SSA Project

Michele Spina

January 2024

Abstract

This document presents an analysis of the security aspects of a given Taxpayer.sol contract, employing the Echidna tool for testing and evaluation. The focus is on enhancing the code's robustness, specifically by utilizing require() functions exclusively. This approach is chosen over other methods, such as incorporating invariants or direct code modifications, due to the greater flexibility offered by require() functions. This report is a component of the 'Security in Software Application' course at La Sapienza University of Rome and is designed as a laboratory exercise report rather than a scientific research paper.

Contents

1 Introduction

1.1 Fuzz Testing

Fuzz testing, more informally known as fuzzing, stands as a dynamic and automated software testing method designed to expose bugs, security vulnerabilities, and potential crashes in software applications. At its core, fuzz testing is about introducing a program to a range of invalid, unexpected, or random data inputs, and then meticulously observing the program's response to these unorthodox inputs. The overarching objective of this approach is to pinpoint weaknesses or flaws in the software that might be susceptible to exploitation by malicious entities or that could result in unintended behaviors when the software is deployed in real-life situations.

The process of fuzz testing is not just a random exercise; it is a sophisticated, methodical approach that employs various types of data inputs to rigorously test the software's boundaries and error-handling capabilities. These inputs can range from completely random bytes to modified valid inputs, or even specially crafted inputs designed to trigger specific behaviors in the software. By subjecting the software to such a diverse array of inputs, fuzz testing can effectively uncover a wide spectrum of issues, including memory leaks, buffer overflows, unhandled exceptions, and more.

One of the key strengths of fuzz testing is its ability to simulate a hacker's perspective, probing the software for vulnerabilities in much the same way a cyber attacker would. This proactive approach to security helps in identifying and rectifying vulnerabilities before the software is exposed to real-world threats. Additionally, fuzz testing can be instrumental in verifying the robustness and reliability of the software, ensuring that it can gracefully handle unexpected or erroneous inputs without crashing or behaving unpredictably.

1.2 Echidna

In the realm of smart contract fuzz testing Echidna stands out as a significant player. Developed in Haskell, Echidna is tailored for fuzzing and property-based testing of Ethereum smart contracts. It executes sophisticated fuzzing campaigns that leverage the contract's ABI (Application Binary Interface) to challenge user-defined predicates or Solidity assertions. A unique aspect of Echidna is its integration with additional tools, such as Slither, enhancing its capabilities beyond conventional fuzzing software. Echidna distinguishes itself with its 'property-based fuzzing' approach. Unlike traditional fuzzers that primarily hunt for crashes, Echidna focuses on invalidating user-defined invariants or properties. This method is particularly adept at uncovering subtle vulnerabilities that may elude other testing methodologies. This report zeroes in on applying fuzz testing to the Taxpayer.sol contract, an Ethereum blockchain smart contract. Integral to this contract are various require() statements, which are essential conditions for the contract's proper execution. These conditions form the basis of the properties Echidna will endeavor to invalidate during the fuzz testing procedure.

The objective of this report is to assess the capability of Echidna in detecting potential vulnerabilities within the Taxpayer.sol contract. By scrutinizing Echidna's performance across a range of edge cases and unforeseen behaviors, this analysis aims to contribute to the ongoing conversation about bolstering smart contract security.

1.3 Echidna Contract

1.3.1 Echidna Function: Using the echidna functions, characterized by the fact that their name must begin `echidna_` as in the example, define some properties of the code to be tested. In the following example, the `echidna_check_balance` function checks the value of the balance variable, representing the property for which the value must always be greater than 0. If a different case

occurred, the test would fail, and the transaction sequence would be shown on the screen which invalidated the property described.

```
function echidna_check_balance() public returns (bool) {
    return(balance >= 20);
}
```

