Security in Software Application Assignment

Owanesh

Jan 2024

Abstract

This report will analyze the security of a provided Taxpayer.sol contract. Security will be analyzed and tested through the use of the Echidna tool. To make the code more robust throughout the report only `require()` functions will be used to increase backward compatibility. Where there are lines of code with the exception of `require()`, this will be explained by demonstrating its usefulness based on the assumptions made.

This report is part of the Security in Software Application course at La Sapienza University of Rome. It is therefore not intended as a scientific research paper, but as a report for laboratory exercise.

Contents

1 Introduction to fuzz testing

Fuzz testing is a dynamic testing technique used to discover coding errors and security loopholes in software, by inputting massive amounts of random data, called fuzz, to the system in an attempt to make it crash. This technique is especially effective in finding vulnerabilities in software applications, including smart contracts like in this report.

In the context of smart contracts, other popular tool for fuzz testing is Malticore and Foundry. Echidna is an Haskell[1] program designed for fuzzing/property-based testing of Ethereum smart contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI[2] to falsify user-defined predicates or Solidity assertions. Instead of other softwares, Echidna includes also other tools like slither.

One of the key features of Echidna is its unique 'property-based fuzzing', which tries to falsify user-defined invariants (properties) instead of looking for crashes like a traditional fuzzer. This makes it particularly effective at finding subtle vulnerabilities that might not be caught by other types of testing.

This report focuses on the application of fuzz testing to the Taxpayer.sol contract, a smart contract in the Ethereum blockchain. The contract includes several `require()`[3] statements, which are conditions that must be met for the contract to execute correctly. These conditions serve as the properties that Echidna will attempt to falsify during the fuzz testing process.

The goal of this report is to evaluate the effectiveness of Echidna in identifying potential vulnerabilities in the Taxpayer.sol contract. By examining how Echidna handles various edge cases and unconsidered behaviors, we aim to contribute to the broader discussion on improving the security of smart contracts.

2 Other testing tools

There are several tools available for conducting security checks on smart contracts, each designed to identify and mitigate potential vulnerabilities that could compromise the integrity and security of blockchain-based applications. Solidity static analyzers such as Myhtil and Slither[4] are widely used to perform automated scans of smart contract code, flagging potential security issues.

2.1 Mythril

Mythril is a powerful open-source security analysis tool specifically designed for Ethereum smart contracts. It performs static and dynamic analysis to detect a wide range of security issues, including potential vulnerabilities such as reentrancy attacks, integer overflows, and more.

Mythril supports various installation methods, including pip (Python package manager) and Docker. You can find detailed installation instructions on the official Mythril GitHub repository: `github.com/ConsenSys/mythril`. The method chosed in this report is via Docker due to compatibility with python3.12. Below there are execution command and respective output given by tool.

Before we look at what changes are needed to make the contract more secure and robust, let's take a look at what Mythril's report on the contract provided. (currently available as original.Taxpayer.sol)

```
$: docker run -v $(pwd):/tmp mythril/myth analyze /tmp/original.Taxpayer.sol
```

[1] Haskell is a functional programming language

[2] A contract ABI, or Application Binary Interface, in the context of Ethereum, is essentially a specification for how to interact with a contract on the Ethereum blockchain

[3] The require() function in Solidity is used for input validation and conditional checking. It throws an exception and terminates execution if the specified condition is not met.

[4] Already included into Echidna stack

```
==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Low
Contract: Taxpayer
Function name: transferAllowance(uint256)
PC address: 614
Estimated Gas Usage: 10205 - 99568
A call to a user-supplied address is executed.
An external message call to an address specified by the caller is executed. Note that the callee
↪  account might contain arbitrary code and could re-enter any function within this contract.
↪  Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that
↪  no state modifications are executed after this call and/or reentrancy guards are in place.
--------------------
In file: /tmp/original.Taxpayer.sol:56

sp.getTaxAllowance()

--------------------
Caller: [CREATOR], function: marry(address), txdata:
↪  0xbccb358e000000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef, decoded_data:
↪  ('0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef',), value: 0x0
Caller: [ATTACKER], function: transferAllowance(uint256), txdata:
↪  0x5f68c4390000000000000000000000000000000000000000000000000000000000001121, decoded_data: (4385,),
↪  value: 0x0
```

and analog results are available also for

```
56.    sp.setTaxAllowance(sp.getTaxAllowance()+change)
47.    sp.setSpouse(address(0))
48.    spouse = address(0)
```

