    if (parentHash ≠ bytes32(0)) {
        address parentOwner = _ownerOf(uint256(parentHash));
        require(parentOwner ≠ address(0), "Parent domain does not exist");
162:        require(parentOwner == msg.sender, "Sender does not own the
    parent domain");
    }
}
```

Recommendations:

We could bypass the check at `line 162` when the caller is `BatchANSAndJobApplicationDelegate`.

Spectral: Resolved with @d3b1c768ee...

Zenith: Verified.

[M-5] The registerSubdomain function of the ANSRegistrar is incorrect

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- ANSRegistrar.sol

Description:

Suppose a `company` registers its `top-level domain` name, `kCompany`, using the `registerTopLevelDomain` function. The `labelHash` is `keccak256(keccak256('kCompany'))` and it is recorded in `ansRegistry`. ANSRegistrar.sol#L138

```
function registerTopLevelDomain(
    string calldata label, // 'kCompany'
    uint256 duration,
    address _resolver
) external payable nonReentrant returns (bytes32 labelHash) {
    labelHash = label.compute();  // `keccak256(keccak256('kCompany'))`

    ansRegistry.setRecord(labelHash, msg.sender, _resolver, 0); // Resolver
    set to default
    ansRegistry.setDomainName(labelHash, domainData[labelHash].name); // Set
    domain name
}
```

The `company` restricts users who want to apply for jobs at the `company` to have an `ANS name` like `user1.kCompany`. To facilitate this, the `company` registers `subdomains` such as `user1.kCompany`, `user2.kCompany`, etc. For `user1.kCompany`, the `parentLabel` will be `kCompany` and the `subLabel` will be `user1`. ANSRegistrar.sol#L218

```
function registerSubdomain(
    string calldata parentLabel,
    string calldata subLabel,
    address associatedAddress, // Address to associate with the subdomain
    address _resolver,
    uint64 ttl
) external {
212:    bytes32 parentNode = parentLabel.compute();

    require(ansRegistry.isAuthorized(parentNode, msg.sender), "Not
    authorized to manage this domain");

218:    bytes32 subNode = keccak256(abi.encodePacked(parentNode,
    keccak256(bytes(subLabel))));

    ansRegistry.setRecord(subNode, msg.sender, _resolver, ttl);
}
```

At `line 212`, the `parentNode` will be `keccak256(keccak256('kCompany'))`. At `line 218`, the `subNode` will be `keccak256(keccak256(keccak256('kCompany')), keccak256('user1'))`, which is recorded in `ansRegistry`.

When a user with this `ANS label` tries to apply for a job, the `_checkAnsNode` function is called. AgenticCompany.sol#L218

```
function applyToJob(bytes32 jobId)
    external returns (bytes32 newJobApplicationId_) {
    bytes32 callerAnsNode = _checkAnsNode(_msgSender());
}
```

This function calls the `getName` function of the `ANSReverseRegistrar`. AgenticCompany.sol#L661

```
function _checkAnsNode(address caller)
    private view returns (bytes32 ansNode_) {
    string memory ansName
    = IANSReverseRegistrar(ANS_REVERSE_REGISTRAR).getName(caller);
    if (bytes(ansName).length == 0) {
        revert AddressMustResolveToAnsName(caller);
    }
    ansNode_ = ansName.compute();
}
```

To return `user1.kCompany` as a name, it should be set using the `setName` function. ANSReverseRegistrar.sol#L102

```
function getName(address addr) external view returns (string memory name) {
    string memory reverseName
    = string(abi.encodePacked(addr.toAsciiString(), ".addr.ethAgent"));
    bytes32 reverseNamehash = reverseName.compute();
    name = ansRegistry.getReverseName(reverseNamehash);
}
```

The `nameHash` is calculated using the `Namehash` library. ANSReverseRegistrar.sol#L76

```
function setName(string calldata name) external {
    bytes32 namehash = name.compute();

