Upgradeability: Good, Bad, Ugly

Mehdi Salehi, Jeremy Clark, and Mohammad Mannan

Concordia University

Abstract.

1 Introductory Remarks

2 Classification

2.1 Retail Changes

This is not a standardized pattern. The development team must consider the ways to upgrade the contracts before deploying the smart contract. Known patterns are different on the level of intervention to change the logic that they need to change in the future. The amount of changes are limited and system design can not be changed after deployment and just some system variables can be changed. We will describe three famous patterns here:

Parameter Configuration . The easiest way to upgrade the logic of the smart contract is to have some critical parameter that can change the whole logic of the system. For instance, in economy we have different variables which have effect in the interest rates. By changing those variables the governors will response to changes needed for the system. In this model we have a setter function to change the upgradable parameters if the system needs upgrades. The best example for this type of upgradeability is MakeDao project. In Maker there are some variables like Dai Saving Rate (DSR) or Stability fee that can be changed through governance vote. The logic behind the smart contract and the tokenomic of the Dapp completely depends on these variables.

Strategy pattern . The strategy pattern is an easy way for changing part of the code in a contract responsible for a specific feature. Instead of implementing a function in your contract to take care of a specific task, you call into a separate contract to take care of that and by switching implementations of that contract, you can switch between different strategies. An example for this pattern is Compound project and how they used strategy pattern for their interest rate model. There is a interest rate model contract in Compound that can be changed during the time.

Pluggable Modules . In this pattern we have a core contract that have some immutable features and then new contracts generated by the main contract and each have some or all features of the main contract. This pattern is mostly used in wallets and DeFi services like DeFi saver and InstaDapp. Users can decide to add new features into their wallet.

2.2 Wholesale Changes

In contrast to previous session, sometime we need to change the whole or a big part of the logic of our smart contract. This update could be a response to an incident happen to the smart contract or a planed upgrade of the system. Before deployment, the core developer team should have a plan for the upgrade events. We categorized these types into two main classes:

Contract Migration In the migration plan we should write a completely new contract with our desired new logic. In migration method our new version contract doesn't have any communication with the previous versions. The challenges we face in migration method are:

1. Grab the needed data (from previous contract or new data): It depends on the data type. It is easy for simple data structures (*e.g.,* uint, address, or even arrays) to collect the data just by reading storage slots from the 0 slot. we should take care of complex structures (*e.g.,* mapping) in the latest versions of our contract by adding event updates whenever a data added to a mapping variable. In case of an upgrade we can use Logs to find storage slot (using key hash) and collect the data. Sometime we need to push new data into our upgraded smart contract. For example in airdrops we need new coin to be added to some specific addresses.
2. Push the data into the new contract: Using Constructor, we can use batch transfer function with arrays of addresses and amounts as inputs. This way we can push lots of data using a single transaction. One limitation here is block gas limit. If we exceeds the block gas limit we need to push all data in different blocks (pausing in the first block and unpause at the end). Recently, Devs are using merkle distribution tree to push data on to the smart contracts. The most important thing here is the cost of pushing data to the new version. It depends on 1)the number of storage slots to be updated and 2)Method used to push the data on-chain. (Can be tested).
3. Stop the previous contract: Suppose that we have a token contract and we want to migrate to a new version. We should be confident that nobody can use the previous contract. If not, it is possible that a person sell a token from previous contract (which should be valueless after migration) to a person who doesn't aware of the migration plan.Because of the decentralized nature of the blockchain and Dapps you cannot reach to your customers to alert them from using the previous contract. One way to do that is have a pause option the your contracts and pause the old versions before migration.

Contract Migration is less riskier than other types of upgrades, not cost effective compare to some upgradeability types but more decentralized to the other solutions. Also, it's not good for frequent updates.The other advantage of this method is that it removes transaction gas cost needed for patterns like proxy, registry or call-based methods.

diamonds

Create2-based approach

Data Separation patterns The other type of wholesale methods is to separate data and logic part of our codes. In the case of the upgrade we can keep the storage contract and just upgrade the logic contract and link the new version of logic contract to the storage contract. There is a debate on whether this type of upgradeability is cheaper or not in comparison to migration method. But, this method is more efficient for Dapps in which we need frequent updates. The other important issue here is who decides on the changes we need for the system which we will discuss on further sections. Here we have 2 different choices using Call method and Delegate Call method to link storage and logic contracts together.

Call based patterns In this type the interaction between logic and storage contact is handle by Call opcode in Ethereum Virtual Machine. In call based patterns user is supposed to call the logic contract and the logic contract will call the storage contract. The logic contract is the one that can be upgraded.

