

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. Of the various possible ways including adding invariants or directly modifying the code, the use of requires was preferred as they are more versatile.

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	2
2	Other testing tools	2
2.1	Mythril	2
3	Echidna setup	3
4	If person x is married to person y, then person y should be married to person x	4
4.1	Analyze the original code	5
5	Married persons can pool their tax allowance as long as the sum of their tax allowances remains the same	7
5.1	Testing	9
6	People aged 65 and over have a higher tax allowance, of 7000	9
7	Extras	10
7.1	Every person has an income tax allowance on which no tax is paid	10
7.2	Style edits and warning removal	11
7.3	Useful (getter) function	12
7.4	Requirements	12
7.4.1	Can born only from addr(0x0) or married couple	12
7.4.2	setSpouse valid only for divorce purpose	13
7.4.3	Solve tax pooling before divorce	13
7.4.4	Ensure to be married before divorce	14
7.5	Echidna through CI/CD pipeline	14

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 Haskell¹ program designed for fuzzing/property-based testing of Ethereum smart contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI² 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 Mythril and Slither⁴ 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 conducts static and dynamic analyses to identify a broad spectrum 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. Detailed installation instructions can be found on the official Mythril GitHub repository: github.com/ConsenSys/mythril. The method chosen in this report is via Docker due to compatibility with Python 3.12. Below, you will find the execution command and the respective output provided by the tool.

Before we delve into the changes needed to enhance the contract's security and robustness, let's review the report that Mythril generated for the provided contract (currently available as `original.Taxpayer.sol`).

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

¹Haskell is a functional programming language

²A 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

³The `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.

⁴Already included into Echidna stack


```

5
6 contract TaxpayerTesting is Taxpayer {
7     uint constant ADULT_AGE = 18;
8     uint constant ADULT_OLD_AGE = 65;
9     Taxpayer alpha;
10    Taxpayer bravo;
11
12    constructor() Taxpayer(address(0), address(0)) {
13        alpha = new Taxpayer(address(0), address(0));
14        bravo = new Taxpayer(address(0), address(0));
15        for (uint i = 0; i < ADULT_AGE; i++) {
16            alpha.haveBirthday();
17            bravo.haveBirthday();
18        }
19        bravo.marry(address(alpha));
20        alpha.marry(address(bravo));
21    }
22    ...

```

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

```
40      original.Taxpayer.sol
41      function marry(address new_spouse) public {
42          spouse = new_spouse;
43          isMarried = true;
44      }
```

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() != 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 you are under sixteen
- Your spouse needs to be not married or divorced by previous marriage

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

```
40      Taxpayer.sol
41      function marry(address newSpouse) public {
42          require(age > 16, "You must have at least 16 years old");
43          require(
44              newSpouse != address(parent1) && newSpouse != address(parent2),
45              "You cannot marry with your parents"
46          ); // marriage with siblings is allowed by code
47          require(newSpouse != address(this), "You cannot marry with yourself");
48          require(newSpouse != getSpouse(), "Already married to this spouse");
49          require(
50              spouse == address(0) && getIsMarried() == false,
51              "Already married"
52          );
53          require(newSpouse != address(0), "Invalid spouse address");
54          require(
55              address(Taxpayer(address(newSpouse))).code.length > 0,
56              "Invalid spouse, is it already born?"
57          );
58          require(
```

```

58         (Taxpayer(address(newSpouse)).getSpouse() == address(0) &&
59         Taxpayer(address(newSpouse)).getIsMarried() == false) ||
60         (Taxpayer(address(newSpouse)).getIsMarried() == true &&
61         Taxpayer(address(newSpouse)).getSpouse() == address(this)),
62         "Your partner should be single or at least not married with another person"
63     );
64     spouse = newSpouse;
65     isMarried = true;
66 }

```

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 the 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 TDD⁸ (mixed with BDD⁹), where the requirements are written in the test, and after that the code is modified to be compatible.

```

TaxpayerTesting.sol
65 function echidna_simple_marry() public view returns (bool) {
66     bool alpha_to_bravo = alpha.getSpouse() == address(bravo) &&
67     alpha.getSpouse() != address(0);
68     return alpha_to_bravo;
69 }
70
71 function echidna_both_married() public view returns (bool) {
72     bool alpha_to_bravo = alpha.getSpouse() == address(bravo) &&
73     alpha.getSpouse() != address(0);
74     bool bravo_to_alpha = bravo.getSpouse() == address(alpha) &&
75     bravo.getSpouse() != address(0);
76     return alpha_to_bravo && bravo_to_alpha;
77 }
78
79 function echidna_divorce() public returns (bool) {
80     bravo.divorce();
81     bool alpha_is_divorced = alpha.getSpouse() == address(0) &&
82     alpha.getIsMarried();
83     bool bravo_is_divorced = bravo.getSpouse() == address(0) &&
84     !bravo.getIsMarried();
85     return alpha_is_divorced && bravo_is_divorced;
86 }