    (address owner, , ) = ansRegistry.getRecord(namehash);
    require(owner == msg.sender, "Not authorized to manage this domain");

    string memory reverseName
    = string(abi.encodePacked(msg.sender.toAsciiString(),
    ".addr.ethAgent"));
    bytes32 reverseNamehash = reverseName.compute();

    ansRegistry.setReverseRecord(reverseNamehash, msg.sender, address(0), 0,
    name);
    emit ReverseRecordChanged(msg.sender, name);
}
```

In the library, the `compute` function returns `keccak256(keccak256(keccak256('user1')), keccak256('kCompany'))` for `user1.kCompany`. Namehash.sol#L23

```
function compute(string memory name) public pure returns (bytes32) {
    bytes32 node;
    bytes memory labels = bytes(name);
    uint256 labelStart = 0;

    for (uint256 i = 0; i <= labels.length; i++) {
        if (i == labels.length || labels[i] == ".") {
            if (i > labelStart) {
                node = keccak256(abi.encodePacked(node,
    keccak256(substring(labels, labelStart, i))));
            }
            labelStart = i + 1;
        }
    }
    return node;
}
```

This differs from the registered node `keccak256(keccak256(keccak256('kCompany')), keccak256('user1'))`.

This issue arises because the `registerSubdomain` function calculates the `label hash` incorrectly; the hash order is reversed compared to the `compute` function.

Recommendations:

```
function registerSubdomain(
    string calldata parentLabel,
    string calldata subLabel,
    address associatedAddress, // Address to associate with the subdomain
    address _resolver,
    uint64 ttl
) external {
    bytes32 parentNode = parentLabel.compute();

    require(ansRegistry.isAuthorized(parentNode, msg.sender), "Not
    authorized to manage this domain");

-    bytes32 subNode = keccak256(abi.encodePacked(parentNode,
    keccak256(bytes(subLabel))));
+    bytes32 subNode = string(abi.encodePacked(subLabel, ".",
    parentLabel)).compute();

}
```

Spectral: Resolved with @70c17d83b08...

Zenith: Verified

[M-6] Users cannot withdraw excess funds in ANSRegistrar

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ANSRegistrar.sol

Description:

When users register a `top-level domain` or renew a `domain`, they need to pay, and any excess funds are stored in `excessPayments` when they send more money than the `price`. ANSRegistrar.sol#L262

```
function renewDomain(bytes32 labelHash, uint256 duration)
    external payable nonReentrant {
    // Handle excess payment using the withdrawal pattern
    uint256 excess = msg.value - requiredPrice;
    if (excess > 0) {
        excessPayments[msg.sender] += excess;
        emit ExcessPaymentStored(msg.sender, excess);
    }
}
```

Users can `withdraw` excess funds at any time. ANSRegistrar.sol#L276

```
function withdrawExcess() external nonReentrant {
    uint256 amount = excessPayments[msg.sender];
    require(amount > 0, "No excess payment to withdraw");

    excessPayments[msg.sender] = 0;

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Withdrawal failed");

    emit ExcessPaymentWithdrawn(msg.sender, amount);
}
```

However, when the `owner` calls the `withdrawFunds` function, the entire `balance` is withdrawn regardless of whether some users have unclaimed excess funds. ANSRegistrar.sol#L297

```
function withdrawFunds() external onlyOwner {
    uint256 balance = address(this).balance;
    require(balance > 0, "No funds to withdraw");

    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Withdrawal failed");
}
```

As a result, users are unable to claim their excess funds.

Recommendations:

```
+ uint256 totalExcessPayments;

function withdrawFunds() external onlyOwner {
-^^I uint256 balance = address(this).balance;
+    uint256 balance = address(this).balance - totalExcessPayments;
    require(balance > 0, "No funds to withdraw");

    (bool success, ) = msg.sender.call{value: balance}("");
    require(success, "Withdrawal failed");
}

function registerTopLevelDomain(
    string calldata label,
    uint256 duration,
    address _resolver
) external payable nonReentrant returns (bytes32 labelHash) {
    uint256 excess = msg.value - requiredPrice;
    if (excess > 0) {
        excessPayments[msg.sender] += excess;
+        totalExcessPayments += excess;
        emit ExcessPaymentStored(msg.sender, excess);
    }
}

function renewDomain(bytes32 labelHash, uint256 duration)
    external payable nonReentrant {
    uint256 excess = msg.value - requiredPrice;
    if (excess > 0) {
        excessPayments[msg.sender] += excess;
+        totalExcessPayments += excess;
        emit ExcessPaymentStored(msg.sender, excess);
    }
}

function withdrawExcess() external nonReentrant {
    uint256 amount = excessPayments[msg.sender];
    require(amount > 0, "No excess payment to withdraw");

    excessPayments[msg.sender] = 0;

