PUBLIC

Security Audit

of Dot Crypto Smart Contracts

November 21, 2019

Produced for

UNSTOPPABLE DOMAINS

by

CHAINSECURITY

Table Of Contents

Foreword

We would like to thank Unstoppable Domains for choosing ChainSecurity to audit their smart contracts. This document outlines our methodology, limitations and results.

– ChainSecurity

Executive Summary

Unstoppable Domains engaged ChainSecurity to perform a security audit of Dot Crypto, an Ethereum-based smart contract system. The Dot Crypto smart contracts of Unstoppable Domains implements an on-chain naming system similar to ENS. Each Domain and subdomain is represented by an ERC721 token. Minting of SLDs can only be performed by Unstoppable Domains, it is centralized.

ChainSecurity audited the smart contracts which are going to be deployed on the public Ethereum chain. Audits of ChainSecurity use state-of-the-art tools for detection of generic vulnerabilities and checks of custom functional requirements. Additionally, a thorough manual code review by leading experts helps to ensure the highest security standards. During the audit, ChainSecurity was able to help Unstoppable Domains in addressing several security, trust and design issues of high, medium and low severity.

The employed coding practices and partial documentation increased the complexity of the audit. However, the great communication with Unstoppable Domains during the audit has lead to swift answering of any questions from ChainSecurity.

All of the reported issues have been fixed/acknowledged by Unstoppable Domains. ChainSecurity has no further concerns regarding this project.

Audit Overview

Methodology

CHAINSECURITY's methodology in performing the security audit consisted of four chronologically executed phases:

1. Understanding the existing documentation, purpose and specifications of the smart contracts.
2. Executing automated tools to scan for generic security vulnerabilities.
3. Manual analysis covering both functional (best effort based on the provided documentation) and security aspects of the smart contracts by one of our CHAINSECURITY experts.
4. Preparing the report with the individual vulnerability findings and potential exploits.

Limitations

Security auditing cannot uncover all existing vulnerabilities: even a contract in which no vulnerabilities are found during the audit is not a guarantee of a secure smart contract. However, auditing enables the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire smart contract application, while others lack protection only in certain areas. This is why we carry out a source code review aimed at determining all issues that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed a security audit in order to discover as many vulnerabilities as possible.

Scope

Source code files received	October 18, 2019
Git commit	`c466643b82836a5e0dacee344e871706965654c3`
EVM version	PETERSBURG
Initial Compiler	SOLC compiler, version 0.5.11
Final code update received	November 13, 2019
Final commit	`685035e1bae9231e71320e0baaa8009276c51c84`
Final Compiler	SOLC compiler, version 0.5.12

The scope of the audit is limited to the following source code files.

In Scope	File	SHA-256 checksum
✓	registry/Children.sol	5823fe9f426503a5078ce0c3b534549eaa9c3c38cde5ca73306a2ffdcc1c4568
✓	registry/IChildren.sol	3c945d0217aa37ad957f52f4e680b2c906ecd4148ba0eff578e9e2dc2cfa5d8d
✓	registry/Metadata.sol	043ad1527d5c9e6ed8d8303a0bac42975308cc4ca805bccced1c7d2dc70c9fa9
✓	registry/Resolution.sol	9ffe75b69a4bded1e412f0b970fdb843f4f747491a2274aada5ce5387fe3eeaf
✓	registry/Root.sol	6d4ab99f8c535681023bca721a5f1044770cb873f4bd853be7e8aecb70b3fa3d
✓	IRegistry.sol	08798140d0de868170b8ae2d92067552213b94ed1dce7a83db9f7967cce7c201
✓	Registry.sol	ffa1c4f39e663acd43ecb9939a39a214fd49c67737ecbeddade05c1bb483c2ac
✓	controllers/ISunriseController.sol	83a9b0c6e8a7c5b856add4a0912645da5d55bf712af08c3cc850e25186b1f631
✓	controllers/SunriseController.sol	7fc1f810a5220cafdd84566c06d4bdd5679c669bb32ccf8620fe6df13d5ae948
✓	controllers/IMintingController.sol	9268ae61106e541dfa99e97bafec4b181d21db1d78053fa4ae6bfa41a70e93ea
✓	controllers/ISignatureController.sol	229093bdd2b855a36682d1bcc6ad67ca9a311198b52f872483ee6848d9f3a488
✓	controllers/IURIPrefixController.sol	b0df7d5fe763de07af1c410011fcf32bd9120abc3b09d410f245262b25380079
✓	controllers/MintingController.sol	633732d9e95f375264a453d339a4d6aa3404e42c776ef375dca6034d08bf0a3c
✓	controllers/SignatureController.sol	5d629726dfee9c37755e717f21e76df3a3f4237411b6cde5e96f465e5859ca9d
✓	controllers/URIPrefixController.sol	74f370aa6fe7d210a58236f7c35455fbc35ce4bacdc7e2cffc3d33d89ffa7c42
✓	resolver/Resolver.sol	ce27300387ba9aa82167849a4c210304da606cffa5fd45a43c40b184b9d4f515
✓	resolver/SignatureResolver.sol	a236f3887de24f8fcc72bf1ba0cc08b5dfbac4e9ab3dc3627a2f8c5be7d0ca2a
✓	util/BulkWhitelistedRole.sol	9eb456fae5f36a18f23e82a3ee7001ef29aaecdeb017a7297a00f2f3791f4070
✓	util/ControllerRole.sol	ab1a0ed6376aa43e03e14bca6a0d8003fae8173da2efcb561b1d86bd49a6361c
✓	util/SignatureUtil.sol	0c3b65dbb7c4d5d6f580fb57a5c285853d159094f27f7597487c29c3eea14785
✓	util/Multiplexer.sol	4bf790fbdcd4eea7795892fc9d8c8adba511eb63afd7b423c418bef9b47fb206
✓	util/WhitelistedMinter.sol	1ed575b9cf2436328f7e649e84b683748855306b069801b37fc350679caee797