1.3.2 Echidna test contract: The smartcontract used for the echidna test, in addition to containing the echidna functions described throughout this report, is an extension of the analyzed contract, which means that it has all its characteristics: methods and variables. In particular, the Taxpayer used in the test is a Taxpayer which has the addresses `0x0` and `0x0` as parents. It is in fact necessary to define which parents are as required by the constructor.

```
pragma solidity ^0.8.22;

import "./Taxpayer.sol";

contract TaxpayerTesting is Taxpayer(address(0), address(0)) {
    constructor() {
    }
}
```

In particular, for practical reasons, an alpha variable has been added to the testing contract, which is also a taxpayer instance with the same parents.

```
pragma solidity ^0.8.22;

import "./Taxpayer.sol";

contract TaxpayerTesting is Taxpayer(address(0), address(0)) {
   Taxpayer alpha;

    constructor() {
        alpha = new Taxpayer(address(0), address(0));
    }
}
```

2 Part 1

2.1 Requirements

The contract should require the following property, easy to overlook: if person x is married to person y, then person y should of course also be married and to person x. In other words, we can say that marriage should be a reflective relationship.
To this requirement, both for reasons useful for modeling and by analyzing the context„we can add these extra requirements:

- A person is married if and only if he or she is in a marriage property relationship with another person.
- No one can be married with itself
- A person under the age of 18 cannot be married

2.2 Contract Analysis

It is possible to observe from the code snippets present, how the marriage relationship, and the property “being married”, are represented in the smartcontract through the use of two variables:

`spouse` and `isMarried`. The boolean `isMarried` variable must be setted to true, if and only if the person associated to the contract is married, and in this case the `spouse` variable will contain the address of the smartcontract associated with the husband or wife, otherwise the associated address will be `0x0`. These two variables are initialized to `false` and `0x0` by the constructor, so the contract is initially in the "unmarried" state.

```
contract Taxpayer {
 uint age;
 bool isMarried;
 /* Reference to spouse if person is married, address(0) otherwise */
 address spouse;
...
 constructor(address p1, address p2) {
   age = 0;
   isMarried = false;
    ...
   spouse = address(0);
   ...
 }
 ...
 }
```

Within the contract there is the marry function which, taking an address as input, has the task of setting the married status, and associating the address given in input as spouse.

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

The divorce function, on the other hand, has the task of passing to the unmarried state.

```
 function divorce() public {
  Taxpayer sp = Taxpayer(address(spouse));
  sp.setSpouse(address(0));
  spouse = address(0);
  isMarried = false;
 }
```

In addition to the `getSpouse` function, the `getIsMarried` function was also implemented to observe the status of the contract.

```
function getSpouse() public view returns (address) {
    return spouse;
}

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

2.3 Echidna property functions

The first function, `echidna_marry`, has the goal to verify that a person is married if and only if he or she is in a marriage property relationship with another person. this means, and is transcribed in the following code, that the `isMarried` variable is true, if and only if, an address other than `0x0` is associated with the `spouse` variable.

```
function echidna_marry() public view returns (bool) {
    return ((getSpouse() != address(0) && getIsMarried())||
        getSpouse() == address(0) && !getIsMarried());
}
```

Using the `echidna_spouse_is_different`function, however, the requirement that no one can marry himself was verified.

```
function echidna_spouse_is_different() public view returns (bool) {
    return getSpouse() != address(this);
}
```

The purpose of the `echidna_marry_monogamy_and_reflexive_relation` function is to verify the reflexivity of the marital relationship. Since this relationship is binary, a control has been added on the taxpayer alpha, and a function to force the marriage between them, so as to verify that the analyzed properties are always verified whether the calls are made to the analyzed contract or one related to it.

```
function echidna_marry_monogamy_and_reflexive_relation() public view returns (bool) {
    bool self_test = true;
    bool alpha_test = true;
    if (getIsMarried()) {
        Taxpayer spouse =Taxpayer(getSpouse());
        self_test = Taxpayer(spouse).getSpouse() == address(this) &&
            Taxpayer(spouse).getIsMarried();
    }

    if (alpha.getIsMarried()) {
        Taxpayer spouse =Taxpayer(alpha.getSpouse());
        alpha_test = Taxpayer(spouse).getSpouse() == address(alpha) &&
            Taxpayer(spouse).getIsMarried();
    }
    return self_test && alpha_test;
}

function marry_alpha() public {
    marry(alpha);
}
```

The `echidna_marry_age_limits` function aims to verify that anyone involved in a marital relationship is at least 18 years old. To facilitate these checks by echidna, functions have been inserted with the aim of reducing the number of making a significant increase in age (15 years) with a single transaction. These functions will also be useful for what will be the third part of this report.

