Introduction

Modern Blockchain platforms employ smart contracts to implement business logic in a decentralized manner. However, smart contracts are currently developed in programming languages that are similar to languages/frameworks used for normal software development outside of the Blockchain space. While this makes it easier for developers to transition to smart contract development, various incidents over the last years have shown that simple bugs in smart contracts can have disastrous consequences. As such, it is important to research ways to ensure the quality of smart contract code. One important aspect is the ability to patch smart contract code and deploy the newly patched version of smart contracts. This study is about current techniques of patching and deploying patched smart contracts in the context of the Ethereum blockchain and the prevalent Solidity programming language.

In this study, you will use various methods of patching Ethereum smart contracts written in Solidity. The are a total of 3 tasks with several sub-tasks. Please perform the tasks in the order given, measure the time it takes to perform the task and answer the questions attached to every task in the provided answer sheet. Read through the task's description in this document before starting the task and start your time measurement after reading through the tasks description. Note that some of the tasks might require less time than other tasks (i.e., the expected time required per sub-task is somewhere between minutes and hours).

Study Environment: We provide you with a VM (username: `user`, password: `user1234`) image where all relevant tools and the answer sheet are pre-installed. All relevant files are located in the `Ethereum` directory, which is linked on the Desktop. A local Remix IDE is accessible via the web browser at `http://localhost:8080`, via the browser bookmarks or via the Icon on the Desktop. When you first start the Remix IDE you will be prompted to connect to the local `remixd` instance, which you must allow to access the files. The files located in the `Ethereum` directory are then accessible via `localhost` in the Remix IDE.

Starting Questionnaire

Please answer the following questions with either yes or no.

- (Q1) Did you write Solidity code in the last two weeks?
- (Q2) Have you previously worked on a production-grade Solidity-based Ethereum contract?
- (Q3) Have you previously worked on a production-grade smart contract on another Blockchain Platform?

Please answer the following questions by rating on a scale from 1 to 7, where 1 means you are not familiar at all and 7 means that you are most familiar.

- (Q4) How familiar are you with Blockchain technologies in general?
- (Q5) How familiar are you with the Ethereum Blockchain in particular?
- (Q6) How familiar are you with the Solidity programming language?
- (Q7) How familiar are you with upgradable contracts in Solidity?

Task 1: Patching Integer Overflow Bugs

In this scenario you are going to patch three contracts that have been previously attacked due to an integer overflow/underflow bug in the given smart contract. You will receive the output of an static analysis tool that discovers integer overflow bugs using symbolic execution. Based on this output, your task is to patch all integer overflow bugs in the given contract.

First, open the Remix Ethereum IDE. Then perform the following steps for every contract that is in the `task_1` directory.

1. Interpret the output of a specialized analysis tool (OSIRIS). You can find a text file called `osiris.log` in the respective directory, which contains the output of running OSIRIS on the respective contract source code. This report points you to various potential integer overflow that were discovered in the contract code.
2. Identify and patch all vulnerabilities in the contract's source code. For reference, the `SafeMath` library is given. This library implements common integer overflow checks for many arithmetic operations. This library can be used as an inspiration or it can be directly copied into the source code of the vulnerable contract.
3. Make sure that the contract still compiles with the Solidity compiler and save the contract source code.

Questionnaire

Please rate how confident you are in the correctness of your patches on a scale from 1 to 7, where 1 is the least confident and 7 is the most confident.

- (T1Q1) How confident are you in the correctness of your patch to `contract_1`?
- (T1Q2) How confident are you in the correctness of your patch to `contract_2`?
- (T1Q3) How confident are you in the correctness of your patch to `contract_3`?

Please report the time you required to patch the contract.

- (T1Q4) How much time did you need that you needed to patch all three contracts?

Task 2: Create and Deploy an Upgradable Contract

In this task you will create an upgradable contract using the so-called *delegatcall-proxy* pattern. Normal Ethereum contracts are immutable, i.e., their code cannot be changed once they are deployed. However, this fundamental design choice severely impact the operation of important smart contracts that are centrally managed. Fortunately, other features of the Ethereum platform can be used to create an upgradable contract using multiple contracts.