For these files the following categories of issues were considered:

In Scope	Issue Category	Description
✓	Security Issues	Code vulnerabilities exploitable by malicious transactions
✓	Trust Issues	Potential issues due to actors with excessive rights to critical functions
✓	Design Issues	Implementation and design choices that do not conform to best practices

Depth

The security audit conducted by ChainSecurity was restricted to:

- Scanning the contracts listed above for generic security issues using automated systems and manually inspecting the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, ChainSecurity has adopted the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business-related consequences of an exploit.

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Severity is derived from the likelihood and the impact calculated previously.

We categorise the findings, depending on their severities, into four distinct groups:

- L Low: can be considered less important
- M Medium: should be fixed
- H High: we strongly recommend fixing it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the table below, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit, concerns might arise or tools might flag certain security issues. After carefully inspecting the potential security impact, we assign the following labels:

- ✓ No Issue no security impact
- ✓ Fixed the issue is addressed technically, for example by changing the source code
- ✓ Addressed the issue is mitigated non-technically, for example by improving the user documentation and specification
- ✓ Acknowledged the issue is acknowledged and it is decided to be ignored, for example due to conflicting requirements or other trade-offs in the system

Findings that are labeled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is listed simply to give the reader a quick overview of what kind of issues were found during the audit.

System Overview

The DOT CRYPTO smart contract system implements an on-chain name system. It is similar to ZNS (Zilliqa Name Service), and inspired by ENS (Ethereum Name Service). The TLD of the DOT CRYPTO name system is `crypto`. Domains that are minted will therefore be of the form: `myname.crypto`. Each minted domain is represented by an ERC721 NFT, provided by the OpenZeppelin `ERC721` contract.

Minting of the second level domains (SLD) is centralized and can only happen through the `MintingController`. However, once a domain is minted to an account, that account can perform a number of actions on it. Besides being able to perform actions on the domain itself, a domain owner also has control over it's immediate child domain(s). To prevent the owner of the root level domain (`crypto`) to control its children (SLDs), the root level domain is immediately burned by transferring ownership to address `0xdead`.

Besides actions that can be performed on a (child) domain, each domain has a number of attributes:

- `tokenURI`, always starting with the set prefix (default is `unr:udc:`), and set when minted.
- An optional resolver address, can be updated at any time.
- Optionally one or multiple records inside the Resolver. A record is a key/value pair for a specific domain(=`tokenId`).
- An optional approved account which can perform actions on the token owner's behalf.

System Roles

In this section we outline the different roles and their permissions and purpose within the system.

Registry Controller A Controller can grant other accounts the Controller role and renounce its own Controller role. It can not take away the Controller role of another account. The deployer of `Registry` is granted the Controller role at deployment. This deployer will then add the `MintingController`, `SignatureController` and `URIPrefixController` as Controllers. If on mainnet, this will be followed by a call where the deployer renounces his Controller role. There are several functions which can only be called by a Controller. The three controllers will each call a specific set of `onlyController` functions.

MintingController Minter The Minter role contract is identical to the above Controller role in what it can do. The deployer of `MintingController` is granted the Minter role at deployment, This deployer will then add the `WhitelistedMinter` as Minter. If on mainnet, this will be followed by a call where the deployer renounces his Minter role. Any whitelisted account in the `WhitelistedMinter` can call the mint functions of the `MintingController`.

WhitelistedMinter WhitelistAdmin The deployer of `WhitelistedMinter` will be granted the WhitelistAdmin role. A WhitelistAdmin can grant other accounts the WhitelistAdmin role, renounce its own WhitelistAdmin role, and add/remove accounts to the whitelist. It can not take away the WhitelistAdmin role of another account. It's only purpose is to add/remove accounts to the whitelist (see below role).

WhitelistedMinter Whitelisted The WhitelistAdmin is the only Role which can add/remove accounts to the whitelist. A Whitelisted account can at any time renounce their Whitelisted role. Furthermore, a Whitelisted account can call three functions of the `MintingController`.

Domain Owner Each (sub)domain is represented by an ERC721 NFT, which has an owner. Therefore, the NFT owner is the Domain Owner. A domain owner can:

- Transfer ownership to another account.
- Burn the domain.
- Set the Resolver address.
- Add/Remove a record inside the Resolver.
- Set an account as Approved for this Domain.
- Mint an immediate child domain to any account.
- Burn an immediate child domain.
- Transfer ownership of an immediate child domain to any account.

Any of the above actions (except add Approved) can also be performed by the Owner off-chain signing the function call. This signed call can then be submitted on-chain by any account through calling the appropriate function inside the `SignatureResolver`.

Domain Owner Approved A Domain Owner can set a single address as Approved for that domain. The Approved address can do any of the actions of the above Domain Owner.

