

Echidna and fuzzing

Fuzzing is a software testing technique based on dynamic analysis that involves providing random or semi-random inputs to a program to discover bugs, vulnerabilities, or unexpected behavior.

In this homework I used **Echidna**, a smart contract fuzzer designed for Ethereum. It is a property-based fuzzer, which in contrast to a classic fuzzer (like AFL++) that tries to find crashes, Echidna aims to break user-defined invariants, typically written in Solidity using **assert**, **require**, or with custom functions.

It performs fuzzing by generating random transactions and inputs for smart contracts, monitors the execution and checks whether these inputs cause the contract to violate the specified properties.

Mythril

Before running Echidna, in some web3 CTF challenges I always like to run *Mythril*, a security analysis tool which leverages symbolic execution to find vulnerabilities in EVM bytecode. This tool is used as a guide to correct the code (as if it was necessary in this simple case).

By calling **myth analyze Person.sol.old**, the tool complains about the fact that there are some instances of SWC ID 107, so a reentrancy vulnerability in the function **divorce()**:

```
== State access after external call ==
SWC ID: 107
Severity: Medium
Contract: Person
Function name: divorce()
PC address: 643
Estimated Gas Usage: 14521 - 89982
Write to persistent state following external call
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.
```

This is because the developer accesses to **isMarried** and **spouse** after doing an external call. The approach I used to fix these issues is simply to work with the state before calling external functions.

Part 1: marriage reciprocity

In the **Person.sol.old** code I have the following code for the **marry(address new_spouse)** function (if line numbers do not match with the code in the Google Drive is just because of indentation and blank lines to improve readability):

```
27      //We require new_spouse != address(0);
28      function marry(address new_spouse) public {
29          spouse = new_spouse;
30          isMarried = true;
31      }
```

The above code changes the state for a Person contract, but the “new spouse” will keep its current state, unacceptable for notarial operations. The first thing that came up to my mind was to use **Person sp = Person(address(new_spouse))** and call the function **sp.setSpouse(address(this))**, similar to the code in the original **divorce()** function after fixing the reentrancy. But while writing some Echidna

test functions (presented later) I figured out that the **setSpouse()** function was a great way to duplicate code: if I have to add invariants and properties that cannot be violated when calling **marry()** and **divorce()**, then I MUST rewrite those conditions also for the **setSpouse()** function. Plus, what is the difference between **marry()** and **setSpouse()**? **marry()** also changes the state of **isMarried**, but to me this is irrelevant because I could also retrieve that info by using **getSpouse()**. For these reasons I decided to get rid of the **setSpouse()** function, but to keep **isMarried** because is more readable than only checking the **getSpouse()** returned address.

So, the edited code is the following:

```
44     spouse = new_spouse;
45     isMarried = true;
46     if(Person(address(new_spouse)).getSpouse() != address(this))
47         Person(address(new_spouse)).marry(address(this));
```

This way I have the **marriage reciprocity**, so if **Alice** calls **marry(Bob)**, then **Bob** will also be married with **Alice**.

This property is satisfied only **temporarily**, because nothing stops **Alice** from calling **marry(Charlie)**, changing its spouse and leaving **Bob** in an inconsistent state because it should be married with **Alice**, but **Alice** is married with **Charlie** now. Again, unacceptable.

For this reason I MUST add some constraints that prevent the above from happening:

```
27     function marry(address new_spouse) public {
28
29         require(
30             spouse == address(0) && !isMarried,
31             "You are already married"
32         );
33
34         // ...
35
36         // ...
37
38         require(new_spouse != spouse, "You are already married to this person");
39         require(
40             Person(address(new_spouse)).getSpouse() == address(0) ||
41             Person(address(new_spouse)).getSpouse() == address(this),
42             "New spouse is already married with another person"
43         );
44         spouse = new_spouse;
45         isMarried = true;
46         if(Person(address(new_spouse)).getSpouse() != address(this))
47             Person(address(new_spouse)).marry(address(this));
48     }
```

Now everything works, I guarantee the **marriage reciprocity**.

Let's have a look at the **divorce()** function.