```
function echidna_marry_age_limits() public view returns (bool) {
    if (getIsMarried()) {
        return getAge() > 17 && Taxpayer(getSpouse()).getAge() >= 17;
    }
    return true;
}

function getting_old(Taxpayer test_tp) public {
    for (uint i = 0; i < 15; i++){
        test_tp.haveBirthday();
    }
}

function im_getting_old() public {
    for (uint i = 0; i < 15; i++){
        haveBirthday();
    }
}
```

In the end, a check of the correct functioning of the divorce function has been added, according to the logic described by the prerequisites.

```
function echidna_divorce() public  returns (bool) {
    if (getIsMarried()) {
```

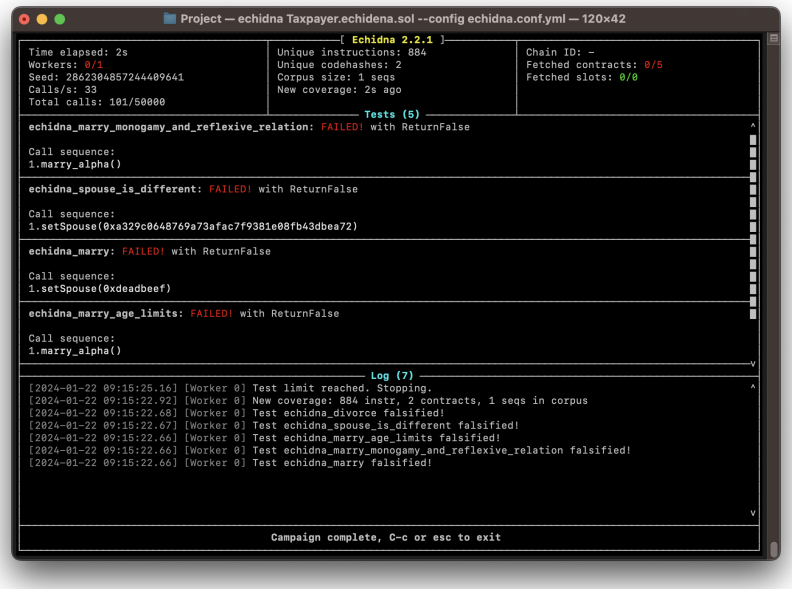

Figure 1: Test result of the original code

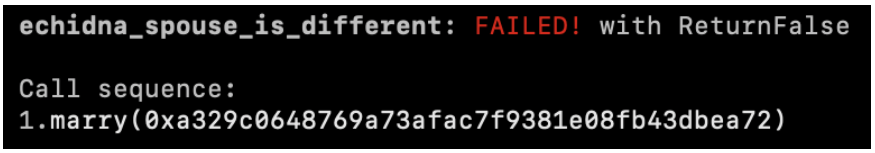

Figure 2: A call sequence that anable to spouse with itself

```
        Taxpayer spouse = Taxpayer(getSpouse());
        divorce();
        return Taxpayer(spouse).getSpouse() == address(0) &&
            !Taxpayer(spouse).getIsMarried() &&
            getSpouse() == address(0) &&
            !getIsMarried();
    }
    return true;
}
```

2.4 Test results and Solution

In figure 1 it's possible to observe the result of the fuzzing tests. As can also be understood from a statistical analysis, to pass the first two tests it was necessary to retouch the code by inserting require on the address given in input, admitting only an address other than zero and different from that of the contract itself.

The reflexivity of the relationship instead required the addition of recursion in the marry and divorce functions, and as regards the former also a check to avoid marrying an already married person by overwriting his marital status and breaking the reflexivity of his relationship. In figure 3 it's possible to observe a call sequence breaking the property. In particular, the setSpouse function turns out to be insecure, especially if not with adequate access control, for this reason it was removed.