```

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

```

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

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

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 found path include an alteration of `spouse` variable, via `setSpouse()` called arbitrary or via `marry()`. With original code some of those scenario are allowed because there are no controls 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 blockchain, the control used is `address(Taxpayer(address(newSpouse))).code.length > 0`¹⁰. A proposed fix on `setSpouse` also in described on `setSpouse()` section.

————— 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 —————

```

55 function transferAllowance(uint change) public {
56     tax_allowance = tax_allowance - change;
57     Taxpayer sp = Taxpayer(address(spouse));
58     sp.setTaxAllowance(sp.getTaxAllowance()+change);
59 }

```

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     );

```

¹⁰stackoverflow.com/a/74511610

```

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.

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.

```

71     function setTaxAllowance(uint ta) public {
72         tax_allowance = ta;
73     }

```

original.Taxpayer.sol

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*).

```

100     function setTaxAllowance(uint256 ta) public {
101         require(
102             ta > 0,
103             "Don't waste gas, save the world, use proper change value"
104         );
105         require(
106             getSpouse() != address(0),
107             "Someone that isn't married with you, tried to change your tax allowance"
108         );
109         require(
110             ta > getTaxAllowance(),
111             "Someone tries to decrease illegally your tax allowance"
112         );
113         Taxpayer spoused = Taxpayer(address(getSpouse()));
114         require(
115             ((getAge() < 65 &&
116              spoused.getAge() < 65 &&
117              ta + spoused.getTaxAllowance() == 2 * DEFAULT_ALLOWANCE) ||
118              (getAge() >= 65 &&
119               spoused.getAge() >= 65 &&
120               ta + spoused.getTaxAllowance() == 2 * ALLOWANCE_OAP) ||
121              (getAge() < 65 &&
122               spoused.getAge() >= 65 &&
123               ta + spoused.getTaxAllowance() ==
124                DEFAULT_ALLOWANCE + ALLOWANCE_OAP) ||
125              (getAge() >= 65 &&
126               spoused.getAge() < 65 &&
127               ta + spoused.getTaxAllowance() ==
128                DEFAULT_ALLOWANCE + ALLOWANCE_OAP)),
129             "Tax pooling violation"
130         );

```

Taxpayer.sol


```

131     taxAllowance = ta;
132 }

```

5.1 Testing

```

_____ TaxpayerTesting.sol _____
function echidna_transfer_allowance() public returns (bool) {
    alpha.transferAllowance(4000);
    return
        bravo.getTaxAllowance() + alpha.getTaxAllowance() ==
        2 * DEFAULT_ALLOWANCE;
}
function echidna_transfer_huge_amount_of_allowance() public returns (bool) {
    try alpha.transferAllowance(40000) {
        return false;
    } catch {
        return true;
    }
}
}

```

```

_____ results on original.Taxpayer.sol _____
echidna_transfer_allowance: failed! [X]
  Call sequence:
    marry(0x62d69f6867a0a084c6d313943dc22023bc263691)
    transferAllowance(1)

echidna_transfer_huge_amount_of_allowance: failed! [X]
  Call sequence:
    setTaxAllowance(35514)
    setSpouse(0xb4c79dab8f259c7aee6e5b2aa729821864227e84)
    transferAllowance(35044)

```

As shown by the fuzzer results, with the original code there is a path, then a call sequence such that the constraints required by the test, are not met. This is because the original code places no constraints on functions such as `marry(address sp)`, `setTaxAllowance` or `setSpouse`.

Basically, the `echidna_transfer_huge_amount_of_allowance` test shows how it is possible to set an arbitrary value of `taxAllowance`, after which set a spouse (without going through `marry`) and after which do a transfer. From the original code this behavior is permissible, but since it is undesirable that is why the requirements with the constraints on marriage and reciprocity were inserted. The `echidna_transfer_allowance` test on the other hand shows how it is possible to do a `marry` on an address (in this case the fuzzer is using the address of `alpha`) and then transfer 1, also this behavior from the original code is permissible. This behavior is undesirable since before transferring the `taxAllowance` we must necessarily check the reciprocity constraint. If fuzzer marries with `alpha`, and transfers 1, then `alpha`'s allowance will always be greater than expected, even when `alpha` decides to join in marriage with `bravo`. To highlight the effectiveness of the inserted requirements, the result of the usual tests performed instead on `Taxpayer.sol` is shown below

```

_____ results on Taxpayer.sol _____
echidna_transfer_allowance: passing
echidna_transfer_huge_amount_of_allowance: passing

```

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.)

