P2P FinalTerm - Report

Student: Andrea Lisi

1 Introduction

The proposed solution includes 3 files:

- **Catalog.sol**: the Catalog contract implementation;
- **BaseContentManagement.sol**: the BaseConentManagement contract implementation;
- **BaseContentImplement.sol**: 3 examples of contents: PhotoContentManagement contract, SongContentManagement contract and VideoContentManagement contract.

The solution is developed and tested on *Remix*.

2 The Content Management

The BaseContentManagement stores the values common to each content, such as `title`, `author`, `views` and `catalogAddress`. Assuming that the Catalog exists "from the beginning", and its address is easily available, `catalogAddress` is assigned by the constructor.

2.1 Access and consumption

The BaseContentManagement provides the function `grantAccess()` which stores the permission of consumption of a user. This function is supposed to be called only by the Catalog to avoid permission tampering. Thus, the `grantAccess` function checks if the caller (`msg.sender`) is `catalogAddress`. The `consumeConent()` function is called directly by a user who has the permission to consume the content. This function checks if the user is premium or not calling the Catalog's `isPremium()` function in order to increment the views and finally notifies the Catalog of the new consumption. Both `grantAccess()` and `consumeContent()` are summerized in figure 1.

2.2 Instance examples

The BaseContentManagement contract cannot be deployed since it doesn't implement the `getGenre()` function. The provided examples of instance of the base management are PhotoContentManagement, SongContentManager and VideoContentManager contracts. In this solution the sub-contracts hard-code the `getGenre()` function returning the bytes32 representation of their genre, e.g. "photo". In this way each possible content will have in common the same information (such as title, author), but additional metadata could be different from content to content. For example a PhotoContentManager could store the details of the photo (since each photo has the same representation), a SongContentManager could store song's duration etc etc. These are just examples, for simplicity the state of each sub-contract is empty.

3 The Catalog

3.1 Deployment

Whoever deploys the Catalog becomes its CEO and the only action that he/she can do as CEO is destruct the Catalog.

3.2 Adding a content

Once deployed the Catalog accepts as contents only BaseContentManagement contents: in this way is enough to extend that contract in order to deploy a valid content on the Catalog and it's not needed to modify the Catalog in order to accept new genre of contents.
A user can add a content to the Catalog calling the `addContent()` function which takes as input the address of the content. The Catalog checks if the caller is the author of the content and checks if the stored `catalogAddress` inside the content management is its own address. A malicious user could try to bypass the Catalog to add views to its content storing its own EOA address or the address of another malicious contract extending the Catalog (no error will be raised at compile time and during the assignment since a external accounts and contracts share the same address space[1]): in the first case the tampering attempt will fail as soon as `consumeContent()` is called since an EOA address doens't have the requested Catalog's functions; in the second case the deployed malicious extension of the Catalog will have a different address from the real Catalog (which by hypothesis exists "from the beginning") and linking the tampered content to the real Catalog will fail.
Each time a content is successfully linked the Catalog updates the "latest content" information about the author and the genre of the content, and fires "new content", "new author's latest" and "new genre's latest" events. This brings to further costs in terms of gas spent but it's reasonable that an author would accept to pay a little extra to speed up his/her the last content retrieval (avoid loops, see section 3.6) and to get more visibility in case someone listens to "new content" events.

3.3 Getting a content

A user can obtain the access to a content calling the `getContent()` function passing as parameter the title of the desired content. The `getContent()` checks if the sent ether are enough and calls the `grantAccess()` function of the content management.
The user can get a content for free calling the `getContentPremium()` function: this checks first if he/she has a valid premium and behaves in a similar way to `getContent()`.
Finally, a user can also gift contents to other users with the `giftContent()` function.
In any case, the content's management contract stores the interested user's permission to consume its content (section 2.1) and the Catalog fires a "new access" event.

3.4 Consuming a content

A user consumes a content interacting with the content's management contract and, as stated in section 2.1, the management sends a "cunsumption notification" to the Catalog, invoking the `notifyConsumption()` function. This notification is needed to let the Catalog fire the "new consumption" event and, if triggered, pay the author.
The `notifyConsumption()` function checks also if a content of a genre or an author is now the most popular: in this case the Catalog updates the relatives "most popular" states and fires "new most popular by genre" and/or "new most popular by author" events. These last updates are analyzed more deeply in section 3.7.

[1] `http://solidity.readthedocs.io/en/v0.4.21/introduction-to-smart-contracts.html#accounts`

3.5 Destructing COBrA

The COBrA CEO can close the Catalog calling the `destructCOBrA()` function. This triggers the payments to the authors proportionally to the views obtained. In case no content has a view (for example, every consumption was performed by premium accounts) the balance will be equally distributed among the authors, proportionally to the number of contents.
Right before destructing the contract the "shut down" event is fired.