```
echidna_marry_monogamy_and_reflexive_relation: FAILED! with ReturnFalse

Call sequence:
1.marry_alpha()
2.marry(0xee35211c4d9126d520bbfeaf3cfee5fe7b86f221)
```

Figure 3: A call sequence to fail echidna marry monogamy and reflexive relation test

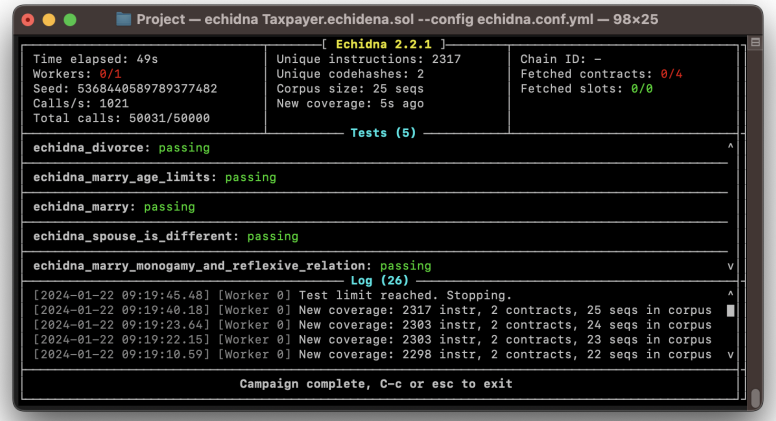

Figure 4: Test results of part 1 solution

In conclusion, due to the added age restrictions it was necessary to add a check on this value. The solution proposed at the end of this step is then presented.

```
function marry(Taxpayer new_spouse_tp) public {
    address new_spouse = address(new_spouse_tp);
    require(new_spouse != address(0), "You cannot spose address 0x0");
    require(new_spouse != address(this), "You cannot marry with yourself");
    require(!getIsMarried(), "You are already married, be faithful");
    require(age > 17, "You have to be 18 years old or more to marry");
    spouse = new_spouse;
    isMarried = true;
    if (new_spouse_tp.getSpouse() != address(this)){
        new_spouse_tp.marry(this);
    }
    return;
}

function divorce() public {
    require(isMarried, "You have to be married to divorce");
    Taxpayer sp = Taxpayer(address(spouse));
    spouse = address(0);
    isMarried = false;
    if (sp.getIsMarried()) {
        sp.divorce();
    }
}
```

3 Part 2

3.1 Requirements

The tax system uses an income tax allowance (deduction):

1. Every person has an income tax allowance on which no tax is paid. There is a default tax allowance of 5000 per individual, and only income above this amount is taxed.

2. Married persons can pool their tax allowance as long as the sum of their tax allowances remains the same. For example, the wife could have a tax allowance of 9000, and the husband a tax allowance of 1000. Add invariants that express these constraints, and, if necessary, fix/improve the code to ensure that they are not violated.

3.2 Contract analysis

The variable `tax_allowance` is already present within the contract, which represents the tolerance value that we want to model with the code, and is initialized to 5000. Furthermore, the `transferAllowance` function has the task of fulfilling requirement 2. We can observe that both the input data to `transferAllowance` and the `tax_allowance` variable have a uint type, therefore they cannot take on a negative value.

```
 /* Constant default income tax allowance */
 uint constant  DEFAULT_ALLOWANCE = 5000;
...
 /* Income tax allowance */
 uint tax_allowance;

constructor(address p1, address p2) {
...
   tax_allowance = DEFAULT_ALLOWANCE;
 }
 ...
 /* Transfer part of tax allowance to own spouse */
 function transferAllowance(uint change) public {
  tax_allowance = tax_allowance - change;
  Taxpayer sp = Taxpayer(address(spouse));
  sp.setTaxAllowance(sp.getTaxAllowance()+change);
 }
...
function setTaxAllowance(uint ta) public {
    tax_allowance = ta;
}
```

3.3 Echidna property function

In order to verify these requirements, a single test function has been implemented, `echidna_cumulative_tax_allowance`, which carries out a check on the value of `tax_allowance`, verifying that for those who are not married it is always 5000, and vice versa for those who are, that the sum of the `tax_allowance` of the two spouses is 10,000.