The **divorce()** function is much simpler, other than removing the **setSpouse()** function, I just added a simple and logical check that allows you to proceed if and only if you are married:



The image shows two versions of the `divorce()` function side-by-side. The top version, labeled 'OLD' in a red box, is a simple function that sets the spouse to `address(0)` and sets `isMarried` to `false`. The bottom version, labeled 'NEW' in a green box, includes a `require` statement to check if the caller is married and the spouse is not `address(0)`. It also calls `sp.divorce()` on the spouse's address.

```

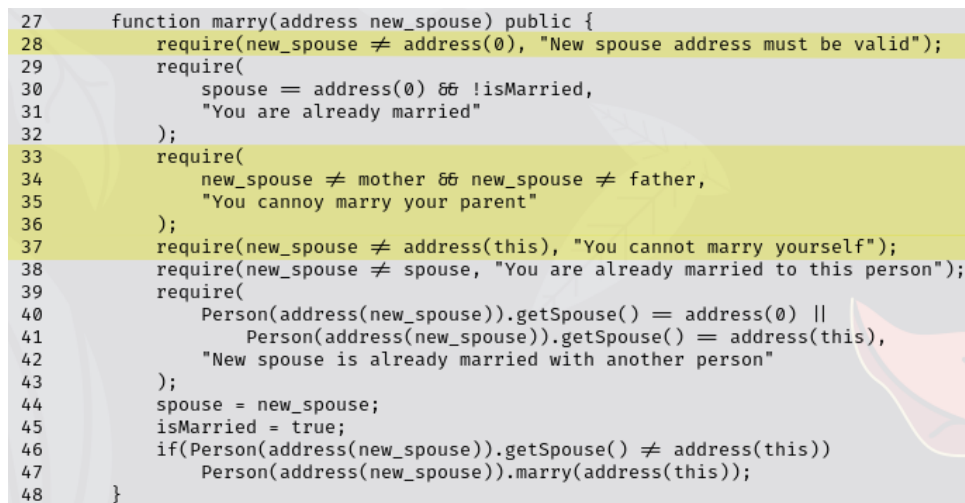
33  function divorce() public {
34      Person sp = Person(address(spouse));
35      sp.setSpouse(address(0));
36      isMarried = false;
37      spouse = address(0);
38  }

50  function divorce() public {
51      require(isMarried && spouse != address(0), "You are not married");
52
53      Person sp = Person(spouse);
54
55      spouse = address(0);
56      isMarried = false;
57      if(sp.getSpouse() == address(this))
58          sp.divorce();
59  }

```

If I left the previous code, I would have **Alice** which calls **marry(Bob)**, becoming mutually married, but then if **Alice** called **divorce()**, then **Bob** would have **address(0)** as **spouse** and **isMarried** to **true**! With the edited code, when **Alice** calls **divorce()**, then **Bob** will also get the divorce and the **isMarried** state will be reset.

Going back to the **marry()** function, I also added some more input validation and other nice to have properties:



The image shows the `marry()` function with several `require` statements for input validation. It checks if the new spouse is `address(0)`, if the caller is already married, if the new spouse is a parent, if the new spouse is the caller, if the new spouse is the current spouse, and if the new spouse is already married to someone else.

```

27  function marry(address new_spouse) public {
28      require(new_spouse != address(0), "New spouse address must be valid");
29      require(
30          spouse == address(0) && !isMarried,
31          "You are already married"
32      );
33      require(
34          new_spouse != mother && new_spouse != father,
35          "You cannot marry your parent"
36      );
37      require(new_spouse != address(this), "You cannot marry yourself");
38      require(new_spouse != spouse, "You are already married to this person");
39      require(
40          Person(address(new_spouse)).getSpouse() == address(0) ||
41          Person(address(new_spouse)).getSpouse() == address(this),
42          "New spouse is already married with another person"
43      );
44      spouse = new_spouse;
45      isMarried = true;
46      if(Person(address(new_spouse)).getSpouse() != address(this))
47          Person(address(new_spouse)).marry(address(this));
48  }

```



Testing the code

After editing the code I ran Mythril once more to check if I introduced some vulnerabilities, here is the output:

```
root@76df6f8191d0:/code# myth analyze Person.sol
The analysis was completed successfully. No issues were detected.
```

I used Echidna heavily during this homework, especially when I removed the **setSpouse()** function and after adding each property. A quick note on the setup, I used a the **trailofbits/eth-security-toolbox** docker image, I ran **solc-select install 0.8.22**, **solc-select use 0.8.22** and I was ready to go.