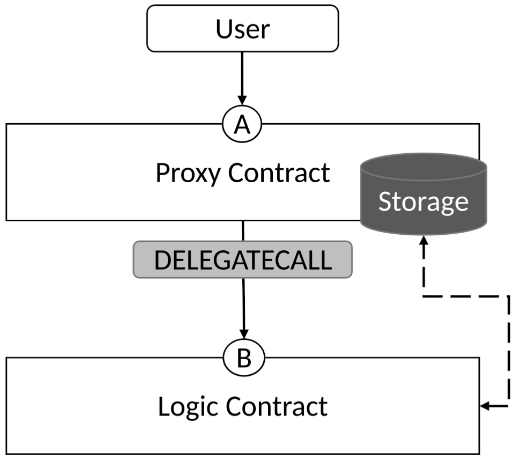

Figure 1: The delegatcall-proxy pattern

The delegatecall proxy pattern uses two contracts: a proxy contract, which is an immutable contract with a constant address A. The proxy contract holds all the contract state and cryptocurrency (ether) funds. The proxy contract forwards all actual business logic to the logic contract at a variable address B. Furthermore, the proxy contract includes minimal logic to change the address of the logic contract. This way an authorized user can change the address of the logic contract an therefore update the business logic of the contract.

Your task will be to convert an existing contract into this delegatecall-proxy pattern style. If you are not familiar with this pattern then you will first need to read up on the intricacies of using this pattern and how it works.

Finally, you will use an automated tool, *EVMPatch*, to convert and deploy the contract automatically into an upgradable delegatecall-proxy contract.

Step 1 - Manual Conversion to Delegatecall Proxy

Navigate to `task_1/1_manual` and open the `Wallet.sol` file in the Remix Ethereum IDE. Your task is to implement the delegatecall proxy pattern for the given contract `Wallet`, whose source code is located in `Wallet.sol`. Test the newly upgradable contract by deploying it from within the Remix IDE. To access the functionality of the `Wallet` contract, you need to "cast" your Proxy contract to a `Wallet` contract by using the *At Address* button in Remix to create a `Wallet` contract at the address of your Proxy contract. Your proxy should now be able to handle `deposit` and `withdraw` transactions. Please note, that the `Proxy` contract should also be able to handle any additional methods that are not yet part of the `Wallet` contract, but will be added in a future version of the `Wallet` contract.

To simulate a varying degree of prior knowledge we do not provide any additional help. Use any resource available to you as a developer to achieve this task.

Important: to limit the time required for this study, please use a timer to abort this task after a maximum of 3 hours.

Step 2 - Automatic Conversion with EVMPatch

Navigate to the `task_1/2_evmpatch` directory. Open the Terminal and run the evmpatch deployment script:

```
$ python evmpatch-deploy.py --help
usage: evmpatch-deploy.py [-h] [--verbose] contract contract_name

EVMPatch Command Line Interface for Contract Deployment

positional arguments:
  contract_source
  contract_name

optional arguments:
  -h, --help     show this help message and exit
```

You need to provide the path and the name of the contract in the Solidity source file to EVMPatch, for example:

```
$ python evmpatch-deploy.py ./source.sol ContractName
```

EVMPatch will then automatically convert the given contract to use the delegatecall-proxy pattern and deploy an upgradable contract.

Questionnaire

Please answer the following questions with yes or no:

- (T2Q1) Have you previously used the delegatecall-proxy pattern in a Solidity contract?
- (T2Q2) Have you previously used a different pattern to make a Solidity contract upgradable?
- (T2Q3) Have you previously used a different upgradable smart contract?

Please Rate the following questions on a scale from 1 to 7, with 1 being you do not agree and 7 being you agree the most.

- (T2Q4) How confident are you in the correctness of your conversion? (1 least confident, 7 most confident)
- (T2Q5) How difficult was the manual conversion? (1 easy, 7 most difficult)
- (T2Q6) How difficult was the conversion using the evmpatch tool? (1 easy, 7 most difficult)

Please report the time you required for each step.

- (T2Q7) How much time did you need to manually convert the given contract to an upgradable using the delegatecall proxy pattern (Step 1)?
- (T2Q8) How much time did you need to convert the contract using EVMPatch (Step 2)?