As part of the security analysis conducted on the provided smart contract, Mythril revealed a number of functions that could have potential vulnerabilities. The full report of this analysis is available in the attached Mythril.report file available on repository[5]. The findings highlight several areas of concern, including possible risks related to reentrancy[6] (line 159 of report). In addition, it should be noted that the synergic integration of Mythril with Echidna provided substantial support in defining the constraints (require) necessary to mitigate the identified vulnerabilities. The combination of these two powerful resources provides a comprehensive overview of contract security. At the end of listed enhancement described in this report, the result of Mythril is this reported below

```
$: docker run -v $(pwd):/tmp mythril/myth analyze /tmp/Taxpayer.sol
>>> The analysis was completed successfully. No issues were detected.
$: docker run -v $(pwd):/tmp mythril/myth analyze /tmp/TaxpayerTesting.sol
>>> The analysis was completed successfully. No issues were detected.
```

3 Echidna setup

In order to write a full suite of tests, echidna offers two way o write a Tester contract. The first one represented below, is a test with a sort of whitelist on method which are allowed to be called.[7]

Another method is by inheritance

TaxpayerTesting.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;

import "./Taxpayer.sol";
```

[5] github.com/owanesh/SSA2324/blob/master/report/mythril.report

[6] Reentrancy vulnerability in smart contracts occurs when external calls can be reentered before completing the initial operation, potentially leading to unintended and malicious behavior.

[7] github.com/crytic/echidna/wiki/How-to-use-Echidna-with-multiple-contracts

```

contract TaxpayerTesting is Taxpayer {
    uint constant ADULT_AGE = 18;
    uint constant ADULT_OLD_AGE = 65;
    Taxpayer alpha;
    Taxpayer bravo;

    constructor() Taxpayer(address(0), address(0)) {
        alpha = new Taxpayer(address(0), address(0));
        bravo = new Taxpayer(address(0), address(0));
        for (uint i = 0; i < ADULT_AGE; i++) {
            alpha.haveBirthday();
            bravo.haveBirthday();
        }
        bravo.marry(address(alpha));
        alpha.marry(address(bravo));
    }
...
```

By inheriting from Taxpayer, the testing contract gains access to all the public and external functions of Taxpayer, enabling comprehensive testing without the need for explicit method declarations in TaxpayerTesting. This approach can enhance the efficiency of testing, especially in scenarios where extensive coverage of the target contract's functionality is desired. Additionally, the second method seems to align with a fuzzer-based testing strategy, wherein automatic identification of public methods is crucial for generating diverse inputs during testing, as opposed to relying on a predefined whitelist. The decision to adopt the second method is likely driven by the desire for flexibility, automation, and a more dynamic testing environment.

4 If person x is married to person y, then person y should be married to person x

This function can be securely validated only through the proper execution of their respective marry functions. It is crucial that both marry functions are called accurately to maintain the consistency of marriage states. Therefore, the validation of this property is inherently tied to the accurate implementation and execution of the marry functions. An optimal implementation might involve code optimizations, ensuring that the marry function of one person correctly invokes the marry function of the spouse, thereby making the contract more robust and resilient to potential logic errors that could compromise the correctness of the application.

```
function marry(address newSpouse) public {
    <all_require()_conditions>
    spouse=newSpouse;
    isMarried = true;
    if(!Taxpayer(address(newSpouse)).getIsMarried())
        Taxpayer(address(newSpouse)).marry(address(this));
}
```

In this way we achieve the isMarried status on both contract with one transaction.

The decision to not implement the marriage verification within the marry function could stem from the principle of separation of concerns and the idea that a contract should ideally modify only its own state. In a well-designed system, each contract should be responsible for managing its own data and logic independently

4.1 Analyze the original code

original.Taxpayer.sol