This is the **PersonTests.sol** contract that was used to test the properties:

```
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.22;
3
4  import "./Person.sol";
5
6  contract PersonTests is Person {
7      Person alice;
8      Person parent;
9      Person bob;
10     Person charlie;
11
12     constructor() Person(address(0), address(0)) {
13         parent = new Person(address(0), address(0));
14         alice = new Person(address(parent), address(parent));
15         bob = new Person(address(0), address(0));
16         charlie = new Person(address(0), address(0));
17
18         alice.marry(address(bob));
19     }
20 }
```

I'm using the inheritance approach (*line 4-6*), so the testing has access to all the **Person** functions. To me, this seems to be the most reasonable way to perform the tests, also thinking to bigger and more complex contracts. In the constructor (*line 12-19*) I create 4 person, **Alice** is the only one having a real Person as parent and the only one who starts by calling **marry(address(bob))**.

The next two functions are used to test the **marriage** and **divorce reciprocity properties**:

```
21  function echidna_check_marriage() public returns (bool) {
22      return (alice.getSpouse() == address(bob)
23             && bob.getSpouse() == address(alice));
24  }
25
26  function echidna_divorce() public returns (bool) {
27      try alice.marry(address(bob)){
28          alice.divorce();
29      } catch {
30          alice.divorce();
31      }
32      return (alice.getSpouse() == address(0)
33             && bob.getSpouse() == address(0));
34  }
```

The next function is used to test if you can marry yourself:

```
36     function echidna_marry_yourself() public returns (bool) {
37         try alice.divorce() { /* NOP */ } catch { /* NOP */ }
38         try alice.marry(address(alice)) {
39             return false;
40         } catch {
41             return true;
42         }
43     }
```

The next function is used to test if you can marry your parent, this is why **Alice** is the only one having a real parent in these tests:

```
45     function echidna_marry_your_parent() public returns (bool) {
46         try alice.divorce() { /* NOP */ } catch { /* NOP */ }
47         try alice.marry(address(parent)) {
48             return false;
49         } catch {
50             return true;
51         }
52     }
```

This last function is used to test if you can marry a different Person while you are already married:

```
54     function echidna_ho_visto_lei_che_bacia_lui_che_bacia_lei_che_bacia_me() public returns (bool) {
55         // mon amour, amour, ma chi baci tu?
56         try alice.divorce() { /* NOP */ } catch { /* NOP */ }
57         alice.marry(address(bob));
58         try alice.marry(address(charlie)) {
59             return false;
60         } catch {
61             return true;
62         }
63     }
```

Results of the tests on the original code:

- **echidna_marry_your_parent**: *FAILED*
- **echidna_check_marriage**: *FAILED*
- **echidna_marry_yourself**: *FAILED*
- **echidna_ho_visto_lei_che_bacia_lui_che_bacia_lei_che_bacia_me**: *FAILED*
- **echidna_divorce**: *passing*

Results of the tests on the edited code:

- **echidna_ho_visto_lei_che_bacia_lui_che_bacia_lei_che_bacia_me**: *passing*
- **echidna_divorce**: *passing*
- **echidna_marry_yourself**: *passing*
- **echidna_check_marriage**: *passing*
- **echidna_marry_your_parent**: *passing*

Part 2: fair state subsidy

For this part, I added some simple code to manage the events that trigger a change in the state subsidy of a person:

- When the person marries someone
- When the person divorces
- When the person reaches 65 years old and is not married

I introduced the following three constants:

```
4  contract Person {
5      uint age;
6      bool isMarried;
7
8      /* Reference to spouse if person is married, address(0) otherwise */
9      address spouse;
10     address mother;
11     address father;
12
13     uint constant DEFAULT_SUBSIDY = 500;
14     uint constant DEFAULT_MARRIED_SUBSIDY = 350; /* 500 - 30% */
15     uint constant DEFAULT_065_UNMARRIED_SUBSIDY = 600;
16
17     /* Welfare subsidy */
18     uint state_subsidy;
```