3.6 The views

The Catalog implements various view functions to let a customer to easily visualize the contents, such as the latest or the most popular one. The Catalog is designed to limit the for loops needed by these functions (e.g. find the most popular photo). Even if for loops are gas-free for an external account in view functions, this is not true for contracts. Let's suppose someone would like to build a contract that calls these functions: the caller would pay the gas to execute the loops since the functions are called by a contract, and in case the Catalog has a big library of contents this will consume a lot of gas[2] as well as a large amount of time.
In this way the Catalog needs additional state update, such as after an author links a new content and after a customer consumes one. Section 3.7 takes in consideration the consume content action and gives an estimation and an analysis of the gas spent.

3.7 Consuming a content: gas estimation and analysis

Estimation

Considering these major costs:

- Storage writing: 21K gas if the current value is "zero" (0, 0x0, false); 5K otherwise[3];
- Event firing: 1500 gas;

Considering the first possible content consumption: the action follows these big steps:

- revoke the access rights of the caller user in the manager contract: (true $\rightarrow$ false), **5K gas**;
- first view increase in the manager: (0 $\rightarrow$ 1), **21K gas**;
- first total views increase in the Catalog: (0 $\rightarrow$ 1), **21K gas**;
- new consumption event: **1500 gas**;
- first update new most popular content genre and author: ("0x0" $\rightarrow$ data); ("0x0" $\rightarrow$ data), **21K gas $\times$ 2**;
- fire the two related events: **1500 gas $\times$ 2**;

plus a few function calls and conditional statements (the author's payment isn't triggered).
The sum of the operations listed above makes **93500 gas**. Performing the first time a consumption Remix states the gas spent by the execution is around **104K units**.

[2] `http://solidity.readthedocs.io/en/v0.4.24/security-considerations.html#gas-limit-and-loops`
[3] Gas spent when writing on storage, mapping example (stack exchange link)

Analysis

While we could think an author to be willing to spend a few more money for a better Catalog service and thus more revenue, maybe this won't work for the customers since they already pay for the content and for the gas to get and consume that content. So, an alternative would be to not implement any storage information about "most popular by author/genre" since the updates on this state will be triggered after a consumption and thus payed by the customers. But, as explained in section 3.6, to avoid loops and to keep the solution homogeneous these updates are also performed. Assuming that in the long run the most popular content by genre and author won't change frequently (for example if the most popular song of Green Day is Boulevard Of Broken Dreams it will be really hard nowadays to pay the gas to update the most popular content of Green Day) the overhead a user should pay to update these two information would be rarerly spent.

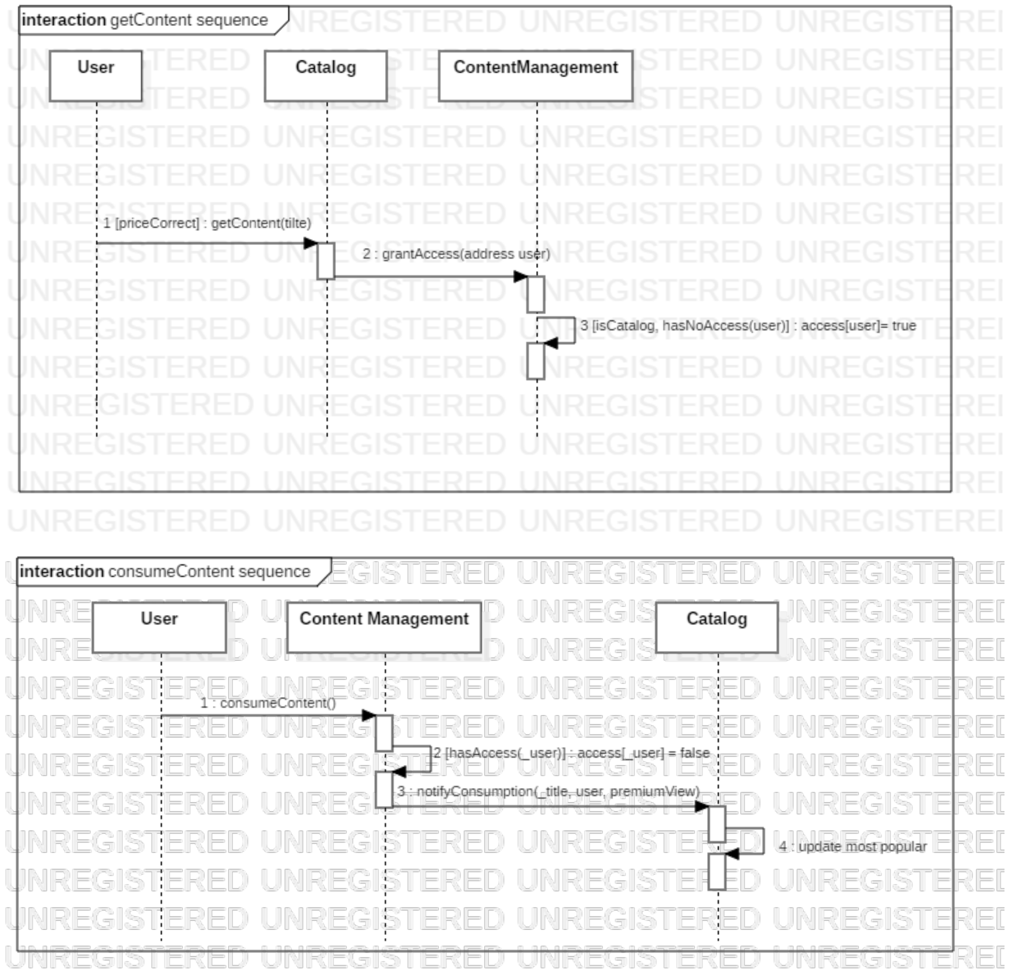
interaction getContent sequence
User
Catalog
ContentManagement
1 [priceCorrect] : getContent(tilte)
2 : grantAccess(address user)
3 [isCatalog, hasNoAccess(user)] : access[user]= true
interaction consumeContent sequence
User
Content Management
Catalog
1 : consumeContent()
2 [hasAccess(_user)] : access[_user] = false
3 : notifyConsumption(_title, user, premiumView)
4 : update most popular