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Withdrawal failed");

+    totalExcessPayments -= amount;
    emit ExcessPaymentWithdrawn(msg.sender, amount);
}
```

Spectral: Resolved with @8e82b510d6c...

Zenith: Verified.

[M-7] Whitespace Characters Can Be Used to Bypass Domain Length Requirements and Mimic Registering Premium Domains at Lower Prices

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ANSRegistrar.sol

Description:

The `registerTopLevelDomain` function in ANSRegistrar.sol has a character validation issue where users can bypass the minimum length requirement (3 characters) by using whitespace characters. The contract calculates the price and validates the length based on the raw string length, including whitespace characters.

For example:

```
"  x" // 3 characters (2 spaces + 'x')
```

This allows users to:

1. Register visually single-character domains while bypassing the 3-character minimum requirement
2. Pay a lower price tier for what appears to be a single-character domain (the worst condition)
3. Create misleading domain names that could be used for phishing or impersonation

The test demonstrates that while "x" is rejected, " x" is accepted and stored as " x.ethAgent", effectively mimicing registration of invalid single-character domains at a much lower price point.

Please insert below in `ANSRegistrar.t.sol` file and run `forge test --mt testShortNamePriceExploit -vvv`

```
function testShortNamePriceExploit() public {
    vm.startPrank(deployer);
    vm.deal(deployer, 10 ether);

    // Test case: "  x" (2 spaces + x)
    string memory shortLabel = "x";
    string memory exploitLabel = "  x"; // 2 spaces before x

    console.log("\nTesting short name exploit with whitespace");
    console.log("Short label:", shortLabel);
    console.log("Exploit label:", exploitLabel);

    // Calculate prices
    vm.expectRevert("Label must be at least 3 characters long");
    uint256 shortPrice = registrar.calculatePrice(shortLabel, 30 days);

    uint256 exploitPrice = registrar.calculatePrice(exploitLabel, 30 days);
    console.log("Exploit price:", exploitPrice);

    // Try to register with exploit label
    bytes32 labelHash = registrar.registerTopLevelDomain{value:
    exploitPrice}(
        exploitLabel,
        30 days,
        address(resolver)
    );

    // Get the stored domain info
    (string memory storedName, uint256 expiration)
    = registrar.domainData(labelHash);

    console.log("\nRegistration results:");
    console.log("Stored domain name:", storedName);
    console.log("Expected name:", string(abi.encodePacked(shortLabel,
    ".ethAgent")));
    console.log("Duration:", expiration - block.timestamp);

    vm.stopPrank();
}
```

Console output:

```
Ran 1 test for test/ANSRegistrar.t.sol:ANSRegistrarTest
[PASS] testShortNamePriceExploit() (gas: 243911)
Logs:
  AnsRegistry implementation deployed at:
    0x522B3294E6d06aA25Ad0f1B8891242E335D3B459
  ANSRegistry deployed at:  0x535B3D7A252fa034Ed71F0C53ec0C6F784cB64E1
  ANSResolver implementation deployed at:
    0xc051134F56d56160E8c8ed9bB3c439c78AB27cCc
  ANSResolver deployed at:  0x2c1DE3b4Dbb4aDebEbB5dcECAe825bE2a9fc6eb6
  ANSReverseRegistrar implementation deployed at:
    0x83769BeEB7e5405ef0B7dc3C66C43E3a51A6d27f
  ANSReverseRegistrar deployed at:
    0x00EFd0D4639191C49908A7BddbB9A11A994A8527
  ANSRegistrar implementation deployed at:
    0x147B09A8C7d5E4A8253a3e01De4356D3c132010D
  ANSRegistrar deployed at:  0x062C88B4ba954955746eDA6f475C26eeaC04614B