```

167 function haveBirthday() public {
168     age++;
169     if (age == 65 && !getIsMarried())
170         taxAllowance += (ALLOWANCE_OAP - DEFAULT_ALLOWANCE);
171     else if (age == 65 && getIsMarried())
172         this.setTaxAllowance(this.getTaxAllowance()+(ALLOWANCE_OAP - DEFAULT_ALLOWANCE));
173 }

```

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 $2 \times \text{ALLOWANCE_OAP}$ if both are over 65 years old, or $\text{DEFAULT_ALLOWANCE} + \text{ALLOWANCE_OAP}$ where the couple is of mixed age. (from line 118 of `Taxpayer.sol`)

Below are reported the tests.

```

function echidna_allowance_older_than_sixtyfive_for_both() public returns bool {
    be_old(bravo);
    be_old(alpha);
    alpha.transferAllowance(2000);
    return bravo.getTaxAllowance() + alpha.getTaxAllowance() == ALLOWANCE_OAP + ALLOWANCE_OAP;
}

function echidna_allowance_older_than_sixtyfive_for_only_one() public returns (bool) {
    be_old(bravo);
    alpha.transferAllowance(2000);
    return bravo.getTaxAllowance() + alpha.getTaxAllowance() == DEFAULT_ALLOWANCE + ALLOWANCE_OAP;
}

```

```

results on Taxpayer.sol
echidna_allowance_older_than_sixtyfive_for_only_one: passing
echidna_allowance_older_than_sixtyfive_for_both: passing

```

7 Extras

7.1 Every person has an income tax allowance on which no tax is paid

From the provided code, it appears that the value of "income" is initially set to 0 and is not subsequently modified. This is also due to the absence of getter and setter methods. Below is a potential fix, primarily addressing the implementation of these two methods. It is important to note that, in order to secure the setter method, it must include constraints on the rule policy of those who can invoke the function. This is essential to prevent a contract from calling `setIncome` on itself by temporarily declaring a lower value and then subsequently declaring a higher one. To mitigate this, the use of Role-Based Access Control (RBAC)¹¹ policies is recommended. To implement this specification, assuming the existence of secure getter and setter methods, the code can be structured as follows:

¹¹Is a security model that restricts system access to authorized users based on their assigned roles and responsibilities, enhancing access management and reducing potential security risks.

```

function getIncome() public view returns (uint256) {
    return income;
}

// Secure setter method with RBAC control (added onlyAdmin invariant)
function setIncome(uint256 _newIncome) public onlyAdmin {
    // useful require() to avoid unexpected behaviour
    income = _newIncome;
}

function isTaxableIncome(bool considerCouple) public returns (bool){
    if(considerCouple){
        require(getSpouse() != address(0) && getIsMarried(), "You have to be marry to use
        ↪ considerCouple flag");
        Taxpayer spouse = Taxpayer(address(getSpouse()));
        return getIncome() + spouse.getIncome() <= getTaxAllowance() + spouse.getTaxAllowance();
    }else {
        return getIncome() < getTaxAllowance();
    }
}

function getTaxableIncome(bool considerCouple) public returns (uint256){
    if(!isTaxableIncome(considerCouple)) return getIncome();
    if(considerCouple){
        Taxpayer spouse = Taxpayer(address(getSpouse()));
        return getIncome() - getTaxAllowance() - (spouse.getIncome() + spouse.getTaxAllowance());
    }
    else{
        return getIncome() - getTaxAllowance();
    }
}

```

the `considerCouple` flag is used to allow the user to assess his income even if taxpooling has been done. In the case where `considerCouple = false`, then the person is considered to be a single individual, even if he is married (*this could be detrimental at the time when a person has transferred all of his taxallowance and then wants to assess the taxability of his income, which will be 100 percent*). In the case where `considerCouple = true` then the two `taxAllowances` will be added together, subtracted from the spouse's income, and that value represents the new "taxAllowance" to be used to first assess whether the income is taxable, and if so, by how much.

7.2 Style edits and warning removal

In all files within the repository, the `slither-format`¹² 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.