```
function echidna_cumulative_tax_allowance() public view returns (bool){
    if (getIsMarried()){
        Taxpayer spouse = Taxpayer(getSpouse());
        return (getTaxAllowance() + spouse.getTaxAllowance() == 10000);
    }
    else {
        return (getTaxAllowance() == 5000);
    }
}
```

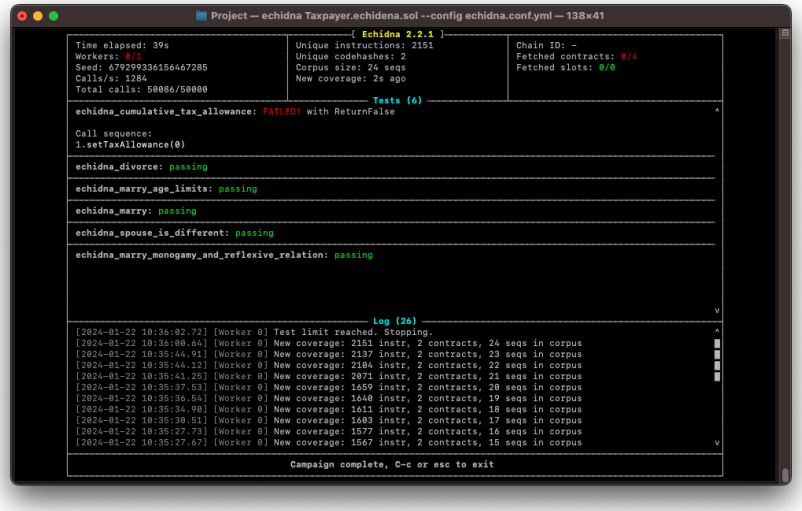

Figure 5: Part 2 test result of part 1 solution

It is important to note that the reliability of this would be undermined if one or more of the previously proposed tests were invalidated.

3.4 Test results and Solution

In the testing phase it was possible to observe how the `setTaxAllowance` function generates a fragility in the code if it lacks the necessary access control, figure 5, and in the following snipped is proposed the solution of this issue.

```
function setTaxAllowance(uint ta) public {
    require(msg.sender == spouse);
    tax_allowance = ta;
}
```

As the tests continued, the need to update the divorce() function arose, see figure 6, to restore the `default` values of `tax_allowance` in case the call is successful.

```
function divorce() public {
    ...
    tax_allowance = 5000
    if (sp.getIsMarried()) {
        sp.divorce();
    }
}
```

4 Part 3

4.1 Requirements

The new government introduced a measure that people aged 65 and over have a higher tax allowance, of 7000. The rules for pooling tax allowances remain the same.

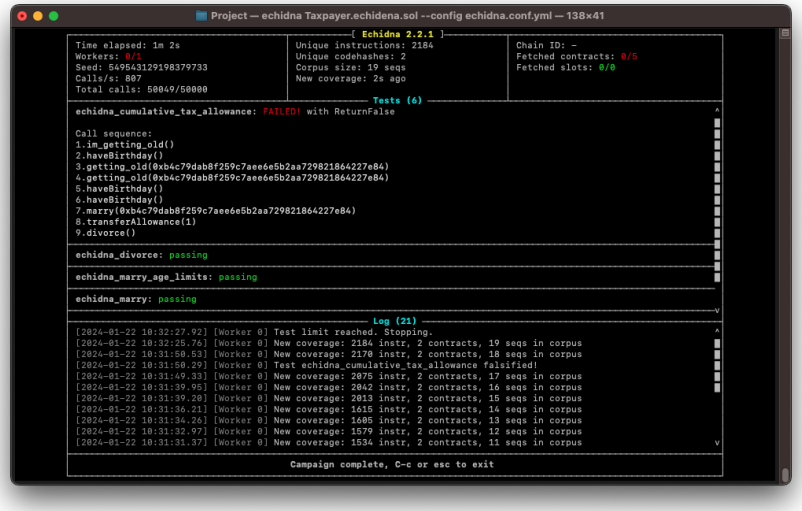

Figure 6: Part 2 test results after introduction of access control in `setAllowance` function

4.2 Contract Analysis

In this case the contract does not present any implementation of this feature, other than the definition of a constant `ALLOWANCE_OAP` with the associated tax allowance value for those over 65.