There are two concerns in this approach: how to store data and how to perform an upgrade.

Storage. The easiest way to store data in storage contract is to have a modifier on the setter functions in the storage contract that allow just the logic contract to change the variables. The owner of the contract can change the address of logic contract for the modifier. In this approach for adding a new persistent variable, a new data contract should be deployed which may be costly in case that the application needs lots of upgrades.

The other way to store data is so called Eternal storage (ERC930). Eternal storage uses mapping (key-value pair) to store data, using one mapping per type of variable. The EVM storage layout and how it handles mapping helps the Eternal storage pattern to be more amenable to evolution but also more complex.

Upgrade implementation. There are three main ways to implement upgrades using data separation pattern. The easiest way is to change the ownership of storage contract into new upgraded logic contract and then **Pause** the old contract or set its pointer to 0x0 address. The other solution is to forward the calls receive by the old contract into the new logic contract. The last option is to set a proxy contract that just keeps the address of logic contract and call into logic contract.

DelegateCall-based upgrades Similar to call based patterns here we have two contracts, Storage and Logic contract. we may have more than one logic contract. The difference here is that the user is calling storage contract first(called proxy contract), and the proxy contract DelegateCalls to the logic contract(s). The main difference between delegatecall and call-based approach is that in delegatecall proxy pattern the storage layout of proxy and logic contract should be the same. The difference between storage layouts will result in storage clashes. There are three different methods to mitigate the risk of storage clashes:

Inherited Storage. In this method the proxy contract and all logic contracts are inherited from a storage contract that contains storage variables. Using this method we are confident that the proxy and logic contracts are using the same storage layout and storage clashes will be mitigated. After deployment if we need new logic contract with new storage variables, we should deploy a new storage contract that inherits the previous storage contract. Then the new logic contract must inherit the new version of the storage contract.

This method is not efficient because of variables that declared but not used in some logic contracts. On the other hand, each logic contract is coupled with a storage contract and it is hard to take care of this track.

Eternal Storage. In this pattern, we defined mappings for all variable types that we need to use in our logic smart contract. For storing mapping variables EVM selects random slots on the storage based on the variable's name so we can mitigate the clashes using this randomness.

The main problem of this type is that the logic contract and all other contracts that are using the storage must use the mapping structure to access the storage variables and use complex syntax whenever they want to access a variable. This also results in the gas usage inefficiency because we need to call and update a mapping each time we need to change a variable.

Also it is hard to use eternal storage for complex variables like mappings and structure (need mapping of mapping pattern). Also finding a state variable of the proxy smart contract is hard because we store them in arbitrary slots of the storage.

Unstructured Storage. The other way of mitigating the storage clashes is to assign some randomly selected slots to critical variables like address of logic contract. For instance, open zeppling uses hash of "org.zeppelinos.proxy.implementation" to store the address of the logic contract in this slot.

The downside of this approach is that we need getter and setter function for each variable. We also can use unstructured storage for simple variables and not for mapping and structures.

3 Evaluation of different methods

In this section we compare and evaluate different methods discussed in previous section. There are some characteristics that can help the designer to decide which method should be used on the system and add upgradeability to the Dapp.

For our evaluation framework, we provide the definition of each evaluation criteria (i.e., column of the table), specifying what it means to receive a check mark (✓) or nothing.

Can replace entire logic An upgradeability method in which the upgrader is able to replace the entire logic of the system earns a check mark(✓) otherwise it receives nothing.

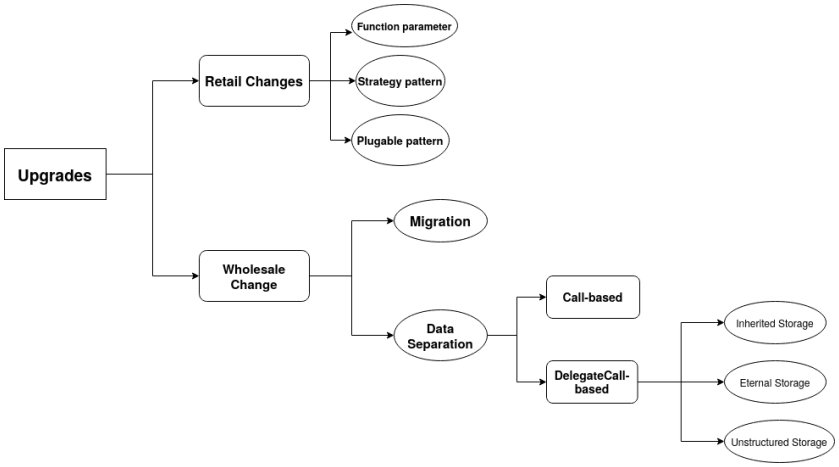


Fig. 1. Classification

can replace pre-specified part of logic An upgradeability method in which the upgrader can change *just* pre-specified part of logic of the system (and not entire logic) earns a check mark(✓) otherwise it receives nothing.