Approved for All Any address can set one address as being Approved for All. This works the same as the above Domain Owner Approved, except that Approved for All is global instead of per domain. The Approved for All can perform any action on a domain that is owned by the account which set it as Approved for All.

User Can only call view functions.

Trust Model

Here, we present the trust assumptions for the roles in the system as provided by DOT CRYPTO. Auditing the enforcement of these assumptions is outside the scope of the audit. Users of DOT CRYPTO should keep in mind that they have to rely on DOT CRYPTO to correctly implement and enforce these trust assumptions.

Deployer The deployer is fully trusted to use the correct code during deployment and set the right parameters.

Registry Controller A Controller is trusted to act honestly at all times and not abuse its powers.

MintingController Minter A Minter is trusted to act honestly at all times and not abuse its powers.

WhitelistedMinter WhitelistAdmin A WhitelistAdmin is trusted to act honestly at all times and not abuse its powers.

WhitelistedMinter Whitelisted A Whitelisted account is trusted to act honestly at all times and not abuse its powers.

User, Domain Owner, Domain Owner Approved, Approved for All These roles are untrusted and assumed to be potentially malicious. They are expected to only execute attacks which are economically beneficial for them.

Best Practices in UNSTOPPABLE DOMAINS's project

CHAINSECURITY is determined to deliver the best results to ensure the security of a project. To enable us to do so, we are listing Hard Requirements which must be fulfilled to allow us to start the audit. Furthermore we are providing a list of proven best practices. Following them will make audits more meaningful by allowing efforts to be focused on subtle and project-specific issues rather than the fulfilment of general guidelines.

Hard Requirements

These requirements ensure that the UNSTOPPABLE DOMAINS's project can be audited by CHAINSECURITY.

- ✓ **All files and software for the audit have been provided to CHAINSECURITY**
  The project needs to be complete. Code must be frozen and the relevant commit or files must have been sent to CHAINSECURITY. All third party code (like libraries) and third-party software (like the solidity compiler) must be exactly specified or made available. Third party code can be located in a folder separated from client code (and the separation needs to be clear) or included as dependencies. If dependencies are used, the version(s) need to be fixed.
- ✓ The code must compile and the required compiler version must be specified. When using outdated versions with known issues, clear reasons for using these versions are being provided.
- ✓ There are migration/deployment scripts executable by CHAINSECURITY and their use is documented.
- ✓ The code is provided as a Git repository to allow reviewing of future code changes.

Best Practices

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable.

- ✓ There are no compiler warnings, or warnings are documented.
- ✓ Code duplication is minimal, or justified and documented.
- ✓ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ✓ The project only contains audit-related files, or, if this is not possible, a meaningful distinction is made between modules that have to be audited and modules that CHAINSECURITY should assume are correct and out-of-scope.
- ✓ There is no dead code.
- ✓ The code is well-documented.
- ☐ The high-level specification is thorough and enables a quick understanding of the project without any need to look at the code.
  EXPLANATION: There is no specification.
- ☐ Both the code documentation and the high-level specification are up-to-date with respect to the code version CHAINSECURITY audits.
  EXPLANATION: There is no specification.
- ✓ Functions are grouped together according to either the Solidity guidelines[2], or to their functionality.

[2] https://solidity.readthedocs.io/en/v0.4.24/style-guide.html#order-of-functions

Smart Contract Test Suite

In this section, ChainSecurity comments on the smart contract test suite of Dot Crypto. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

Although the number of provided tests seems to be ok, ChainSecurity is unsure if the test coverage is high enough (Unstoppable Domains should add an npm script to get the test coverage). Furthermore, multiple checks seem to be done inside a single test instead of splitting them into separate tests. There are several contracts in which only the happy path is tested. For contracts where errors are also checked the returned error message is not checked. This could lead to another error besides the expected error being thrown, resulting in the error check in the test to pass, even though the expected error is not the cause of the rejection. ChainSecurity found some issues which could have been discovered with more elaborate tests.

Security Issues

This section relates to our investigation into security issues. It is meant to highlight times when we found specific issues, but also mentions what vulnerability classes do not appear, if relevant.

Locked token possible when sent to `Registry` contract L ✓ Acknowledged

Every (sub)domain in the DOT CRYPTO project is represented with an ERC721 token. It is possible to transfer a DOT CRYPTO ERC721 token to the `Registry` contract itself. If this happens the token will be locked forever as there is no way to transfer tokens sent to the `Registry` contract to another address. UNSTOPPABLE DOMAINS should consider either disallowing tokens to be send to the `Registry` contract, or implementing a way to transfer such tokens from the `Registry` to another address.

Likelihood: Low
Impact: Medium

Acknowledged: UNSTOPPABLE DOMAINS explained this is not a major concern at this moment.

Trust Issues

This section reports functionality that is not enforced by the smart contract and hence correctness relies on additional trust assumptions.

Whitelisted accounts can add/renounce a Minter M ✓ Fixed

The `SunriseController` inherits from the `MintingController` contract, which inherits from `MinterRole`. An address having the MinterRole can perform the following actions:

- It can assign the MinterRole to another address.
- It can renounce its own MinterRole.
- It can call any of the functions inside `SunriseController`, including those with `onlyMinter`.