```
/* Constant income tax allowance for Older Taxpayers over 65 */
uint constant ALLOWANCE_OAP = 7000;
```

However, the `getBaseTaxAllowance` function has been introduced, defined as public and view, therefore its direct call does not modify the state of the contract and therefore does not alter the test results, but is useful for obtaining what should be the default `tax_allowance` value of the contract.

```
function getBaseTaxAllowance() public view returns (uint) {
    if (age < 65) {
        return DEFAULT_ALLOWANCE;
    }
    else {
        return ALLOWANCE_OAP;
    }
}
```

4.3 Echidna property functions

It was necessary to update the test function `echidna_cumulative_tax_allowance` using the new function introduced.

```
function echidna_cumulative_tax_allowance() public view returns (bool){
    if (getIsMarried()){
        Taxpayer spouse = Taxpayer(getSpouse());
        return (getTaxAllowance() +
            spouse.getTaxAllowance() == getBaseTaxAllowance() +
            spouse.getBaseTaxAllowance()
            );
    }
    else {
        return (getTaxAllowance() == getBaseTaxAllowance());
```

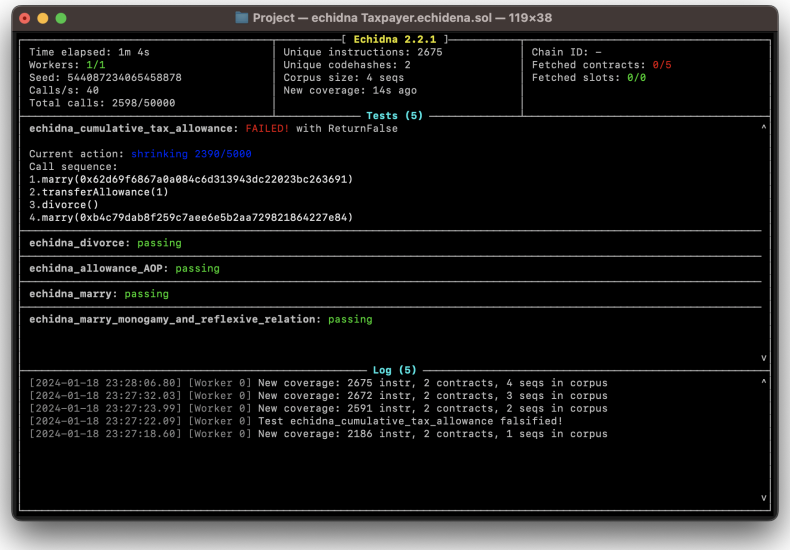

Figure 7: A simplified Echidna comulative tax allowance fail, without marry age requirements

```
    }
}
```

Finally, it was necessary to introduce a test to verify the correct functioning of the `getBaseTaxAllowance` function, by the function `echidna_allowance_AOP`

```
function echidna_allowance_AOP() public view returns (bool) {
    if (getAge() < 65 && getBaseTaxAllowance() != 5000){
        return false;
    }
    if (getAge() >= 65 && getBaseTaxAllowance() != 7000){
        return false;
    }
    return true;
}
```

4.4 Test results and Solution

The code proposed so far does not present any update of the tax allowance value with increasing age, for this reason echidna has detected a call sequence capable of breaking the properties verified by the functions just defined. In figure 8 it is possible to observe how the simple increase via the `haveBirthday` function invalidated the test, remember that the `im_getting_old` function does nothing but call the `haveBirthday` function 15 times.Following the update of the `haveBirthday` function, it was sufficient to add a successful call of the `marry` function and the `divorce` function to the aging (Figure 9), similarly to what was highlighted in the previous section.

With the update shown below of the `haveBirthday` and `divorce` functions it was possible to update the value of `tax_allowance` and restore the correct value following a divorce.

