SAPIENZA
UNIVERSITÀ DI ROMA

Security in Software Applications Proj 3

Alex Parri - 2140961 - Master's Degree in Cybersecurity

A.Y. 2024/25

Abstract

This is the report for the **third project** of the Security in Software Applications course directed by Daniele Friolo for the Academic Year 24/25 for the Master's Degree in **Cybersecurity** at Sapienza University of Rome. In this homework, the goal was to experiment with **smart contracts** through **fuzz testing**, a form of software dynamic analysis.

Specifically, it was asked to use the Echidna tool to **test** a given smart contract with the goal of validating its properties through the use of **requires** and **invariants**.

The **hardware** utilized for testing is Ryzen 5800X 8-Core 16-Thread @ 4.850GHz CPU with Echidna 2.2.6 in Ubuntu 24.04.02 LTS x86_64 and 16GB of RAM.

Echidna

Fuzzing is a software testing technique in which a software is inputted with **random inputs** in order to find bugs, hangs, crashes in order to find and subsequently patch security flaws.

In this project it is done with **Echidna**, a Haskell tool designed for analyzing Ethereum smart contracts by invoking their publicly available functions to look for violations of Solidity **require** assertions or developer-defined **invariants**. The tool is particular in its own way as it uses grammar-based campaigns thus generating inputs tailored to the contract's **actual code**.

First part

The provided untouched smart contract code is the following

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

contract Person {
  uint age;
  bool isMarried;

  /* Reference to spouse if person is married, address(0) otherwise */
  address spouse;
  address mother;
  address father;

  /* welfare subsidy */
  uint constant  DEFAULT_SUBSIDY = 500;
  uint state_subsidy;

  constructor(address ma, address fa) {
    age = 0;
    isMarried = false;
    mother = ma;
    father = fa;
    spouse = address(0);
    state_subsidy = DEFAULT_SUBSIDY;
  }

  // We require new_spouse != address(0);
  function marry(address new_spouse) public {
    spouse = new_spouse;
    isMarried = true;
  }

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

  function haveBirthday() public {
    age++;
  }

  function setSpouse(address sp) public {
    spouse = sp;
  }

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

In this part the assignment was to **ensure** the following statement: *if person x is married to person y, then person y should of course also be married and to person x.*

Note: each country decides for its own laws, including marriage. In this context, the law of **Italy** was followed when considering constraints.

The following **changes** were applied, going from top to bottom of the contract:

Contract fields

```
uint internal age; // made internal

/* Reference to spouse if person is married, address(0) otherwise */
address internal spouse;  // made internal
address immutable mother; // made immutable
address immutable father; // made immutable
```

The age and spouse fields shall not be modifiable from outside the method, therefore they were made `internal` and the following **getter method** was introduced, while the `getSpouse()` method was made into `view`

```
  // added
  function getAge() public view returns (uint) {
    return age;
  }

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

Meanwhile, the `mother` and `father` fields were made `immutable` to ensure that they don't get accidently modified.

Constructor

The following `require` costraints have been added, as well as the `uint8 _age` field to avoid it being initialized to zero

```
// added uint8 _age
constructor(address ma, address fa, uint8 _age) {
  require(ma != address(0), "Please specify a non-zero mother address.");
  require(fa != address(0), "Please specify a non-zero father address.");
  require(ma != fa, "Mother and father must be different people.");
  require(age < 120, "Please provide an age lower than 120.");
  require(age >= 0, "Please provide an age higher or equal than 0.");

  age = _age;
  mother = ma;
  father = fa;
  spouse = address(0);
  isMarried = false;
  state_subsidy = DEFAULT_SUBSIDY;
}
```

Marrying

This was the most complicated method to fix due to the large amount of requirements needed to marry a significant other

```
// we require new_spouse != address(0);
function marry(address new_spouse) public {
  require(new_spouse != address(0), "Please specify a non-zero spouse.");
  require(new_spouse != address(this), "You cannot marry yourself.");
  require(new_spouse != mother, "You cannot marry your own mother.");
  require(new_spouse != father, "You cannot marry your own father.");
  require(spouse == address(0), "You are already married.");
  require(age >= 18, "You must be at least 18 years old to get married.");

  Person other = Person(new_spouse);
  require(other.getAge() >= 18, "Your spouse must be at least 18 years old
   ↪  to get married.");
  require(other.getSpouse() == address(0), "The other person is already
   ↪  married with someone else.");

  // actually marry
  spouse = new_spouse;
  other.acceptMarriage(address(this));
  isMarried = true;
}
```

Divorcing

```
function divorce() public {
  require(spouse != address(0), "You are not currently married.");

  // actually divorce
  address temp = spouse;
  spouse = address(0);
  Person(temp).acceptDivorce(address(this));
  isMarried = false;
}
```

`setSpouse()`

The `setSpouse()` method in question was **removed** as it needed to be `public` in order to ensure **mutual marriage** and **mutual divorce**, but with the downside that anyone can still change anyone's `spouse` without any consent.

To fix that, the `acceptMarriage()` and `acceptDivorce()` were introduced

```
// requires the significant other to already have spouse = address(this)
function acceptMarriage(address sp) public {
  require(Person(sp).getSpouse() == address(this), "The other part must
  ↪  have accepted to marry you.");
  require(isMarried == false, "You are already married to this person.");

  spouse = sp;
  isMarried = true;
}

// requires the significant other to already have spouse = address(0)
function acceptDivorce(address sp) public {
  require(spouse == sp, "You cannot divorce someone you are not married
  ↪  with.");
  require(Person(sp).getSpouse() == address(0), "The other part must have
  ↪  accepted to divorce you.");

  spouse = address(0);
  isMarried = false;
}
```

Echidna invariants

```
function echidna_isMarried_consistency() public view returns (bool) {
  return isMarried == (spouse != address(0));
}

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

```
function echidna_no_mother_marriage() public view returns (bool) {
  return spouse != mother;
}

function echidna_no_father_marriage() public view returns (bool) {
  return spouse != father;
}
```

Second part