In the DOT CRYPTO project the `Multiplexer` contract is assigned the MinterRole. The `Multiplexer` contract only implements a fallback function. This fallback function uses `call()` to call any function inside the `SunriseController` based on the function selector. Only addresses added to the whitelist inside the `Multiplexer` are allowed to call this fallback function. For an address to be whitelisted the WhitelistAdmin of the `Multiplexer` needs to add it to the whitelist.

There are several problems caused by this setup. First off, since the fallback function uses `call()` with a function selector, a whitelisted address can call `SunriseController.addMinter`. This works because the caller of the `addMinter` function needs to be a Minter, which is the case since the `Multiplexer` contract will call it. This allows a whitelisted address in the `Multiplexer` to add any address as Minter inside the `SunriseController`.

Another problem is that a whitelisted address could call `SunriseController.renounceMinter` to make the `Multiplexer` contract renounce its MinterRole inside the `SunriseController`. If this happens there is no way to add back the `Multiplexer` as a Minter since only a Minter can add another Minter, and the only Minter on mainnet is the `Multiplexer` (according to the deployment script). Ofcourse, if the first problem has been used to add another Minter this doesn't apply as that Minter could add another Minter. But as explained above that is a problem in itself which should be prevented.

As explained above the usage of `call()` makes it possible to call any function inside the `SunriseController`. UNSTOPPABLE DOMAINS should prevent above issues. As can be read in another issue the `Multiplexer` could be better removed entirely.

Fixed: UNSTOPPABLE DOMAINS replaced the `Multiplexer` with a new contract `WhitelistedMinter` which does not use `call()`.

Label allowed to have "." (dot) M ✓ Acknowledged

A label in a child node is allowed to contain `"."` (dot). There is no check preventing this.

Due to this, there could be a possible attack scenario: The attacker tries to successfully register an already registered URI. In the example, we will use `donate.ethereum.crypto`. The attacker has no powers over the `ethereum.crypto` domain and `donate.ethereum.crypto` has been successfully minted and a resolver has been set.

The attacker registers `donate.ethereum`. Hence, it instructs a minter to call `mintSLD(attackerAddress, "donate.ethereum")`. Here, it might have to trick a minter, but there are generally no guarantees on the checks a minter has in place.

The minting process will generate the following childId: `keccak256(abi.encodePack(_CRYPTO_HASH, keccak256(abi.encodePack("donate.ethereum"))))`. This does not collide with the existing entry as it has the following childId: `keccak256(abi.encodePack(keccak256(abi.encodePack(_CRYPTO_HASH, keccak256(abi.encodePack("ethereum")))), keccak256(abi.encodePack("donate"))))`. Hence, the IDs will not collide.

After successful minting, the attacker can set the resolver using the `resolveTo` function. As a result of the attack, the attacker has a `tokenId`, which successfully resolves to `donate.ethereum.crypto` and where it totally controls the resolver. This can now be used for social engineering.

However, ENS (Ethereum Name Service) also allow "." (dot) character as a part of the domain name.

Acknowledged: Unstoppable Domains acknowledges the problem but explained the back-end of Unstoppable Domains can disallow purchasing SLDs that include a dot, and that an uncompromised client app will not show such domains.

Multiple Minters/WhitelistAdmins possible L ✓ Acknowledged

The Minter and WhitelistAdmin roles will in the current setup only be assigned to a single account. It is possible for a Minter or WhitelistAdmin to assign the same role to other addresses. Both the Minter and WhitelistAdmin roles are allowed to execute powerful actions. Since the roles are only assigned to a single account, having the possibility to assign the roles to multiple accounts is simply not needed. Users need to currently trust the roles to not be assigned to multiple accounts. If instead the code is updated such that there can only ever be one WhitelistAdmin and Minter, the users would not have to put trust in Unstoppable Domains, as the code enforces having only a single Minter/WhitelistAdmin.

Acknowledged: Unstoppable Domains acknowledged the issue and explained that future features might require more than one Minter/WhitelistAdmin.

Design Issues

This section lists general recommendations about the design and style of UNSTOPPABLE DOMAINS's project. These recommendations highlight possible ways for UNSTOPPABLE DOMAINS to improve the code further.

Unused _tokenResolvers mapping M ✓ Fixed

In the Metadata contract the _tokenResolvers mapping is defined as:

```
// Mapping from token ID to resolver address
mapping (uint256 => address) internal _tokenResolvers;
```

However, this mapping is not used in the Metadata contract. Inside the Resolution contract the same variable is declared, and in this contract it is used. Therefore, the unused variable _tokenResolvers should be removed from Metadata. This is another reason CHAINSECURITY recommends to get rid of the unnecessary complex inheritance hierarchy in the Registry contract.

Fixed: UNSTOPPABLE DOMAINS removed the unused mapping.

Resolution._burn is never called H ✓ Fixed

The Registry contract (in)directly inherits Resolution and Metadata. Both of these contracts define an internal function called _burn.

```
// Metadata.sol
function _burn(uint256 tokenId) internal {
  super._burn(tokenId);
  // Clear metadata (if any)
  if (bytes(_tokenURIs[tokenId]).length != 0) {
    delete _tokenURIs[tokenId];
  }
}

// Resolution.sol
function _burn(uint256 tokenId) internal {
  super._burn(tokenId);
  // Clear resolver (if any)
  if (_tokenResolvers[tokenId] != address(0x0)) {
    delete _tokenResolvers[tokenId];
  }
}
```

If we track the calling of _burn in the Registry contracts we see that:

- Children.burnChild calls Metadata._burn, which calls ERC721._burn.
- Resolution._burn calls ERC721._burn.