Testing short name exploit with whitespace
  Short label:  x
  Exploit label:    x
  Exploit price:  259200000000000

Registration results:
  Stored domain name:    x.ethAgent
  Expected name:  x.ethAgent
  Duration:  2592000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.95ms (2.60ms
    CPU time)

Ran 1 test suite in 28.99ms (9.95ms CPU time): 1 tests passed, 0 failed, 0
    skipped (1 total tests)
```

Recommendations:

Add input validation to strip or reject leading/trailing whitespace.

Spectral: Resolved with @d3b1c768ee...

Zenith: Verified.

[M-8] Domain Name Spoofing Using Right-to-Left Override Characters

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- ANSRegistrar.sol

Description

The `registerTopLevelDomain` function accepts domain labels without validating or sanitizing Unicode control characters. This allows malicious users to register domains containing Right-to-Left Override (U+202E) characters, which can be used to spoof domain names and conduct phishing attacks.

It's because:

1. The contract uses `Namehash.compute()` which processes raw strings without character validation
2. No input sanitization is performed on the `label` parameter
3. The domain name is stored and displayed as provided

Example path:

```
// Register a domain that appears as "bank" but is actually "knab"
// Using RTL override: U+202E
registerTopLevelDomain("bank[RTLO]knab", duration, resolver); // Github
    doesn't show the character once input here as an example so [RTLO] is
    [U+202E]

// Visually appears as: "bank.ethAgent"
// Actually stored as: "knabknab.ethAgent"
```

This can be exploited to:

1. Create phishing domains that appear legitimate
2. Impersonate established organizations/domains

3. Trick users in both web interfaces and blockchain explorers

Recommendations:

Add input validation to reject RTL and other Unicode control characters.

Spectral: Resolved with @8e82b510d6c...

Zenith: Verified

4.4 Low Risk

A total of 2 low risk findings were identified.

[L-1] There is an incorrect parameter in the SubdomainRegistered event

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ANSRegistrar.sol

Description:

The fourth parameter of the `SubdomainRegistered` event is the `owner address`. ANSRegistrar.sol#L71

```
event SubdomainRegistered(
    bytes32 indexed subLabelHash,
    string subDomainName,
    bytes32 indexed parentLabelHash,
    address owner
);
```

This event is emitted in the `registerSubdomain` function. In this function, the input parameter `associatedAddress` is only used for the event. ANSRegistrar.sol#L231-L236

```
function registerSubdomain(
    string calldata parentLabel,
    string calldata subLabel,
    address associatedAddress, // Address to associate with the subdomain
    address _resolver,
    uint64 ttl
) external {
    // Update the registry with the subdomain record (msg.sender is the
    subdomain owner)
```

```
        ansRegistry.setRecord(subNode, msg.sender, _resolver, ttl);

        // Set the parent-child relationship in the registry
        ansRegistry.setParent(subNode, parentNode);

        // Emit an event
        emit SubdomainRegistered(
            subNode,
            string(abi.encodePacked(sanitizedSubLabel, ".",
        sanitizedParentLabel, ".ethAgent")),
            parentNode,
            associatedAddress
        );
    }
```

I thought that `associatedAddress` should be the `owner` of the `subdomain`. However, the comments indicate that `msg.sender` should be the `owner` of the `subdomain`. If this is true, the `fourth parameter` of the event should be `msg.sender`, making the `associatedAddress` parameter redundant.

I have set the severity to Low. However, if `associatedAddress` is indeed the `owner` of the `subdomain` according to the event, then the severity should be Medium.

Recommendations:

```
function registerSubdomain(
    string calldata parentLabel,
    string calldata subLabel,
-    address associatedAddress, // Address to associate with the subdomain
    address _resolver,
    uint64 ttl
) external {
    // Emit an event
    emit SubdomainRegistered(
        subNode,
        string(abi.encodePacked(sanitizedSubLabel, ".",
    sanitizedParentLabel, ".ethAgent")),
        parentNode,
-        associatedAddress
+        msg.sender
    );
}
```

Spectral: Resolved with @ec8f09cb4fd...

Zenith: Verified.

[L-2] Last-Minute Grace Period Renewals Might Lead to Immediate Domain Expiration

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ANSRegistrar.sol

Description:

The `renewDomain` function extends the expiration from the previous expiration date rather than the current time. When users renew during the last moments of the grace period, this can lead to immediate or near-immediate expiration of the domain.

For example:

1. Domain expires at t=30 days
2. Grace period ends at t=60 days
3. User renews at t=59.99 days for 30 days
4. New expiration = 30 + 30 = 60 days
5. Result: Domain expires almost immediately after renewal possibly someone calling `releaseDomain`

```
function renewDomain(bytes32 labelHash, uint256 duration)
    external payable nonReentrant {
    // ... other checks ...

