# 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.

## 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 program designed for fuzzing/property-based testing of Ethereum smart contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI[1] 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()`[2] statements, which are conditions that must be met for the

[1] 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

[2] 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.

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[3] 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
```

```
==== 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
```

[3] Already included into Echidna stack

```
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 (`github.com/owanesh/SSA2324/blob/master/report/mythril.report`). The findings highlight several areas of concern, including possible risks related to reentrancy[4] (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.[5] Another method is by inheritance

TaxpayerTesting.sol

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

import "./Taxpayer.sol";

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

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

[5]https://github.com/crytic/echidna/wiki/How-to-use-Echidna-with-multiple-contract

```
    Taxpayer bravo;

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

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 `new_spouse` 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 your status is already set to `isMarried=True`
- 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 not married with another person"
    );
    spouse = newSpouse;
    isMarried = true;
}
```

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

# 6 The new government introduced a measure that people aged 65 and over have a higher tax allowance, of 7000

# 7 Extra requirements

Some extra requirements are added to make the contract more "real". 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

```
    constructor(address p1, address p2) {
        require(
            (p1 == address(0) && p2 == address(0)) ||
                (Taxpayer(p1).getSpouse() == p2 &&
                    Taxpayer(p2).getSpouse() == p1),
            "A new born is allowed only form init and married couple"
        );
        age = 0;
        isMarried = false;
        parent1 = p1;
        parent2 = p2;
        spouse = (address(0));
        income = 0;
        tax_allowance = DEFAULT_ALLOWANCE;
    }
```

# 8 Conclusion

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.