Looking at the above list it becomes apparent that Resolution._burn is never called. This means the tokenResolver for a burned token is not wiped from contract storage. Although this does not lead to any problems, the tokenResolver should still be wiped when a token is burned.

As was explained in another issue, the over-use of inheritance might have contributed to this bug. Also, more tests would have likely uncovered this bug. UNSTOPPABLE DOMAINS should make sure the tokenResolver inside Resolution is wiped when a domain is burned.

Fixed: UNSTOPPABLE DOMAINS merged the two _burn functions.

Unused imports L ✓ Fixed

The Root contract imports Resolution.sol and IRegistry.sol. However, these are not used inside Root. Also, inside the Multiplexer contract there is an unused import of OpenZeppelin Address.sol. Hence, UNSTOPPABLE DOMAINS should either use or remove the unused imports.

Fixed: UNSTOPPABLE DOMAINS removed the unused imports.

Missing third "safe" mint functions L ✓ Fixed

In the ERC721 contract there are three functions defined to transfer a token:

- transferFrom: Transfers token, without notification on receiver address.
- safeTransferFrom: Transfers token and calls the onERC721Received function on receiver address, if the function exists. Also takes bytes memory _data parameter to be used inside the receiver's contract.
- safeTransferFrom: Transfers token and calls the onERC721Received function on receiver address, if the function exists. However, does not have a _data argument, but will pass in an empty string to onERC721Received.

UNSTOPPABLE DOMAINS has chosen to also add the ability to do a "safe" mint, which will call onERC721Received in the recipient contract, if it is a contract. These "Safe" mint functions reside in the Children contract and are called mintChild, safeMintChild, controlledMintChild, and controlledSafeMintChild.

When comparing the above functions to the three-function pattern used inside ERC721.sol for transferring a token, it becomes clear that the "safe" mint function without a _data argument is missing in both cases. However, the transferFrom related functions inside Children do use the three-function pattern. UNSTOPPABLE DOMAINS should consider also adding the missing third function for the mint related functions.

Fixed: UNSTOPPABLE DOMAINS added the third safe mint function.

Intermediate child can be burned M ✓ Acknowledged

Using the burnChild function, an Approved or Owner of a parent token can burn its child tokens.

```
function burnChild(uint256 tokenId, string calldata label) external
    onlyApprovedOrOwner(tokenId) {
  _burn(_childId(tokenId, label));
}
```

If there is a child node which has itself child nodes, than deleting an intermediate child node does not delete all its sub-child nodes.

For example: a node like child3.child2.child1.crypto is present. However, when deleting the child2 node from the contract, the child3 node would still exist and would resolve to the child3.child2.child1.crypto address.

Acknowledged: UNSTOPPABLE DOMAINS acknowledged and explained that all child tokens can be controlled with a combination of transactions. Even if an intermediate child is burnt, it can be recovered and used to burn/manage all subdomains.

transferFromChildFor visibility could be external L ✓ Fixed

The transferFromChildFor function has array arguments and visibility set to public. This function is not called internally. Therefore it can be set to external instead. If a function with array arguments and visibility public is called externally, it will cost more gas to execute. If the function has visibility external array arguments will be directly read from calldata. If on the other hand it is public, the array arguments will first be copied to memory. This makes the function cheaper to execute when it is marked as external. UNSTOPPABLE DOMAINS should set the function visibility to external. CHAINSECURITY would also like to note that all the other functions inside the SignatureController are set to external, so this function being public likely is a mistake.

Fixed: UNSTOPPABLE DOMAINS updated the visibility of the function to `external`.

Inconsistent ordering of parameters L ✓ Fixed

In the `SignatureController` contract the `safeTransferFromFor` function ABI encodes the parameters:

```
function safeTransferFromFor(address from, address to, uint256 tokenId, bytes
    calldata signature) external {
  _validate(keccak256(abi.encodeWithSelector(msg.sig, from, to, "",
      tokenId)), tokenId, signature);
  _registry.controlledSafeTransferFrom(from, to, tokenId, "");
}
```

Unlike all the other functions inside this contract, this function does not put the empty string as last parameter in the ABI encode call. To be consistent, UNSTOPPABLE DOMAINS should also put the empty string as last parameter here.

Fixed: UNSTOPPABLE DOMAINS updated the order of parameters.

Empty `data` present in signed data, but not forwarded M ✓ Fixed

Inside the `SignatureController` the ABI encoding call always includes an empty string as argument. This makes sense when the called `Registry` function is called with the empty string. If however the called `Registry` function is not called with an empty string, it does not make sense to add it in the ABI encoding call. The following functions do not call a `Registry` function with an empty string argument:

- `transferFromFor`
- `safeTransferFromFor`
- `resolveToFor`
- `burnFor`
- `mintChildFor`
- `transferFromChildFor`
- `safeTransferFromChildFor`
- `burnChildFor`

UNSTOPPABLE DOMAINS is advised to remove the empty string from the ABI encoding call in the above functions.

Fixed: UNSTOPPABLE DOMAINS removed the empty string from the signed data of mentioned functions.