Can replace entire state An upgradeability method that gives the upgrader ability to replace or transfer the entire state to the newer version earns a check mark(✓) otherwise it receives nothing.

can change pre-specified state variables An upgradeability method in which the upgrader can just change some pre-specified state variables receives a check mark (✓) otherwise it awarded nothing.

3.1 No need to deploy a new contract

Using some upgradeability patterns, the upgrader needs to deploy a new smart contract in the process of upgrade which receives nothing. Upgradeability methods which do not need to deploy a new contract receive a check mark (✓).

3.2 No need to migrate state from old contract

In some patterns, the upgrader needs to collect and transfer old data from previous version to the newer one which receive nothing (✓) otherwise it receives a check mark (✓).

3.3 No need to separate State and Logic

An upgradeability pattern that does not requires separation of logic and storage contracts awarded a check mark (✓) otherwise it receives nothing.

3.4 Not using DelegateCall opcode

Some of the upgradeability methods utilize *Delegate call* opcode. Using this opcode bring complexity to the system and needs more security considerations (*e.g.,* checking storage layout compatibility). Upgradeability methods that do not need to use delegate call opcode receive a check mark (✓) otherwise it awarded nothing.

3.5 No indirection

Upgradeability methods that do not need any redirection receive a check mark (✓).An upgradeability pattern that adds an extra gas because of adding one or more layers of indirection awarded nothing. An upgradeability method in which just a portion of its transactions need indirection receive square (⊠).

Retail Changes and *Migration* methods do not add any indirections to the system so will not alter the transaction costs (✓). However, *Call-based* and *DelegateCall-based* approaches increase the transaction cost for users (receive nothing). *Call-based* method will increase the transaction cost because whenever a transaction needs to add or change a data, the logic contract must **CALL** the storage contract which adds an extra gas. Also in *DelegateCall-based* approach all transactions will be **Delegate Called** to the logic contract using a **proxy** which adds an extra gas on each transaction.

We need a comparison between Call based and Delegate call based approaches (if possible. I think Call based approach is gas efficient for systems that do not need to add or change data frequently. Also there are other extra adds-on that can add useability but adds gas like adding feature for uninformed users in Call based approaches.)

3.6 User endpoint address not changed

The title could be: *Having 2 Dapps at the end* ?! In some upgradeability methods, after the upgrade process, users must call a new contract address to use the Dapp. For instance, Alice uses a DApp at address A before the upgrade. After upgrade, she can be unaware that upgrade happened or she may need to use address B instead (✓).

3.7 Users do not require taking some action (withdraw deposit)

The title could be: *Users are involved? (Client)* ?!

In some upgradeability patterns, users are responsible to transfer their related data from the previous version to the new one. In some others, users do not need to take any further actions like this (✓)).

3.8 Change pattern after deployment

Migration is the only way to add upgradeability feature to out smart contract

3.9 Speed of an Upgrade

Another difference between upgradeability methods is the speed in which an upgrade can be processed. This depends on the type of decision makers which will be discussed in 4.1.

Retail changes method is the fastest way to upgrade a system comparing to other methods. Using an EOA as the decision maker is the fastest option of an upgrade. Using multi-sig is a bit slower than using EOA. Utilizing a decentralized governance scheme to decide about upgrades will put an inherit time delay to the upgrades.

Migration has the slowest upgrade process between other methods. The reasons are discussed in the previous parts (see **??**).

Call-based and *DelegateCall-based* are very similar to each other in the speed of upgrade. These two are not as quick as *Retail changes* because the developer needs to find the root cause of a bug or find the upgrades needed for system and then implement the smart contract and deploy it to the system which is time consuming. On the other hand these two approaches are faster than *Migration* because as mentioned before, in Migration plans we need to collect and push old data into newer version as well.