Figure 1: `getContent()` and `consumeConent()` actions sequence diagrams

4 Action list and simulation

The simulation follows Remix environment and exploits its interaction system. Since the solution works with bytes32 the listed names such as titles and authors will be meaningless.

List of actors using the 5 user addresses Remix provides:

- **Addr1**: the deployer of the Catalog;
- **Addr2**: an author with name ”0xa”;
- **Addr3**: an author with name ”0xaa”;
- **Addr4**: a customer;
- **Addr5**: a customer;

List of actions with format **User** - `contract.action` {*eventual comment*}:

Convention: *with xxx.**address** is intended to copy and paste the address of the account/contract using the Remix copy button.*
Attention: *increase the default Remix’s gas limit from 3M to 4M: the deploy of the Catalog consumes 3.1M gas*

Fine execution

- **Addr1** - `Deploy Catalog()`
- **Addr2** - `Deploy PhotoContentManagement("0xa", "0xb", Catalog.address)` {*from now as 0xbContent*}
- **Addr2** - `Catalog.addContent(0xbContent.address)`
- **Addr3** - `Deploy PhotoContentManagement("0xaa", "0xbb", Catalog.address)` {*from now as 0xbbContent*}
- **Addr3** - `Catalog.addContent(0xbbContent.address)`
- **Addr4** - `Catalog.getContent("0xb")` {*value = 0.001 ether*}
- **Addr4** - `0xbContent.consumeContent()`
- **Addr4** - `Catalog.getContent("0xbb")` {*value = 0.001 ether*}
- **Addr4** - `0xbbContent.consumeContent()`
- **Addr5** - `Catalog.getPremium()` {*value = 0.04 ether*}
- **Addr5** - `Catalog.getContentPremium("0xb")`
- **Addr5** - `0xbContent.consumeContent()` {*Does not increase the views*}
- **Addr5** - `Catalog.giftContent("0xbb", Addr4.address)` {*value = 0.001 ether*}
- **Addr4** - `0xbbContent.consumeContent()`

Note: *at this point we should have 0xbb with 2 views and 0xb with 1 view*

Bad AddContent

- **Addr2** - `Deploy PhotoContentManagement("0xa", "0xbbb", Catalog.address)` {*from now as 0xbbbContent*}
- **Addr3** - `Catalog.addContent(0xbbbContent.address)`

***Error:* *Addr3* *can't add to the Catalog* *Addr2*'s content**

New most popular photo

- **Addr2** - `Catalog.addContent(0xbbbContent.address)` {*from now as 0xbbbContent*}
- **Addr4** - `Catalog.getContent("0xbbb")` {*value = 0.001 ether*}
- **Addr4** - `0xbbbContent.consumeContent()` {*Repeat get/consume twice more*}
- **Addr4** - `Catalog.getMostPopularByGenre("0x70686f746f")` {*"photo"*}

Return: *0xbbb*

Close the Catalog

- **Addr1** - `Catalog.destructCOBrA()` {*triggers the payments*}

5 Implementation details, major decisions

- As mentioned before, this implementation limits the amount of for loops needed by the functions using some more storage information. This choice aims to avoid too much gas consumption in case other contracts want to call the view functions of the Catalog and aims to save time while searching for a single content in a possibly big library.
- The contents are stored using 2 structures: an array of titles to support iterations needed by `getNewContentList()`, `getStatistics()` and the final cycle of payments; a mapping(title → base management) to support random access needed by `getContent()` and `giftContent()` functions. The title of each content is supposed to be unique.
- The consumption is simulated simply by resetting the access rights of the user on that content and notifying the consumption to the Catalog.
- The unit of time to measure the deadline for premium accounts is the block number. With the hypothesis of an average of 14 seconds between each block, 6170 are the estimated number of blocks to cover 24 hours of period. This will be affected by possible future changes of the average mining time[4].

[4] `https://etherscan.io/chart/blocktime`