Fallback function is unnecessarily `payable` M ✓ Fixed

The fallback function inside `Multiplexer` is `payable`. However, there are no other contracts that accept ETH in the DOT CRYPTO smart contracts. Hence, UNSTOPPABLE DOMAINS should remove `payable` from the `Multiplexer` fallback function.

Fixed: UNSTOPPABLE DOMAINS removed the `Multiplexer`.

Unnecessary `Multiplexer` M ✓ Fixed

The `Multiplexer` contract has a variable `account`, set at deployment, which is the contract to forward all calls to. This is set to the `SunriseController` inside the deployment script (and cannot be changed afterwards). The only function implemented inside `Multiplexer` is a fallback function, which makes use of assembly to call any function inside the `SunriseController`. The use of assembly is generally not recommended, unless there is a real need for it.

The `Multiplexer` contract inherits `WhitelistedRole` from OpenZeppelin. This contract inherits `WhitelistAdminRole` from OpenZeppelin. The deployer of `Multiplexer` will be added as a `WhitelistAdmin` at deployment. The `WhitelistAdmin` can add/remove whitelisted accounts, renounce its whitelist admin role, and add new whitelist admins.

The `Multiplexer` fallback function can only be called by accounts that are whitelisted (by the whitelist admin). It can call any method inside the `SunriseController`. Some functions inside the `SunriseController` can only be called by `onlyMinter`. Therefore, these functions can only be called through the `Multiplexer`.

CHAINSECURITY thinks this design is overcomplicated. If the `SunriseController` (and `MintingController`) would inherit from `WhitelistedRole` and `onlyMinter` was replaced with `onlyWhitelisted`, than the same would be achieved. This would make the use of (dangerous) assembly inside the `Multiplexer` unnecessary. As a matter of fact, the entire `Multiplexer` contract would be unnecessary.

Therefore, CHAINSECURITY recommends UNSTOPPABLE DOMAINS to remove the `Multiplexer` contract.

Fixed: UNSTOPPABLE DOMAINS removed the `Multiplexer`.

Missing `tokenId` parameter in event `Sync` H ✓ Fixed

The `Sync` event emitted from `Resolution.sync` contains two parameters. The resolver contract address that called this function, and the keccak256 of the set `key`. The `tokenId` is not a parameter of this event. Therefore, this event currently is not useful as it does not tell for which `tokenId` a `key` has been set. UNSTOPPABLE DOMAINS should add the `tokenId` as a parameter to the event (and set it as `indexed`).

Fixed: UNSTOPPABLE DOMAINS added the `tokenId` parameter to the event.

Missing `sync` call in `setFor` H ✓ Fixed

The `Resolver` contract's `set` function calls `Registry.sync` to emit an event that tells which `key` got updated. However, the `Registry.sync` function is not called when `SignatureResolver.setFor` is executed. CHAINSECURITY thinks this function should also call `sync` and thereby emit a `Sync` event.

Fixed: UNSTOPPABLE DOMAINS added the missing `sync` call.

`onlyController` can be more specific M ✓ Acknowledged

The `onlyController` modifier allows only a Controller to call certain functions. Inside the deployment script the `SignatureController` and `SunriseController` are added as Controller. Furthermore, the address that deploys the `Registry` is also added as Controller. Any of the controllers can call `renounceController` to renounce its Controller role. The functions that have this modifier and the Controllers that call each function are:

- `Metadata.controlledSetTokenURIPrefix`, called by Controller EOA.
- `Resolution.controlledResolveTo`, called by `SignatureController`.
- `ControlledERC721.controlledTransferFrom`, called by `SignatureController`.
- `ControlledERC721.controlledSafeTransferFrom`, called by `SignatureController`.
- `ControlledERC721.controlledBurn`, called by `SignatureController` and `SunriseController`.
- `Children.controlledMintChild`, called by `SignatureController` and `SunriseController`.
- `Children.controlledSafeMintChild`, called by `SignatureController` and `SunriseController`.

The code could be improved by replacing the generic Controller role with two variables, one for the SignatureController and one for the SunriseController. By adding a few modifiers, onlySignatureController, onlySunriseController, and onlySignatureOrSunriseController, it would become immediately clear which Controller will call this function. This would increase the overall code readability and quality.

Acknowledged: UNSTOPPABLE DOMAINS acknowledged the problem and explained that because the contracts can't be altered after the controllers are configured, it's not critical to make a Role system more complex than it needs to be.

Confusing name `resolveSunriseSLD` M ✓ Fixed

The SunriseController contract implements a function called resolveSunriseSLD. Inside the Resolution contract "resolve" means to (un)set the resolver address of a specific tokenId. However, inside the SunriseController the resolveSunriseSLD function will unset the token sunrise mapping entry for that tokenId, and optionally also burn that tokenId. Since this is not in any way similar to the resolve inside Resolution, UNSTOPPABLE DOMAINS is advised to change the name of resolveSunriseSLD.

As explained in an open question it is unclear to CHAINSECURITY how this function is expected to function. Besides the name of the function not making sense, the intent argument plus the content of the function are confusing.

Fixed: UNSTOPPABLE DOMAINS removed the sunrise-related code from the project.

Missing error messages M ✓ Acknowledged

Some **require** statements inside the UNSTOPPABLE DOMAINS contracts do not specify an error message. This means currently if such a **require** fails it is not immediately clear where/why it failed. This also makes it possible to test for specific errors being thrown in the tests. UNSTOPPABLE DOMAINS should consider adding error messages to the **require** statements.