```
function haveBirthday() public {
    age++;
    if (age == 65) {
        tax_allowance += ALLOWANCE_OAP - DEFAULT_ALLOWANCE;
```

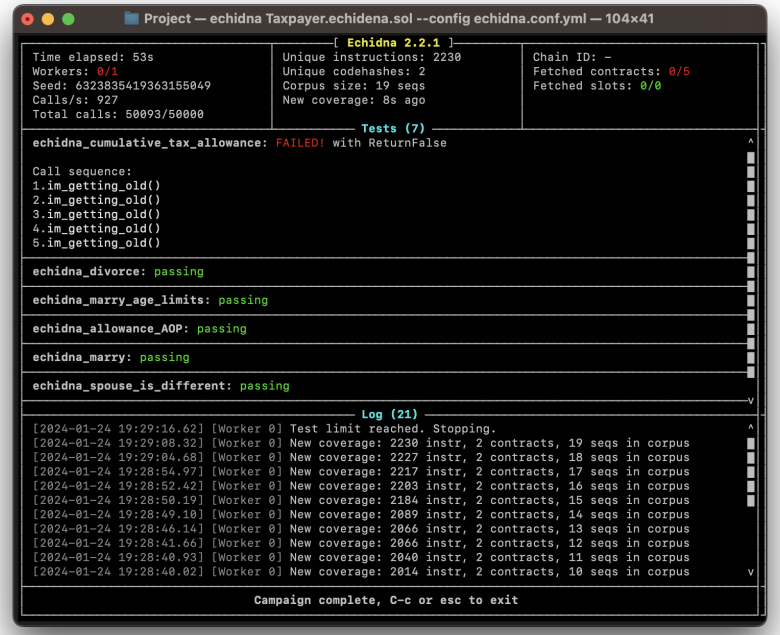

Figure 8: `echidna_cumulative_tax_allowance` fail

```
    }
}
```

```
function divorce() public {
    require(isMarried, "You have to be married to divorce");
    Taxpayer sp = Taxpayer(address(spouse));
    spouse = address(0);
    isMarried = false;