    DomainInfo storage info = domainData[labelHash];
    require(block.timestamp <= info.expiration + gracePeriod, "Domain cannot
    be renewed after grace period");

>>  info.expiration += duration; // Adds to previous expiration, not current
    time

    emit DomainRenewed(labelHash, info.expiration);
    // ...
}
```

This can lead to users unintentionally losing their domains if they don't understand that the renewal extends from the original expiration date. They might pay for 30 days but get only a few seconds or minutes of registration.

Recommendations:

This is not in the hands of the protocol, but a warning can be added accordingly.

Spectral: Acknowledged

4.5 Informational

A total of 5 informational findings were identified.

[I-1] StringUtils.toAsciiString() Omits Standard 0x Prefix When Converting Addresses

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Acknowledged	`LIKELIHOOD`: Low

Target

- StringUtils.sol

Description:

The `StringUtils.toAsciiString()` function, which is used for reverse resolution in the ANS system, converts Ethereum addresses to strings without the conventional "0x" prefix. While this doesn't affect the core ANS functionality since all operations use namehash for lookups rather than string comparisons, it deviates from the standard Ethereum address string representation (which includes "0x"). This could potentially cause confusion or integration issues for third-party systems that expect address strings to follow the standard format.

```
File: ANSReverseRegistrar.sol

        // Compute the reverse node namehash (e.g., "0xABC...123.addr.ethAgent")
        string memory reverseName = string(abi.encodePacked(addr.toAsciiString(), ".addr.ethAgent"));
        bytes32 reverseNamehash = reverseName.compute();
```

Please insert below test to `ANSRegistrar.t.sol` file and run with `forge test --mt testAddressToStringConversion -vvv`:

```
function testAddressToStringConversion() public {
    // Test with a known address
    address testAddr = address(0x1234567890123456789012345678901234567890);
    // Get string representation using StringUtils
    string memory addrString = StringUtils.toAsciiString(testAddr);

    console.log("Address as string:", addrString);

    assertNotEq(
        addrString,
        "0x1234567890123456789012345678901234567890",
        "Address string conversion failed"
    );
}
```

Output:

```
Ran 1 test for test/ANSRegistrar.t.sol:ANSRegistrarTest
[PASS] testAddressToStringConversion() (gas: 40282)
Logs:
  Address as string: 1234567890123456789012345678901234567890

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.95ms
    (602.47µs CPU time)
```

Recommendations:

Modify the `toAsciiString()` function to include the "0x" prefix:

```
function toAsciiString(address x) internal pure returns (string memory) {
    bytes memory s = new bytes(42); // Allocate 42 bytes for "0x" + address
    s[0] = "0";
    s[1] = "x";
    for (uint i = 0; i < 20; i++) {
        bytes1 b = bytes1(uint8(uint(uint160(x)) / (2**(8 * (19 - i)))));
        bytes1 hi = bytes1(uint8(b) / 16);
        bytes1 lo = bytes1(uint8(b) - 16 * uint8(hi));
        s[2 + 2 * i] = char(hi);
        s[2 + 2 * i + 1] = char(lo);
    }
    return string(s);
}
```

Spectral: Acknowledged

[I-2] Redundant Grace Period Check in Domain Registration

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ANSRegistrar.sol

Description:

In `ANSRegistrar.sol`, there is redundant and misleading code that suggests domain renewal functionality during the grace period, even though this was not intended to be a feature:

```
// First check prevents any registration if domain exists and hasn't expired
if(existingDomain.expiration > block.timestamp) {
    revert("Domain already registered and not expired");
}