Acknowledged: UNSTOPPABLE DOMAINS explained that the Registry contract is too large to add error messages and that it would violate the ERC170 requirement.

Missing function that returns sunrise L ✓ Fixed

There is no function present inside the SunriseController that returns the sunrise. However, there are two **view** functions that return the time-till-sunrise and is-sunrise-over, respectively. Since the sunrise variable is defined as **private** it cannot be directly retrieved. Although having the two mentioned sunrise related **view** function gives useful information it could still be of great value to users to be able to get the value of the sunrise variable. UNSTOPPABLE DOMAINS should consider adding a function which returns the sunrise variable value.

Fixed: UNSTOPPABLE DOMAINS removed the sunrise-related code from the project.

Use Checks-Effects-Interactions pattern M ✓ Fixed

The SignatureController has two variants of the mint SLD function, safe and regular. The safe version is called safeMintSunriseSLD and will call recipient.onERC721Received after the NFT has been minted. It will only try to call this function if recipient is a contract.

It seems there is a reentrancy bug in the safeMintSunriseSLD function. The _tokenSunrises mapping is set to true only after the call to mint, and thus after the recipient.onERC721Received call. On closer inspection it becomes clear this does not cause a problem since the second call to mint the same domain will throw since it already exists. Furthermore, for this to work the recipient would need to have the Minter role, which in the current system should not be the case since whitelisted accounts are under control of UNSTOPPABLE DOMAINS and fully trusted.

There is still a chance, although small, that a Minter and/or WhitelistAdmin account is compromised. This could allow an attacker to add another whitelisted account in the Multiplexer, or add another Minter in the SunriseController. If the attacker creates a contract with an onERC721Received function which calls back into safeMintSunriseSLD, and adds this as Minter/Whitelisted, he could use this to batch add multiple SLDs in a single transaction. However, this is unlikely to motivate an attacker to invest money/time in such an attack. In any case, a trusted account's key needs to be compromised.

Still, UNSTOPPABLE DOMAINS is advised to adhere to the Checks-Effects-Interactions[3] pattern and update the `_tokenSunrises` mapping to `true` before calling `safeMintSLD` inside `safeMintSunriseSLD`.

Fixed: UNSTOPPABLE DOMAINS removed the sunrise-related code from the project.

Unnecessary `ownerOf` check in `isSunrise` L ✓ Fixed

Inside the `SunriseController` the function `isSunrise` does the following check on the first line:

```
require(_registry.ownerOf(tokenId) != address(0));
```

However, the `ownerOf` function internally already checks the owner is not address zero. Therefore, wrapping `_registry.ownerOf` inside a `require` inside `isSunrise` is unnecessary. Simply doing `_registry.ownerOf(tokenId);` would suffice.

Fixed: UNSTOPPABLE DOMAINS removed the sunrise-related code from the project.

Unnecessary complex call to `ownerOf` M ✓ Fixed

The `SunriseController` calls `_registry.ownerOf` inside `resolveSunriseSLD`. This function is defined inside the `ERC721` contract. It will throw if the owner is address zero. However, the `resolveSunriseSLD` merely wants to know if the token has an owner, it does not use the returned owner address. To prevent a revert if the `onwerOf` call throws because the token has no owner, the `ownerOf` function is called using `address(_registry).call()`.

This seems overcomplicated. If UNSTOPPABLE DOMAINS were to add a function `exists` to the `Registry` contract, which simply calls `ERC721._exists`, than there would be no need for the use of `.call()`. Instead, `_registry.exists(tokenId)` could be called the normal way. This would lower the complexity of the code inside `resolveSunriseSLD` and increase the overall quality of the project.

Fixed: UNSTOPPABLE DOMAINS removed the sunrise-related code from the project.

Over-complicated use of inheritance M ✓ Fixed

The Registry contract makes heavy use of inheritance. The inherited contracts per contract are:

- `Registry`, inherits: `IRegistry`, `Resolution`, `Children`.
- `Children`, inherits: `IChildren`, `Metadata`.
- `Metadata`, inherits: `Root`, `IERC721Metadata`.
- `Resolution`, inherits: `Root`.
- `Root`, inherits: `ControlledERC721`.
- `ControlledERC721`, inherits: `ControllerRole`, `ERC721`, `ERC721Burnable`.

The above inheritance hierarchy is unnecessarily complex. Having many different small contracts which through a complex inheritance hierarchy are forged into a single contract to be deployed has a number of downsides. The readability of the code is decreased, and the difficulty of auditing the contracts increases. Furthermore, the added complexity caused by the inheritance can lead to inheritance related bugs slipping in. Also, there is more time needed to investigate things such as, in which contract is this internally called function defined? Another example would be, are there functions with the same name defined in multiple contracts, which one is executed when? CHAINSECURITY found two inheritance related bugs during this audit. One related to `Resolution._burn` never being called, and another related to the duplicate declaration of `_tokenResolvers`.

[3] https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

When looking at the style of Solidity smart contracts by other respected companies/projects, it becomes clear that making such heavy use of inheritance used to be the standard, but things have been moving away from that and towards contracts that are as flat as possible (vyper does not even have inheritance or the ability to import code from different files). OOP techniques are still used, but to a minimum. Inheritance is mainly used to inherit interfaces, and composition is used to allow connections to other contracts.