Task 3: Developing a Patch Template for EVMPatch

As you have seen in the previous Task, EVMPatch is a highly automated tool for managing upgradable contracts. EVMPatch goes even further and automatically introduces hardening against certain types of bug classes, such as integer overflows. However, EVMPatch requires developer-assistance for patching logic bugs, such as access control issues. In the *Wallet* contract there is a major security vulnerability: the function `migrateTo(address)` lacks any form of access control. Originally this function was intended to allow the owner/creator of the *Wallet* contract move all funds to a new address in an emergency. However, this function was not properly secured. As a consequence this function allows anyone to steal all the funds in the contract.

Your task is to create and deploy a patch using EVMPatch for the *Wallet* contract. Edit the `Patch.yaml` file and create a patch specification for EVMPatch. Then deploy the patched contract with EVMPatch.

EVMPatch Specification

EVMPatch uses a YAML based configuration syntax for patches. The following is an example for the EVMPatch *patchspec* format. Here multiple patches for a contract can be specified. EVMPatch currently supports two types of manually specified patches, `require` style patches and function deletion.

```yaml
add_require_patch:
  function_name:
    - "1 == 1"
    - "sload(owner) > 0"

delete_function_patch:
  - function_name
```

Function deletion simply removes a public function from the contract. The *patchspec* file must contain the keyword `delete_function_patch`, which is a list of functions names that will be deleted by EVMPatch. Deleting functions is useful to (temporarily) disable functions with vulnerabilities until a proper patch is developed.

The second patch type is the `require` style patch (specified in `add_require_patch`). Here EVMPatch allows to add expressions that are required to be true at the start of a function. This types of patches are especially well suited to implement access control checks. This type of patches can be thought of as being equivalent to adding a custom modifier to a function in Solidity and use the Solidity `require` statement to perform validation.

The require-style patches in EVMPatch use a custom domain specific expression language to specify the actual logic of the patch. This language supports typical comparison operators (`==`, `>`, `!=`, etc.), arithmetic and bit manipulation operators (`+`, `-`, `&`, `|`, `**` for exponentiation, etc.). However, due to the limited scope and the way EVMPatch applies the patches this language is not as expressive as Soldity source code. For example, there is no direct access to state variables. Instead the low-level `sload` construct must be used to load state variables (i.e., located in *storage*). To load the variable `owner`, the expression `sload(owner)` must be used. However, certain special Solidity variables, such as `msg.sender` (i.e., the caller of the current contract) and `msg.value` (the number of transferred ether) are directly available.

Patching with EVMPatch

EVMPatch streamlines a big part of managing and patching upgradable contracts. To automatically deploy an upgradable contract and apply the patches specified in the *patchspec* file format, the EVMPatch deploy and patch script can be used. The following command can be used to deploy a patched contract with EVMPatch.

```
$ python evmpatch-deploy-patch.py Wallet.sol Wallet Patch.yaml
```

To take things one step further, EVMPatch utilizes historic transactions from the blockchain to test the patch before deployment. This means EVMPatch will automatically verify that the newly created patch is compliant with prior transactions and breaks all previously known attack transactions.

Questionnaire

Please Rate the following questions on a scale from 1 to 7, with 7 being you agree the most.

- (T3Q1) How confident are you in the correctness of your patch? (1 least confident, 7 most confident)
- (T3Q2) How difficult was the conversion using the EVMPatch tool? (1 easy, 7 most difficult)

Please report the time you required to perform this task.

- (T3Q3) How much time did you need to create and deploy the patch using EVMPatch?

Submitting the Results

By submitting the results you consent that your answers and your changes to the source code will be analyzed and published as part of a scientific publication in an anonymous way.

To submit the results follow the following instructions:

1. Make sure that all files that you edited are saved.
2. Close all programs/editors
3. Make sure that all the questions in the `study_answers.ods` spreadsheet are answered and the file is saved.
4. Click on the *"Prepare Developer Study Icon"* icon on the desktop. This will create a ZIP file on the Desktop of the VM, containing your answers and your changes to the source code.
5. Submit the ZIP file via E-Mail.

Thank you very much for your participation in this study.