3.10 Level of Decentralization

The last and one of the most important characteristics that are different in upgradeability methods is the level of decentralization. An upgradeability methods that a single third party decides the upgrading process receives a square (⊠). Using an **EOA** to decide about a change is the most central option that a system designer can choose regardless of the upgradeability method uses in the system. In case a group of whitelisted persons can decide on the changes orf the system using **Multi-sig** is not decentralized as well. Although it improves the level of decentralization of the system but at the end a specified number can decide to change the system. So it awarded an empty circle (○).

Utilizing a decentralized governance model to vote for a change is a good way to make the decision making on the upgrades more decentralized. *Retail Changes* using voting scheme is more decentral than *Call-based* and *DelegateCall-based* because boundary of changes are limited on the Retail methods so it awarded a full circle (●). But, in CAll-based and DelegateCall based methods the developers have the power to put some kind of backdoor in the system while upgrading and they receive a half circle (◐).

The *Migration* method is the most decentralized approach because it gives the users chance to decide whether to move to the newer version or not so it awarded a full circle (●). For instance, Uniswap uses this method for its upgrade and the users have choice to transfer their funds from Uniswap V2 to V3 or not and as we can see some users decide to stay on the previous version.

4 Upgrading process

4.1 decision maker(s)

There is a debate on who is responsible for upgrading a Dapp. Different systems can choose one of these schemes to upgrade their Dapp depending on the complexity of the system, frequency of the changes needed for the system and how fast does the system need to upgrade in the incidents.

Externally owned Address The easiest and the fastest way to upgrade a system is through a single address which is the owner of smart contract. This is the most centralized solution we have for upgrading a system. The main problem with this issue is the security of the system because it only depends on a single private key hold by the owner. In case of malicious party or if an attacker find the owner's private or if the owner lose the key the entire system is on the risk.

First Dapps on the ethereum blockchain used this method for the upgrade but it is not used these days because it is far from the idea of *Decentralization*.

Multi-Sig A *m out of n* Multi-sig wallet is a smart contract that can manage a transactions only if m number out of a specified n EOAs agree and sign the transaction. We can use address of a Multi-sig wallet as the owner of the system. In case of a upgrade or responding to an incident m number of the governors can permit to upgrade the system.

This is a better answer to the decision making of the upgrade compare to using an EOA in case of centrality while keeping the speed of an upgrade process. However, it is not decentralized. One way to reduce the level of centralization is to use different trusted teams who are stakeholders of the system in the multi-sig wallet.

Governance Voting The most decentralized way to decide on a system change is to do it using decentralized voting. This can be done by distributing governance voting tokens to the community and then they can vote on a change proposal by staking their voting token.

There are some critique to this method. Governance by voting has an inherit time delay to the upgrading process. This raises a problem when the system needs an instant upgrade (*e.g.,* responding to an incident). This means we need another mechanism to quickly fix bugs and upgrade the system on the event of incidents in conjunction with the voting process (*e.g.,* Global shutdown in MakerDAO).

It is also not cost-efficient for the voters because all token holders must send a transaction and needs to pay network fee.

The other problem with this method is fair distribution of the tokens. If the governance token does not distribute fairly and the majority of tokens granted to the limited number of users, then it is very similar to the multi-sig method which is more costly and complex. Because, whales of the governance token can vote to any desired change of the system similar to the multi-sig.

4.2 Mitigating risks

There are critical setups on the systems to mitigate the possible risks on the upgrading process. We mention some of them here with risk associated with them.

Timelocks In some project, there is a time window between every changes that approved on the system and when they affect the system. This gives opportunity to the users who are not satisfied with the upcoming upgrades to move their funds out of the system. However this is not proper in case of fixing a bug, because we need to patch the problem quickly.

Thereshold In multi-sig and governance upgrade methods we need a threshold on votes to decide whether a change is approved or not. This threshold should be big enough to be confident that upgrading event represents the majority of opinions. On the other hand, the threshold shouldn't be that big because a big threshold will delay a system change. The system designer should consider that a portion of voters (signers in multi-sig or governance token holders in voting method) may not be available in the event of the upgrade and having a big threshold may result in halting the change proposal for a long period of time. In fact threshold has a trade-off between security/decentralization and speed of the upgrade process.

Pausable In pauseable smart contracts, the decision makers (usually a multi-sig wallet) can freeze some or all operations of the system. Pausing a smart contract helps in some specific situations:

1. Time to react to a bug or hack: usually it takes time to analyze and find the reason of a hack and patch the bug. In this time period the core developer team needs to pause the system to stop attacker from draining all the fund.
2. Halting system in the upgrade process: For instance, in an ERC20 token contract upgrade we need to pause the system to stop users from transferring tokens during the upgrade.
3. Inactivating the previous version of the logic contracts: After an upgrade we need to have a plan to stop users from using the previous logic contracts. One way to do so is to make the logic contracts pauseable and pause them after the upgrade.