```

163                                     Taxpayer.sol
164 function getTaxAllowance() public view returns (uint256) {
165     return taxAllowance;
166 }

```

The last warning concerns the `SPDX-License-Identifier`¹³, 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.

¹²github.com/crytic/slither/wiki/Slither-format

¹³The 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.

7.3 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.

```
179 function getAge() public view returns (uint256) {
180     return age;
181 }
182
183 function getIsMarried() public view returns (bool) {
184     return isMarried;
185 }
```

7.4 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.

7.4.1 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.

```
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`.

```
87 function echidna_block_spawn_of_orphan() public returns (bool) {
88     try new Taxpayer(address(1), address(2)) returns (Taxpayer) {
89         return false;
90     } catch {
91         return true; // exception was raised
92     }
93 }
```

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*)

```

37 _____ TaxpayerTesting.sol _____
38 function echidna_couple_make_a_baby() public returns (bool) {
39     try new Taxpayer(address(alpha), address(bravo)) {
40         return true; // works successfully
41     } catch {
42         return false; // an exception was raised by contract
43     }
44 }

```

7.4.2 setSpouse valid only for divorce purpose: The code proposed below helps the understanding of marry() function explained above. 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.

```

100 _____ Taxpayer.sol _____
101 function setSpouse(address sp) public {
102     require(sp != address(this), "You cannot call setSpouse with yourself");
103     require(
104         (getSpouse() != address(0) && getIsMarried() && sp == address(0)),
105         "You are already married, you can call this function only for divorce purpose now"
106     );
107     spouse = (address(sp));
108 }

```

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)

7.4.3 Solve tax pooling before divorce:

```

70 _____ Taxpayer.sol _____
71 function divorce() public {
72     require(
73         getSpouse() != address(0) && getIsMarried(),
74         "You're not already married"
75     );
76     Taxpayer sp = Taxpayer(address(spouse));
77     require(
78         sp.getSpouse() == address(this) && sp.getIsMarried(),
79         "That person isn't married with you"
80     );
81     require(
82         (getTaxAllowance() == DEFAULT_ALLOWANCE && age < 65) ||
83         ((getTaxAllowance() == ALLOWANCE_OAP && age >= 65) &&
84         (sp.getTaxAllowance() == DEFAULT_ALLOWANCE && sp.getAge() < 65)) ||
85         (sp.getTaxAllowance() == ALLOWANCE_OAP && sp.getAge() >= 65),
86         "Before divorcing, fix your tax pool allowance"
87     );
88     /**
89     if(getAge()>=65) {setTaxAllowance(ALLOWANCE_OAP);}
90     else {setTaxAllowance(DEFAULT_ALLOWANCE);}
91     */
92     sp.setSpouse(address(0));
93     spouse = address(0);
94     /*
95     sp.divorce(); // instead of sp.setSpouse(address(0));
96     */
97     isMarried = false;
98 }

```

In real life scenario if we have previously taxpooled with the 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

7.4.4 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)

7.5 Echidna through CI/CD pipeline

Within the DevSecOps¹⁴ framework, where security is integrated from the early stages of the development cycle¹⁵, incorporating Echidna into the CI/CD¹⁶ pipeline allows for continuous assessment of contract security. This significantly reduces the risks associated with security vulnerabilities, as issues are detected and addressed before the code reaches the production environment.

To ensure effective implementation of Echidna within the CI/CD pipeline, it has been integrated with GitHub Actions¹⁷. This ensures that every code change undergoes automatic analysis by Echidna before release, ensuring that the released code is always secure and free from vulnerabilities. This practice provides an additional assurance of code consistency and integrity, contributing to maintaining a secure and reliable development and release environment. The page containing the complete history of triggered GitHub Action implemented¹⁸ is available on GitHub under the [Actions] section of the repository .

¹⁴DevSecOps (Development Security Operations) is a collaborative approach that integrates security practices throughout the entire software development process.

¹⁵The development cycle refers to the iterative process of software development, encompassing stages such as planning, coding, testing, and deployment.

¹⁶CI/CD (Continuous Integration/Continuous Deployment) is a software development practice that involves continuous integration of code changes, automated testing, and seamless deployment to production, ensuring a streamlined and efficient development pipeline.

¹⁷GitHub Actions are automated workflows within the GitHub platform that allow developers to define, customize, and automate software development workflows.

¹⁸github.com/Owanesh/SSA2324/blob/main/.github/workflows/main.yml