    tax_allowance = getBaseTaxAllowance();
    //tax_allowance = 5000
    if (sp.getIsMarried()) {
        sp.divorce();
    }
}
```

5 Conclusion and Future Works

The comprehensive analysis undertaken in this project led not only to the verification of the required properties stipulated by the initial requirements but also to the introduction of an additional constraint regarding the suitable age for marriage. As illustrated in Figure 10 , the Echidna-based test functions, when applied to the smart contract in its final form, did not invalidate these properties. This successful validation signifies the effectiveness of the Echidna tool in ensuring that the contract adheres to the predefined conditions.

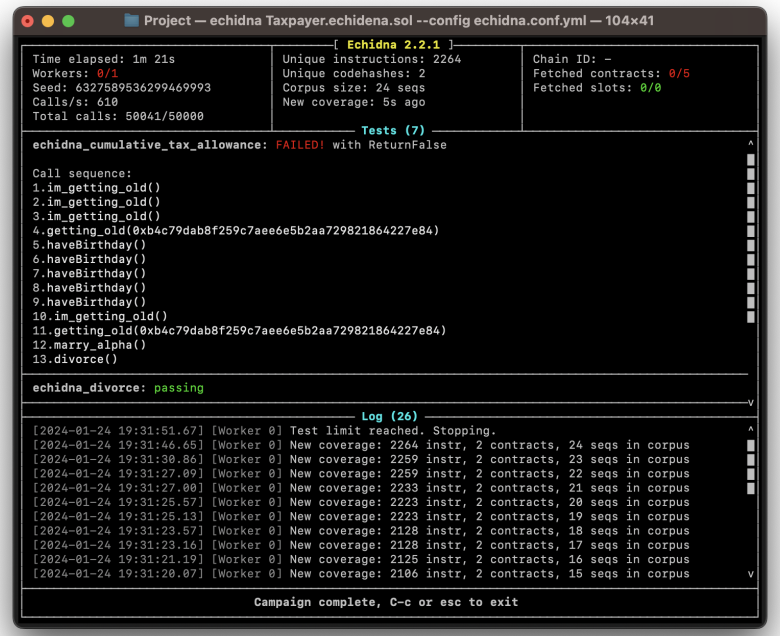

Figure 9: `echidna_cumulative_tax_allowance` fail after first updating

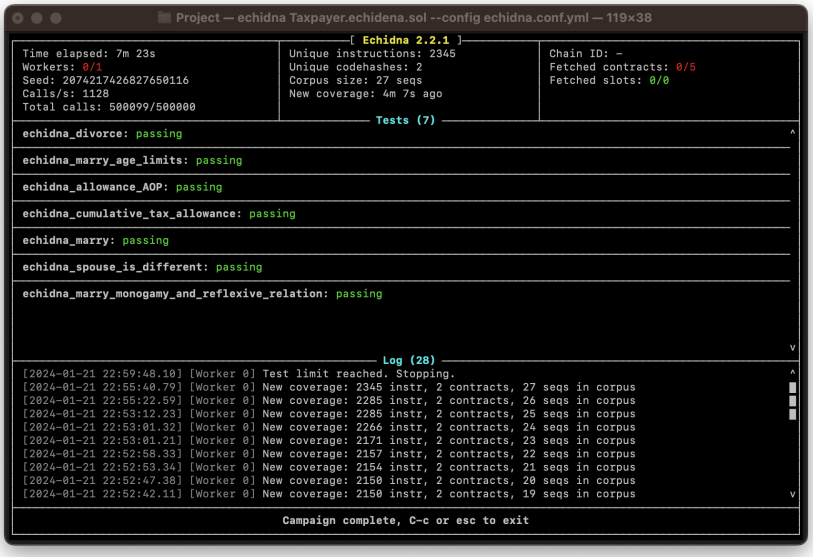

Figure 10: Final test result

However, the scope of our analysis suggests that there is room for a more thorough examination, particularly in the realms of dynamic and static analysis. Such an exploration could focus on properties related to the kinship of various individuals within the smart contract context. This aspect, while not the primary focus of our current analysis, presents an intriguing avenue for further research and could unveil additional insights into the smart contract's behavior and integrity.

Moreover, throughout the process of this project and the composition of this report, it became evident that there is a significant need for the implementation of an access control system within the smart contract. The initial proposal for this system is quite straightforward – limiting the access of various contract functions exclusively to the contract owner. However, this concept could be expanded upon. The extended version of this access control system would involve granting special permissions to other exceptional entities or figures, adding an extra layer of security and functional specificity to the contract.

This proposed expansion of the access control mechanism would not only enhance the security of the smart contract but also provide a more granular level of control over its operations. Testing and refining this system could further ensure that only authorized and intended interactions occur, safeguarding against potential misuse or exploitation.

Added to these is the implementation of the use of variable income, which represents the user's income and is so far unused in the contract. A use of this variable with related functions to set it and apply the tax deduction would lead to the need to further investigate the security of the contract. In summary, while the current project has successfully validated the key properties of the smart contract using Echidna, there remains significant potential for further enhancement and exploration. Delving deeper into dynamic and static analysis, particularly concerning kinship properties, and the development of a more sophisticated access control system, represent promising directions for future work. These efforts will contribute significantly to the ongoing endeavor to improve the security and functionality of smart contracts in the ever-evolving blockchain landscape.