Escape Hatches A escape hatch is a mechanism that lets the users to move their fund out of the system in the pausing events. For instance, in MakerDAO we have an emergency shutdown mechanism that pauses the system in the black swan events. But, users have the ability to extract their funds out of the system while the system is paused.

Front-Runnign Upgrading a smart contract can be done by sending a transaction into the system. If the upgrade is a response to a unknown bug, then the upgrade process will hint attackers who is listening to the mempool to find the bug and hack the smart contract just before the upgrade. So there should be some mechanisms to mitigate front-running attacks. One solution to this issue is to use commit-reveal schemes. The team first sends a commitment of the upgrade (hash of the upgrade) to the system and after the timelock they can push and apply the original code which cannot be front run.

5 Measurement study

In this section we aim to shed light on these vital unanswered questions:

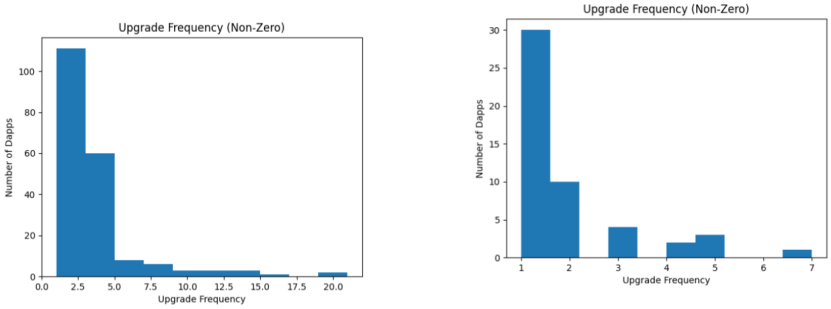
1. How many of the existing Dapps on the Ethereum blockchain are upgradeable?
2. Who can control these upgradeable Dapps (i.e who is the admin)?
3. In what extent the Dapp can be changed?
4. What is the average frequency of updates on the Ethereum Dapps?
5. How many times the admin of upgradeable contracts changes on average?

For now, we focused on the *DelegateCall-based upgrade* pattern because it is the most favorite pattern for Dapp developers (We can have 2 approaches here. One is to find a way to prove that this is the most used approach. The second way is to try to find a way to have measurement study for other patterns. I think the first approach is better because some of the other approaches are dead and nobody uses them anymore (like Call-based approach) and the others (like parameter change approach) is impossible to catch.)

There are three main methodology types to have a research on Ethereum blockchain; Transaction-based analysis, Bytecode based analysis and Etherscan Verified Smart Contracts based analysis. The first and second methodologies need to access to an Ethereum full archival node. But the last approach need to collect smart contracts from a third party service named Etherscan.io in which the Dapp developers are able to put their high-level language smart contracts which will be checked and verified from the bytecode of the Dapp on the Ethereum blockchain.

5.1 Verified Smart Contract Based Analysis

In the first attempt we conducted a measurement study based on verified smart contracts on Etherscan to find instances of OpenZeppelin upgradeable proxy contracts. OpenZeppelin is a team that built well-known libraries for ethereum developers. One of the most famous libraries of the OpenZeppelin is their upgradeability plugins. There two different versions of the OpenZeppelin upgrade libraries; version 2.6 and version 3+. We used smart-contract-sanctuary [2] which is a data set of all ethereum smart contracts verified on Etherscan.



(a) OpenZeppelin V2.6 (b) OpenZeppelin V3+

Fig. 2. Number of Dapps (y-axis) over number of upgrades (x-axis)

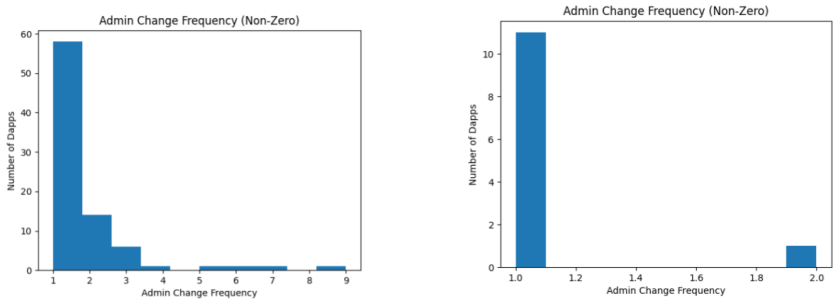
To find the smart contracts which used different versions of the OpenZeppelin upgradeability library, two different approaches are used. First, we used an Abstract Syntax Tree (AST) based similarity detector: Solidity Doppelgaenger [1]. We found 143 smart contract that used version 2.6 and 125 version 3+ OpenZeppelin upgrade libraries.