Figure 1: Default subsidy - 30% already computed for simplicity

Obviously, in the constructor for the **Person** I assign the default subsidy of 500. The new **marry()** function is the following:

```
29     function marry(address new_spouse) public {
30         require(new_spouse != address(0), "New spouse address must be valid");
31         require(
32             spouse == address(0) && !isMarried,
33             "You are already married"
34         );
35         require(
36             new_spouse != mother && new_spouse != father,
37             "You cannot marry your parent"
38         );
39         require(new_spouse != address(this), "You cannot marry yourself");
40         require(new_spouse != spouse, "You are already married to this person");
41         require(
42             Person(address(new_spouse)).getSpouse() == address(0) ||
43             Person(address(new_spouse)).getSpouse() == address(this),
44             "New spouse is already married with another person"
45         );
46         spouse = new_spouse;
47         isMarried = true;
48         state_subsidy = DEFAULT_MARRIED_SUBSIDY;
49         if(Person(address(new_spouse)).getSpouse() != address(this))
50             Person(address(new_spouse)).marry(address(this));
51     }
```

Thanks to the marriage reciprocity property already satisfied, the **state_subsidy** also changes for the spouse.

The same applies for the divorce, with the only difference that the age is checked to correctly set the right subsidy:

```
53     function divorce() public {
54         require(isMarried && spouse != address(0), "You are not married");
55
56         Person sp = Person(spouse);
57
58         spouse = address(0);
59         isMarried = false;
60         state_subsidy = DEFAULT_SUBSIDY;
61         if(age ≥ 65)
62             state_subsidy = DEFAULT_065_UNMARRIED_SUBSIDY;
63         if(sp.getSpouse() == address(this))
64             sp.divorce();
65     }
```

With regards to **haveBirthday()**, since by default a **Person** has a state subsidy of 500, I only change the subsidy if and only if the person reaches 65 years old AND is not married:

```
67     function haveBirthday() public {
68         age++;
69         if(!isMarried && age ≥ 65)
70             state_subsidy = DEFAULT_065_UNMARRIED_SUBSIDY;
71     }
```

To verify the **fair state subsidy** property, I added the following function **getSubsidy()** with a **require** statement that checks all the possible correct combinations of subsidy, age and marriage status:

```
81     function getSubsidy() public view returns (uint) {
82         require(
83             (age < 65 && !isMarried && state_subsidy == DEFAULT_SUBSIDY) ||
84             (age ≥ 65 && !isMarried && state_subsidy == DEFAULT_065_UNMARRIED_SUBSIDY) ||
85             (isMarried && state_subsidy == DEFAULT_MARRIED_SUBSIDY),
86             "Subsidy fraud"
87         );
88
89         return state_subsidy;
90     }
```

continues in the last page...

In the end, in the **PersonTests.sol** constructor I added a **for** loop that calls **alice.haveBirthday()** so that it reaches an age ≥ 65 and then I added the following tests:

```
69     function echidna_young_subsidy() public returns (bool) {
70         try charlie.divorce() { /* NOP */ } catch { /* NOP */ }
71         return (charlie.getSubsidy() == DEFAULT_SUBSIDY);
72     }
73
74     function echidna_elder_subsidy() public returns (bool) {
75         try alice.divorce() { /* NOP */ } catch { /* NOP */ }
76         return (alice.getSubsidy() == DEFAULT_065_UNMARRIED_SUBSIDY);
77     }
78
79     function echidna_married_subsidy() public returns (bool) {
80         try alice.divorce() { /* NOP */ } catch { /* NOP */ }
81         try bob.divorce() { /* NOP */ } catch { /* NOP */ }
82         alice.marry(address(bob));
83         return (alice.getSubsidy() == DEFAULT_MARRIED_SUBSIDY &&
84                 bob.getSubsidy() == DEFAULT_MARRIED_SUBSIDY);
85     }
```

The last test results are:

```
[2025-03-31 12:45:37.55] Running slither on PersonTests.sol ... Done! (0.961079753s)
echidna_young_subsidy: passing
echidna_marry_your_parent: passing
echidna_check_marriage: passing
echidna_elder_subsidy: passing
echidna_marry_yourself: passing
echidna_married_subsidy: passing
echidna_divorce: passing
echidna_ho_visto_lei_che_bacia_lui_che_bacia_lei_che_bacia_me: passing
```