Take for example the ENS[4] smart contracts which only use inheritance to inherit interfaces, and use composition to define connections to other contracts. They do not use inheritance to combine multiple smaller contracts into a single big contract.

Another example to look at is the ERC20 implementation of OpenZeppelin. It used to consist of several small contracts[5] which through the use of inheritance were combined into a single ERC20 contract. This style was later replaced with a single `ERC20.sol` contract[6] which only inherited an ERC20 interface. All of the functions are now implemented in this single Solidity file.

The last example which shows this move towards a flat style is the OpenZeppelin `ERC721.sol` contract that is used by the DOT CRYPTO project. The `Registry` contract inherits from `ERC721.sol`. `ERC721` is a single contract which contains all variables and functions that are needed to implement an ERC721 token contract. It is not divided into several small contract which are then mixed together by inheritance.

UNSTOPPABLE DOMAINS is advised to get rid of the unnecessarily complex inheritance hierarchy in the `Registry` contract.

Fixed: UNSTOPPABLE DOMAINS merged the `Registry` related contracts.

[4]https://github.com/ensdomains/ens/blob/master/contracts/HashRegistrar.sol
[5]https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v1.12.0/contracts/token/ERC20/StandardToken.sol
[6]https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.3.0/contracts/token/ERC20/ERC20.sol

Recommendations / Suggestions

- ✓ In the `Metadata._setTokenURI` function there are some TODOs still open. ChainSecurity recommends that these are addressed before deploying contracts on mainnet.

- ✓ In the `SignatureController` contract there are multiple functions which can be called by anyone by providing signed data. Each of these functions calls a function inside the `Registry`. The parameters passed to these functions are part of the signed data. However, the address of the `Registry` contract is not part of the signed data.

  In case there are multiple `Registry` contracts deployed in the future for different top level domains, these pre-signed function calls can be replayed on the different `Registry` contracts. However, if the `Registry` contract address is part of the signed data, this replay would not be possible. Still, the replay would be pointless as the `tokenId` would never be the same if the top level domain is different.

  In case of redeployment of the `Registry` contract for `.crypto` due to some technical issue in the first deployment, adding contract address of `Registry` contract as the part of signed data would avoid replay attacks.

  Still, it's best practice to make the contract address part of the signed data. Unstoppable Domains is recommended to add the `Registry` contract address to the signed data.

- ✓ The following line is present inside `Resolver`, along with a TODO comment:

  ```
  // TODO: f is this necissary?
  require(msg.sender == owner, "SimpleResolver: caller is not the owner");
  ```

  ChainSecurity wants to note that this check is necessary. The ERC721 `ownerOf` function only checks that the owner is not address zero. Therefore, if the above `require` was not present, anybody could set a key + value for any token.

- ☐ The `Resolver` contract has a modifier called `whenResolver` which calls a function in the `Resolution` to check that the resolver for a certain `tokenId` is the `Resolver` contract itself. Inside the `Resolution` this same check is performed inside `sync`. However, in here it is not a modifier. Unstoppable Domains could consider also adding the `whenResolver` modifier to the `Resolution` contract. This would increase the readability of the code.

- ☐ The `SignatureController` contract allows calling any function inside the `Registry` by using pre-signed function call data by the token owner. However, the `ERC721.approve` function does not have a pre-signed variant. Unstoppable Domains could consider also adding such a function.

- ☐ The truffle version used by Unstoppable Domains allows the use of `async/await` inside the javascript code. This includes the migration file. If Unstoppable Domains where to use `async/await` inside the migrations file, the amount of indentation could be lowered. This would increase the readability of the migration file.

- ✓ The `_validate` function is defined in both the `SignatureController` and `SignatureResolver` contract. Although it is a bit different, it is still very similar. Because of this Unstoppable Domains could consider moving this function into a `library`, which is imported by both contracts.

Addendum and General Considerations

Blockchains and especially the Ethereum Blockchain might often behave differently from common software. There are many pitfalls which apply to all smart contracts on the Ethereum blockchain.

In this section, CHAINSECURITY mentions general issues which are relevant to UNSTOPPABLE DOMAINS's code, but do not require a fix. Additionally, in this section CHAINSECURITY clarifies or extends the information from the previous sections of this security report. This section, therefore, serves as a reminder to UNSTOPPABLE DOMAINS and to raise awareness of these issues among potential users.

Dependence on block time information

UNSTOPPABLE DOMAINS uses `now` inside the `SunriseController` contract. Although block time manipulation is considered hard to perform, a malicious miner is able to move forward block timestamps by around 15 seconds compared to the actual time. However, in the context of the project and given the required effort, this is not perceived as an issue[7].

Forcing ETH into a smart contract

Regular ETH transfers to smart contracts can be blocked by those smart contracts. On the high-level this happens if the according solidity function is not marked as `payable`. However, on the EVM levels there exist different techniques to transfer ETH in unblockable ways, e.g. through `selfdestruct` in another contracts. Therefore, many contracts might theoretically observe "locked ETH", meaning that ETH cannot leave the smart contract any more. In most of these cases, it provides no advantage to the attacker and is therefore not classified as an issue.

[7] https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule

Disclaimer

Contact:

ChainSecurity AG
Krähbühlstrasse 58
8044 Zurich, Switzerland
contact@chainsecurity.com