```
function marry(address new_spouse) public {
    spouse = new_spouse;
```

```
    isMarried = true;
    }
```

Looking at these lines of code, one can immediately see some critical issues that make the contract vulnerable to potential unwanted behavior. This is caused by the absence of some security measures within the code Let us delve into the technical description of some possible vulnerabilities:

- Ability to marry a nonexistent address (address(0))
- Self-Marriage Exploitation:
  - The code lacks a check to ensure the **newSpouse** address is different from the caller's address.
  - Allows a user to marry themselves, potentially leading to unexpected complications.
- Overwriting Past Marriages:
  - No verification for whether the caller is already married before executing a new marriage.
  - Enables repetitive invocation of the **marry** function, overwriting past marriages without constraints.

There are then other useful checks to increase the robustness of the code and avoid :

- You cannot marry twice the same address
- You cannot marry if you are already married, then **getSpouse()**$\neq$**address(0)**
- Your spouse address needs to be a valid address

Finally, some inserted checks are more "logical" such as:

- You cannot marry with your parents
- You cannot marry if your are under sixteen
- Your spouse needs to be not married or divorced by previous marriage

A possible implementation of requirements contraints is listed below. Every **<require()>** function is composed by **<condition>** and **<reason>**. The best way to obtain a 100% retrocompatible code.

Taxpayer.sol

```
function marry(address newSpouse) public {
    require(age > 16, "You must have at least 16 years old");
    require(
        newSpouse != address(parent1) && newSpouse != address(parent2),
        "You cannoy marry with your parents"
    ); // marriage with siblings is allowed by code
    require(newSpouse != address(this), "You cannot marry with yourself");
    require(newSpouse != getSpouse(), "Already married to this spouse");
    require(
        spouse == address(0) && getIsMarried() == false,
        "Already married"
    );
    require(newSpouse != address(0), "Invalid spouse address");
    require(
        address(Taxpayer(address(newSpouse))).code.length > 0,
        "Invalid spouse, is it already born?" //exploitable if new_spouse has another type of
        ↪  contract
    );
    require(
        (Taxpayer(address(newSpouse)).getSpouse() == address(0) &&
            Taxpayer(address(newSpouse)).getIsMarried() == false) ||
            (Taxpayer(address(newSpouse)).getIsMarried() == true &&
```

```
            Taxpayer(address(newSpouse)).getSpouse() == address(this)),
        "Your partner should be single or at least not married with another person"
    );
    spouse = newSpouse;
    isMarried = true;
}
```

One way to check that all these constraints are valid and do not block the normal behavior of the contract is to use echidna, which will do fuzztesting on our contract. Are published below the written test, and the results obtained with `original.Taxpayer.sol` and those obtained as a result of the modifications. In this way used approach is similar to TDD[8] (mixed with BDD[9]), where the requirements are written in the test, and after that the code is modified to be compatible.

TaxpayerTesting.sol

```
function echidna_simple_marry() public view returns (bool) {
    bool alpha_to_bravo = alpha.getSpouse() == address(bravo) &&
        alpha.getSpouse() != address(0);
    return alpha_to_bravo;
}

function echidna_both_married() public view returns (bool) {
    bool alpha_to_bravo = alpha.getSpouse() == address(bravo) &&
        alpha.getSpouse() != address(0);
    bool bravo_to_alpha = bravo.getSpouse() == address(alpha) &&
        bravo.getSpouse() != address(0);
    return alpha_to_bravo && bravo_to_alpha;
}

function echidna_divorce() public returns (bool) {
    bravo.divorce();
    bool alpha_is_divorced = alpha.getSpouse() == address(0) &&
        alpha.getIsMarried();
    bool bravo_is_divorced = bravo.getSpouse() == address(0) &&
        !bravo.getIsMarried();
    return alpha_is_divorced && bravo_is_divorced;
}
```

results on original.Taxpayer.sol

```
echidna_both_married: failed!   [X]
Call sequence:
    marry(0x62d69f6867a0a084c6d313943dc22023bc263691)
    divorce()

echidna_simple_marry: failed!   [X]
Call sequence:
    setSpouse(0xb4c79dab8f259c7aee6e5b2aa729821864227e84)
    divorce()

echidna_divorce: failed!    [X]
Call sequence:
    marry(0x62d69f6867a0a084c6d313943dc22023bc263691)
    divorce()

Event sequence:
error Revert 0x
```

The execution results from Echidna reveal failures in three distinct scenarios. Echidna finds with fuzzing action, a call-sequence able to break the tests. All finded path include an alteration of `spouse` variable, via `setSpouse()` called arbitrary or via `marry()`. With original code some of those

[8] TDD, a software development method, advocates for test creation preceding code writing.

[9] BDD is a collaborative software development approach emphasizing natural language specifications and executable tests.

scenario are allowed because there are no control on validity of spouse nor validity on `setSpouse()` call. In order to fix is important to analyze lines 47,48,52 and 53 of Taxpayer.sol, added require() ensure that newSpouse is a valid address, and the caller contract is not already married with another spouse. To ensure that a contract should be married only with another contract already deployed on blockhain, the control used is `address(Taxpayer(address(newSpouse))).code.length > 0`[10]

results on Taxpayer.sol