In another attempt, we used Regular Expression (regex) to find if the smart contracts on our data set used the interface (function and variable used on the code) of the each version of the OpenZeppelin libraries. The unique interface that used in each version of the OpenZeppelin libraries helps us to find them just by checking the interface. We found 922 contracts using V2.6 and 309 contracts using V3+. All results from AST-based detection was found on the interface detection results. So we used the results from interface detection for the rest of this part.

Now that we have the address of upgradeable smart contracts we can check how many times the upgrade happened for each and how many times the admin of the upgradeable contract is changed. For this purpose, OpenZeppelin libraries have used Ethereum Events on the functions in which the upgrade happens (Event "Upgraded) and the admin changes (Event "AdminChanged) and emitted when the events happened. We capture these events for each contract address using Etherscan Ethereum Log API service.

The results for each version is:

- For V2.6 (922 contracts):
  - Upgrade Happens (see figure 2a):
    - 110 with One Upgrade, 60 with Two Upgrade, 2 with 20 upgrades, 5 with Five upgrade, 4 with 7 Upgrades, 3 with 10,12,15,16 Upgrades, 1 with 16 upgrades
  - Admin Change Happens (see figure 2b):
    - 57 with One Change, 14 with Two change, 4 with Three change, 1 with 4 changes, 1 Dapp with 5, 6,7,9 Changes



(a) OpenZeppelin V2.6 (b) OpenZeppelin V3+

Fig. 3. Number of Dapps (y-axis) over number of Admin Changes (x-axis)

- For V3+ (309 contracts):
  - Upgrade Happens (see figure 3a):
    - 30 with One Upgrade, 10 with Two Upgrade, 4 with Three upgrade, 2 with 4 Upgrades, 3 with 5 Upgrades, 2 with 7 Upgrades
  - Admin Change Happens (see figure 3b):
    - 11 with One Change, 1 with Two change

There are some other measurement studies there like Pausable tokens which I skiped for now and we can add them as well

5.2 Hybrid Analysis (Transaction and Bytecode based)

As mentioned in classification section, the *DelegateCall-based upgrade* approach is consists of a storage contract (a.k.a proxy contract) and a logic contract (a.k.a Implementation contract).

Proxy contract is a simple type of smart contracts in which there is a fall back function. Fall back functions are *Payable* functions that do not have a function name. It means if a user sends a transaction to the contract including a call to a function that does not exists on the contract, it will passed into the fall back function and the logic on the fall back function will be executed. Also because of *Payable* nature of fall back functions if the user sends Ether into a smart contract the fall back function will be executed.

Inside the fall back function of the proxy contracts, there is a delegate call to the address of *Implementation contract* and pass the input data of the transaction to the implementation contract without altering it.

All proxy contracts have the above structure. But *Upgradeable proxy contracts* should have another extra condition as well. The owner of the contract (a.k.a admin) must have the ability to change the address which the proxy contract delegate calls into. If a proxy does not have this condition, it means that the

contract is just forwarding the calls into a fixed implementation contract for the rest of its life and so, this proxy is not upgradeable. There are bunch of patterns that follows this structure (e.g. Minimal Proxies, Delegate call forwarders, etc.) which we call them *Forwarders* in the rest of the paper.

Methodology In this section we describe the processes of finding upgradeable proxy contracts from scratch. Each block of Ethereum blockchain is consist of transactions that processed in that exact block. To get all transaction details we need to replay the transaction and collect the execution data. Ethereum full archival nodes have a method, *trace_transaction*, that gives the transaction traces of transactions inside an specific block. Each transaction trace is composed of actions that each of them consists of an opcode that changes the state, memory or stack of the Ethereum blockchain.

Having the transaction traces for each block, we will search to find actions that consists of *callType* and the call type is *delegate call*. If the input element (a.k.a call data) of these actions are equal to the input element of their previous action it will be interpreted as the transaction goes through a fallback function that delegate calls it to another contract. So, these contracts meet our first condition that described above and we will mark them as *Proxy Contracts*. We will collect *From address*,*To address* and the *transaction hash* of these picked actions and pick the unique from addresses of this data.