// Redundant and misleading check suggesting renewal functionality
if (previousOwner ≠ address(0) && block.timestamp
    <= existingDomain.expiration + gracePeriod) {
    require(msg.sender == previousOwner, "Only previous owner can renew
    before grace period ends");
}

// Later in the function
_safeMint(msg.sender, uint256(labelHash));  // Would fail anyway for
    existing tokens
```

Even if the grace period is reached, the subsequent `_safeMint` would fail for existing tokens.

Recommendations:

Remove the redundant grace period check and its associated comment since domain renewal is not an intended feature.

Spectral: Resolved with @8e82b510d6...

Zenith: Verified.

[I-3] Redundant Expiration Check in releaseDomain Function

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- ANSRegistrar.sol

Description:

The `releaseDomain` function contains a redundant check for domain expiration. The function has two checks:

1. `require(info.expiration < block.timestamp, "Domain has not expired yet")`
2. `require(block.timestamp > info.expiration + gracePeriod, "Within grace period")`

The first check is redundant because the second check already ensures that the domain has expired. If `block.timestamp > info.expiration + gracePeriod` is true, then `info.expiration < block.timestamp` is always true.

```
File: ANSRegistrar.sol

    function releaseDomain(string calldata label) external nonReentrant {
        bytes32 labelHash = label.compute();
        DomainInfo storage info = domainData[labelHash];
>>      require(info.expiration < block.timestamp, "Domain has not expired yet"); // Redundant
        require(block.timestamp > info.expiration + gracePeriod, "Within grace period");

        // Burn the token to release the domain
        _burn(uint256(labelHash));
    }
```

Recommendations:

Remove the redundant check and keep only the grace period verification:

```
function releaseDomain(string calldata label) external nonReentrant {
    bytes32 labelHash = label.compute();
    DomainInfo storage info = domainData[labelHash];
   require(info.expiration < block.timestamp, "Domain has not expired
       yet");
    require(block.timestamp > info.expiration + gracePeriod, "Within
grace period");

    // Burn the token to release the domain
    _burn(uint256(labelHash));
}
```

Spectral: Resolved with @8e82b510d6c...

Zenith: Verified.

[I-4] Inefficient Excess Payment Handling Mechanism

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ANSRegistrar.sol

Description:

The contract uses `excessPayments` mapping to store if the sent funds are above the computed prices to pay. If it is, the users are expected to withdraw those funds by calling `withdrawExcess` after registering their domains.

This mechanism is an overhead for the users and the stored amount value might be less than the gas price used to withdraw them too.

Recommendations:

Strict comparison can resolve this:

```diff
- require(msg.value >= requiredPrice, "Insufficient payment for domain");
+ require(msg.value == requiredPrice, "Insufficient payment for domain");
```

Spectral: Acknowledged

[I-5] NATSPEC / Implementation Clash - Pricing Tiers

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ANSRegistrar.sol

Description:

Fixed pricing tiers are used in price calculation with the comment of `costs per block`

```
File: ANSRegistrar.sol

// Fixed Pricing Tiers based on label length, these are costs per block!
    uint256 public constant PRICE_FIVE_PLUS = 0.000000000001 ether; // 5+ characters
    uint256 public constant PRICE_FOUR = 0.00000000001 ether;       // 4 characters
    uint256 public constant PRICE_THREE = 0.0000000001 ether;       // 3 characters
```

However, the BASE block time is 2 seconds, and the prices are not used accordingly OR the comment needs to be updated.

```
File: ANSRegistrar.sol

    function calculatePrice(string memory label, uint256 duration) public pure returns (uint256 requiredPrice) {
        uint256 length = bytes(label).length;

        if (length >= 5) {
            requiredPrice = PRICE_FIVE_PLUS;
        } else if (length == 4) {
            requiredPrice = PRICE_FOUR;
        } else if (length == 3) {
            requiredPrice = PRICE_THREE;
        } else {
            revert("Label must be at least 3 characters long");
            }
>>      return requiredPrice * duration; //@audit duration is in seconds
        }
```

Recommendations:

The comment or the implementation should be updated to reflect each other

Spectral: Resolved with @6ba595adf1...

Zenith: Verified.