```
echidna_both_married: passing
echidna_simple_marry: passing
echidna_divorce: passing
```

5 Married persons can pool their tax allowance as long as the sum of their tax allowances remains the same

The tax pooling, based on the provided code, appears to have several issues. The foremost concern is the ability to invoke the method with a `change` value higher than the current available `tax_allowance`. This would inevitably lead to a runtime error as `tax_allowance` is defined as a uint. It is always preferable to prevent runtime errors rather than encountering them during execution. To address this potential issue, a condition has been introduced to ensure that the available `tax_allowance` is greater than the value of `change` to be transferred. Additionally, an extra check (*not explicitly requested*) has been implemented to prevent "dummy" operations, ensuring that the input value of `change` is greater than zero.

original.Taxpayer.sol

```
function transferAllowance(uint change) public {
    tax_allowance = tax_allowance - change;
    Taxpayer sp = Taxpayer(address(spouse));
    sp.setTaxAllowance(sp.getTaxAllowance()+change);
}
```

Furthermore, the assignment specifies that tax pooling should only be granted to married couples. Hence, the introduction of the require statements in lines 116 and 122. The first ensures that the caller, i.e., the one initiating the transfer, is indeed married. The second checks for the reciprocity constraint, ensuring that the recipient is married to the caller.

Taxpayer.sol

```
110 function transferAllowance(uint256 change) public {
111     require(
112         change > 0,
113         "Don't waste gas, save the world, use proper change value"
114     );
115     require(taxAllowance >= change, "Insufficient tax allowance");
116     require(
117         getIsMarried() && getSpouse() != address(0),
118         "You have to be married before pooling tax allowance"
119     );
120     taxAllowance -= change;
121     Taxpayer sp = Taxpayer(address(spouse));
122     require(
123         sp.getSpouse() == address(this) && sp.getIsMarried(),
124         "You cannot change allowance of person not married with you"
125     );
126     sp.setTaxAllowance(sp.getTaxAllowance() + change);
127 }
```

Just as with the `transferAllowance(uint256 change)`, the original form of the `setTaxAllowance(uint256 ta)` function appears to have security issues, most notably the ability to assign any arbitrary value without any restrictions.

[10] stackoverflow.com/a/74511610

Before examining the applied modifications, it is essential to consider an assumption made during the project's development: specifically, that `setTaxAllowance(uint256 ta)` should only be called by one's spouse through the `transferAllowance(uint256 change)` function. This assumption is crucial for understanding the nature of the various `require()` statements that have been added.

original.Taxpayer.sol

```
function setTaxAllowance(uint ta) public {
    tax_allowance = ta;
}
```

With the assumptions established earlier and the previously discussed analysis of requirements for the `transferAllowance` function, understanding the importance of the initial three `require()` statements in the Taxpayer.sol code becomes more accessible.

There is then a final `require()` statement that validates the maximum sum a couple can obtain. Specifically, in the case of a couple where both members are under 65 years old, the sum of their `taxAllowance` must be equal to `2 * DEFAULT_ALLOWANCE`.

Examining the code of `transferAllowance`, it's noticeable that the `change` parameter is subtracted first and then added to the spouse's `taxAllowance`. This approach simplifies the sum check since the `ta` parameter in `setTaxAllowance` is already one of the two addends. The remaining analysis involves the `getTaxAllowance()` of the spouse (*which, in the previous step, has already been subtracted by* `change` *to exclude the scenario where it is considered twice*).

Taxpayer.sol