As discussed before, these picked contracts are proxy contracts and not necessarily upgradeable proxy contracts. So, we need to filter the forwarders from proxy contracts to have upgradeable proxy contracts. As mentioned above, the upgradeable proxy contracts must have a condition; the admin of the contract should be able to change the implementation address which proxy contract delegate calls to. In other words, we should check if the implementation address in the proxy contract is changeable or not.

Four different cases is possible for the situation that the implementation address is fixed and not changeable on the contract:

1. The address is hardcoded on the contract without assignment to a variable
2. The address is saved on a constant variable
3. The deployer adds the address via a constructor function
4. The address is defined in a storage variable, but there is no way to change this address after deployment

In the first three situations, the address will be appeared on the bytecode of the smart contract. So, we can get the bytecode of the each collected proxy contract using the *eth_getCode* method of a Ethereum archival node. Then we need to check if the implementation address of the proxy contract is appeared on the bytecode or not. The *To addresses* that we collected on the previous part is supposed to be the implementation addresses because these are the target addresses of the delegate call. So we easily checked if the To addresses is appeared on the bytecode of the From addresses contract. If yes, the addresses are not upgradeable proxies and these are simple forwarder contracts.

For the situation 4, we need other processes to check whether the implementation address is changeable or not. We need to design a filter to tell us is there a way to change the implementation address of the proxy contract or not. We will describe the filter in the later part of the paper, but for now let's assume that we have a software that checks if there is a function on the proxy contract that the admin of the contract can change the implementation address of the proxy by calling the function, and if yes mark the contract as an upgradeable proxy contract. So, now we have output addresses that their contract has fall back functions that delegate calls the whole data into another implementation contract and the address of the implementation is changeable on the proxy contract. So, we have a dataset of proxy contracts.

But this is not the end of story. There is another proposed upgradeable proxy contract pattern, named Universal Upgradeable Proxy Standard (UUPS) also known as EIP-1822. In this type of proxy contracts, the address of implementation address in the proxy contract, is changeable using the implementation smart contract. As mentioned above the code of the implementation contract is implemented in the context of the proxy contract and so it will change the storage state of the proxy contract. If we have a function in the implementation contract that gives us the ability to change the storage slot of the implementation address in the proxy contract, we can upgrade the system by calling that function of implementation contract. So, we couldn't catch these type of upgradeable contracts by the previous processes because we just checked if there is a function on the proxy contract that can change the implementation address.

We can tackle this problem by first finding the storage slot of the proxy contract in which the implementation address is saved, and check if the admin of the implementation contract can alter the variable of the that specific storage slot using a function in the implementation contract. So first we should find the storage slot of the proxy contract in which the implementation address is saved. There are two EIPs out there that makes it easy to find these storage slots; EIP-1967 and EIP-1822. EIP-1967 suggested to use 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc, and the EIP-1822 suggested 0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7 as the storage slot that should be used for implementation addresses (I can find the storage slot from the proxy contract but it needs us to check lots of contracts which needs high resources and with the existing resources it is not doable so I limited it to these EIPs which I think it's enough)

Now we will check the remained proxy contracts, if the variable that holds implementation address, is not saved on one of the mentioned slots, we will mark it as a forwarder. Now we should check whether the implementation address in the proxy contract is changeable via implementation contract. As mentioned before, the To addresses are the implementation contracts. We first find the variable in the implementation contract that is saved in the mentioned slots. Then pass it to our filter to check if the admin of the contract is able to change it or not. If it is changeable, it means that the admin can upgrade the system using

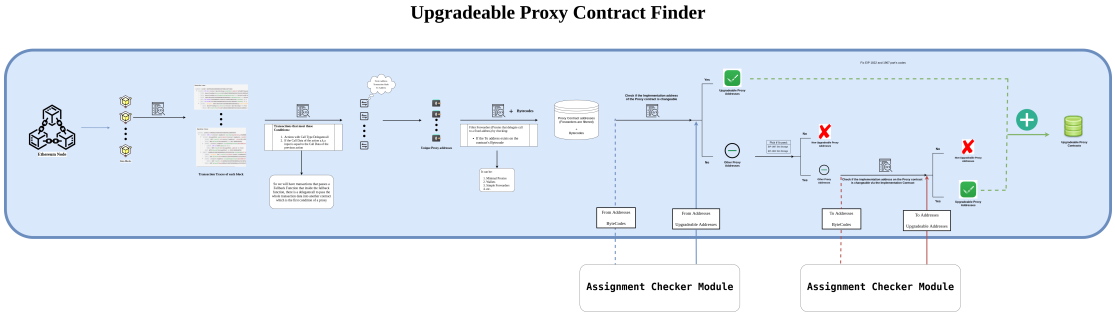


Fig. 4. Upgradeability Proxy Contract Finder

a function on the implementation contract, and so the proxy is an upgradeable proxy contract.

The whole process is depicted on figure 5.2.

Assignment Checker Module As mentioned in the previous part, we need a module to check if the admin can change the implementation address on the proxy contract or implementation contract. For this purpose we need a module that gets *Bytecode* of the proxy/implementation address as input and find the variable name and storage slot of the variable that the implementation address is saved into it. Afterwards check is there any function inside the contract that gives the admin the ability yo change the implementation address.

As the first action we used a bytecode decompiler named *Panoramix decompiler* to decompile the bytecodes into well-formated python language codes. The decompiled code gives us all storage variables and their storage slots in a function named **Storage**. On the other hand, the decompiled code will tell us if a function is *Payable* or not. Among these Payable functions the one that does not have name or it's name is fallback is the fall back function of the contract. So we will try to find the line that *Delegate Call* happened on it and collet these lines. Now that we have storage variable names and sotrage slots of them and also the line of code inside fallback that have the delegatecall, we will check to find the implementation address variable. We are doing that by checking if one of the storage variables are inside the line of code or there is a hash a text is inside the line of the code (if there is 'sha3' inside the line).

There is two other steps here. First if there is a variable name if the Storage function that has the same slot with implementation variables we captured from previous part we will add those variables to implementation variables. Also if the find another variables that being assigned to captured implementation addresses, we will add them to the implementation addresses as well.

Now that we have a list for implementation variables, we will search through the code to find if any assignment happened to one of them. If yes we will pick the variable that is assigned to implementation variable and then check if this picked variable is the input of the function in which the assignment occurred.

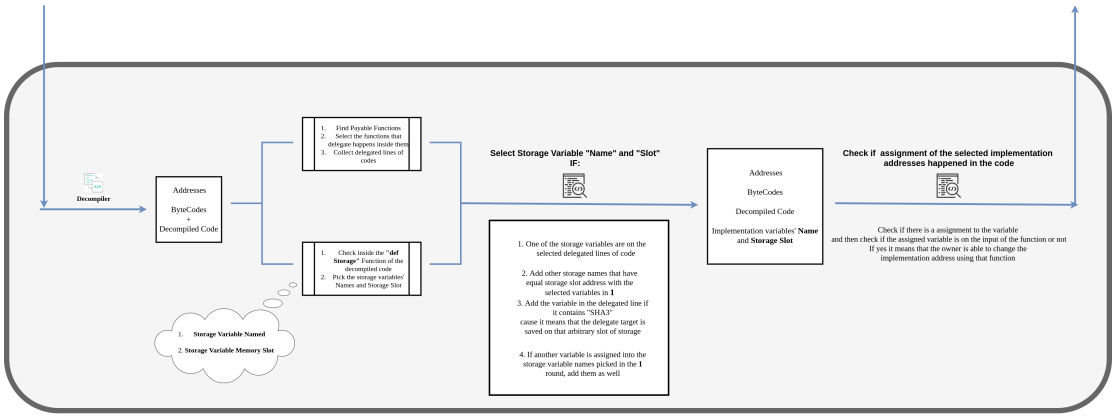


Fig. 5. Assignment CheckerModule

To summarize what we did, we find all possible variables in the code that can change the implementation address of the proxy/implementation contract and check if there is any function inside them that can alter the implementation address.

The whole process is depicted on figure 5.2.

6 discussion

References

1. Ortner, M.: Solidity doppelgaenger `https://github.com/tintinweb/solidity-doppelganger`
2. Ortner, M., Eskandari, S.: Smart contract sanctuary `https://github.com/tintinweb/smart-contract-sanctuary`

[1] Design of system in which a parameter can change the logic is hard

	Can replace entire logic	can replace pre-specified part of logic	Can replace entire state	can change pre-specified state variables	No need to deploy a new contract	No need to migrate state from old contract	No need to separate State and Logic	Not using DelegateCall opcode	No indirection	User endpoint address not changed	Users do not require taking some action (withdraw deposit)
Retail changes (Parameter change)				✓	✓	✓	✓	✓	✓	✓	✓
Retail changes (Strategy Pattern)		✓				✓	✓	✓	⊠	✓	✓
Migration	✓		✓				✓	✓	✓		
Call-based	✓					✓		✓			✓
DelegateCall-based	✓					✓				✓	✓

Table 1. Evaluation