```
function setTaxAllowance(uint256 ta) public {
    require(
        ta > 0,
        "Don't waste gas, save the world, use proper change value"
    );
    require(
        getSpouse() != address(0),
        "Someone that isn't married with you, tried to change your tax allowance"
    );
    require(
        ta > getTaxAllowance(),
        "Someone tries to decrease illegally your tax allowance"
    );
    Taxpayer spoused = Taxpayer(address(getSpouse()));
    require(
        ((getAge() < 65 &&
            spoused.getAge() < 65 &&
            ta + spoused.getTaxAllowance() == 2 * DEFAULT_ALLOWANCE) ||
            (getAge() >= 65 &&
                spoused.getAge() >= 65 &&
                ta + spoused.getTaxAllowance() == 2 * ALLOWANCE_OAP) ||
            (getAge() < 65 &&
                spoused.getAge() >= 65 &&
                ta + spoused.getTaxAllowance() ==
                DEFAULT_ALLOWANCE + ALLOWANCE_OAP) ||
            (getAge() >= 65 &&
                spoused.getAge() < 65 &&
                ta + spoused.getTaxAllowance() ==
                DEFAULT_ALLOWANCE + ALLOWANCE_OAP)),
        "Tax pooling violation"
    );
    taxAllowance = ta;
}
```

6 People aged 65 and over have a higher tax allowance, of 7000

To introduce this functionality within the code, it was necessary to modify the `haveBirthday()` function, granting an extra `ALLOWANCE_OAP-DEFAULT_ALLOWANCE` to the contract when it reaches an age of 65. The choice was made to use increment by difference so as not to overwrite potential allowance changes. (In fact, it would otherwise have been possible to transfer all the allowance, getting to 0, and then at age 65 having 7000 again.)

Taxpayer.sol

```
function haveBirthday() public {
    age++;
    if (age == 65 && !getIsMarried())
        taxAllowance += (ALLOWANCE_OAP - DEFAULT_ALLOWANCE); // added lines
    else if (age == 65 && getIsMarried())
        this.setTaxAllowance(this.getTaxAllowance()+(ALLOWANCE_OAP - DEFAULT_ALLOWANCE));
}
```

The need to have to distinguish the state in which `isMarried` is derived from the fact that one of the assumptions of this report, is that certain functions can only be called in certain contexts. And among these is `setTaxAllowance(uint ta)`, which can only be called via `transferAllowance(uint change)`, which can by definition only be called if `isMarried=true`.

The choice to use the set method instead of direct assignment comes from wanting to further control the validity of operations. Another useful requirement to ensure the required functionality is the one inserted in the `setTaxAllowance(uint ta)`, where the age of the couple is taken into account, ensuring that the sum of the two can be a maximum of `2*ALLOWANCE_OAP` if both are over 65 years old, or `DEFAULT_ALLOWANCE + ALLLOWANCE_OAP` where the couple is of mixed age. (from line 118 of Taxpayer.sol)

7 Extras

7.1 Style edits and warning removal

In all files within the repository, the `slither-format`[11] command was run, which provides a patch to align the code with what the guidelines are (example new_spouse written in camelCase instead of snake_case).

Two other warnings have been resolved, the first concerns the use of the ‘view‘ identifier for those functions that do not change state, typical of getter() methods.

Taxpayer.sol

```
function getTaxAllowance() public view returns (uint256) {
    return taxAllowance;
}
```

The last warning concerns the `SPDX-License-Identifier`[12], set as a common practice under MIT License that permits users to use, modify, and distribute software freely, requiring only that the original license and copyright notice be included.

7.2 Useful (getter) function

Two get functions were added to improve the effectiveness on validation checks. These functions do not alter the state of the contract in fact they are defined with `view` identifier.

[11] github.com/crytic/slither/wiki/Slither-format

[12] The code comment `// SPDX-License-Identifier:<license>` serves as a standardized declaration of the software's licensing terms, specifying that it is released under the certain License.

Taxpayer.sol

```
function getAge() public view returns (uint256) {
    return age;
}

function getIsMarried() public view returns (bool) {
    return isMarried;
}
```

7.3 Requirements

Some conditions have been added to make the code more ”real,” preventing undesirable behavior. The code provided does not require any of these changes, so they can be removed, but these provide robustness to the code and ”logical” continuity.

7.3.1 Can born only from addr(0x0) or married couple: For example a constraint added require a contract needs to be created with 2 parents options.

- You can have parent1 and 2 as address(0)
- You can have parent1 and 2 different than address(0) only if they are married each others.

The choice of second requirements isn’t needed, of course in real life you can have a baby out of a marriage, but for this scenario we assume you cannot.

Taxpayer.sol

```
19 constructor(address p1, address p2) {
20     require(
21         (p1 == address(0) && p2 == address(0)) ||
22             (Taxpayer(p1).getSpouse() == p2 &&
23                 Taxpayer(p2).getSpouse() == p1),
24         "A new born is allowed only form init and married couple"
25     );
26     age = 0;
27     isMarried = false;
28     parent1 = p1;
29     parent2 = p2;
30     spouse = (address(0));
31     income = 0;
32     tax_allowance = DEFAULT_ALLOWANCE;
33 }
```

This improvement is validated by the following test which operates through the use of a `try{}catch{}` statement.

TaxpayerTesting.sol

```
function echidna_block_spawn_of_orphan() public returns (bool) {
    try new Taxpayer(address(1), address(2)) returns (Taxpayer) {
        return false;
    } catch {
        return true; // exception was raised
    }
}
```

As you can read from the test, the code tries to create a contract with `address(1)` and `address(2)` as parent1 and parent2 respectively. But the expected result is that the test fails because of the require on line 21 of Taxpayer.sol. So we handle this event inside the catch{} block. Another test, more easy to read is shown below: (*Notice: alpha and bravo are married from constructor*)

TaxpayerTesting.sol

```
function echidna_couple_make_a_baby() public returns (bool) {
    try new Taxpayer(address(alpha), address(bravo)) {
        return true; // works successfully
    } catch {
        return false; // an exception was raised by contract
    }
}
```

7.3.2 setSpouse valid only for divorce purpose:

The one proposed is an assumption that was made to make the code fluid, namely that setSpouse can be called as from the original code, only via the `divorce()` function. Obviously from the original code no such constraint is present, but the decision to include it forces users not to be able to call a `setSpouse(address sp)` on themselves with a dummy or invalid address, thus avoiding unwanted behavior.

Taxpayer.sol

```
function setSpouse(address sp) public {
    require(sp != address(this), "You cannot call setSpouse with yourself");
    require(
        (getSpouse() != address(0) && getIsMarried() && sp == address(0)),
        "You are already married, you can call this function only for divorce purpose now"
    );
    spouse = (address(sp));
}
```

With the condition placed in the require, three conditions must be validated simultaneously ”sp” as an input parameter must be equal to `addr(0)` the contract executing the `setSpouse(address sp)` must be `isMarried=True` and then its `getSpouse()` must return an address other than `addr(0)`

7.3.3 Solve tax pooling before divorce:

Taxpayer.sol

```
function divorce() public {
    require(
        getSpouse() != address(0) && getIsMarried(),
        "You're not already married"
    );
    Taxpayer sp = Taxpayer(address(spouse));
    require(
        sp.getSpouse() == address(this) && sp.getIsMarried(),
        "That person isn't married with you"
    );
    require(
        (getTaxAllowance() == DEFAULT_ALLOWANCE && age < 65) ||
            ((getTaxAllowance() == ALLOWANCE_OAP && age >= 65) &&
                (sp.getTaxAllowance() == DEFAULT_ALLOWANCE && sp.getAge() < 65)) ||
            (sp.getTaxAllowance() == ALLOWANCE_OAP && sp.getAge() >= 65),
        "Before divorcing, fix your tax pool allowance"
    );
    /**
        if(getAge()>=65) {setTaxAllowance(ALLOWANCE_OAP);}
        else {setTaxAllowance(DEFAULT_ALLOWANCE);}
     */
    sp.setSpouse(address(0));
    spouse = address(0);
    /*
    sp.divorce(); // instead of sp.setSpouse(address(0));
    */
    isMarried = false;
}
```

In real life scenario if we have previously taxpooled with our partner and then decide to get divorced, everyone has to take back what they are fiscally entitled to, without being able to access any deductions. That is why a special requirement has been included. The method for bypassing the require has also been written if it is deemed unnecessary and indeed wasteful in terms of transactions (*See annotated code on lines 88 - 89.*)

From the snippet shown above another interesting detail also emerges, that the way the code has been provided if Alpha after being married to Bravo, performs a `divorce()` its `isMarried` status changes to False, while Bravo's remains at True. The way the various `require()` were implemented this does not turn out to be a problem since the checks were done based on the `getSpouse()`, but for better performance, as well as for the `marry(address newSpouse)` which could be modified to alter the state of both contracts, the same can be done for the divorce by uncommenting the line 94

7.3.4 Ensure to be married before divorce:

As can be seen from the requirement of line 71 of Taxpayer.sol, within the `divorce()` method, the clause that the divorce caller must necessarily be married has been inserted, and the reciprocity constraint has also been added, i.e., that the person set as the spouse is actually married to the divorce caller (line 76)

7.4 Echidna through CI/CD pipeline