Not so immutable: Upgradeability of Smart Contacts on Ethereum

Mehdi Salehi[1], Jeremy Clark[1], and Mohammad Mannan[1]

Concordia University, Montreal, Canada

Abstract. A smart contract that is deployed to a blockchain system like Ethereum is, under reasonable circumstances, expected to be immutable and tamper-proof. This is both a feature (promoting integrity and transparency) and a bug (preventing security patches and feature updates). Modern smart contracts use software tricks to enable upgradability, raising the research questions of *how* upgradeability is achieved and *who* is authorized to make changes. In this paper, we summerize and evaluate six upgradability patterns. We develop a measurement framework for finding how many upgradeable contracts are on Ethereum that use certain prominent upgrade patters. We also measure how they implement access control over their upgradability: about 50% are are controlled by a single Externally Owned Address (EOA), and about 20% are controlled by multi-signature wallets in which a limited number of persons can change the whole logic of the contract.

1 Introductory Remarks

The key promise of a smart contract running on Ethereum is that its code will execute exactly as it is written, and the code that is written can never be changed. While Ethereum cannot maintain this promise unconditionally, its assumptions (*e.g.,* cryptographic primitives are secure and well-intentioned participants outweigh malicious ones) provide a realistic level of assurance.

The immutability of a smart contract's code is related to trust. If Alice can validate the code of a contract, she can trust her money to it and not be surprised by its behaviour. Unfortunately, disguising malicious behaviour in innocuous-looking code is possible ('rug pulls'), and many blockchain users have been victims. On the other hand, if the smart contract is long-standing with lots of attention, and security assessments from third-party professional auditors, the immutability of the code can add confidence.

The flip-side of immutability is that it prevents software updates. Consider the case where a security vulnerability in the code of a smart contract is discovered. Less urgently, some software projects may want to roll out new features, which is also blocked by immutability. There is an intense debate about whether this is a positive or negative, with many claiming that 'upgradability is a bug.' We do not take a position on this debate. We note that upgradability is happening and we seek to study what is already being done and what is possible.

Is there a way to deploy upgradeable smart contracts if all smart contracts are (practically speaking) immutable? Consider a two simple ideas. The first is to deploy the upgraded smart contract at a new address. One main drawback to this is that all software and websites need to update their addresses. A second simple idea is to use a proxy contract (call it P) that stores the address of the 'real' contract (call it A). Users consider the system to deployed at P (and might not even be aware it is proxy). When a function is called on P, it is forwarded to A. When an upgrade is deployed to a new address (call it B), the address in P is changed from A to B. This solution also has drawbacks. For example, if the proxy contract hardcodes the list of functions that might be called on A, new functions cannot be added to B. Another issue is that the data (contract state) is stored in A. For most applications, a snapshot of A's state will need to be copied to B without creating race conditions. Mitigating these issues leads to more elaborate solutions like splitting up a contract logic and state, utilizing Ethereum-specific tricks (fallback functions to capture unexpected function names), and trying to reduce the gas costs of indirection between contracts.

Contributions. Finalize this text later.

2 Background and Related Work

Updating vs. upgrading. Software maintenance is part of software's lifecycle, and the process of changing the product after delivery. Often a distinction is drawn between software *updates* and software *upgrades*. An update modifies isolated portions of the software to fix bugs and vulnerabilities. An upgrade is generally a larger overhaul of the software with significant changes to features and capabilities. In our paper, we will only use the term upgrade and instead distinguish between retail (parameters and isolated code) and wholesale (entire application) changes to a smart contract. While upgrades to a smart contract's user interface (UI) can significantly change a user experience and expose new features, UIs are governed by traditional software maintenance. Our paper only considers the on-chain smart contract component, which is significantly more challenging to upgrade as it is on-chain and immutable under reasonable circumstances.

Related work. A lot has been written in the non-academic arena as technical blog posts []. MOst of section 3 is based on this information.

CREATE2 paper -¿ Measurements for only Metamoprhis, we do mostly proxy. "Using simple heuristics derived from the EIPS we found 223,873 following EIP-897, 0 following EIP-1167, 22,238 following EIP-1822, and 31,432 following EIP-1967." By comparison, our model is more generic.

USENIX paper -¿ tool for actually updating a contract, assuming the contract uses a delegatecall-based data seperation pattern. "Dissimilar Redundancy in DeFi" -¿ another tool.

Other researchers have measurement studies on Ethereum data but concern other aspects: tokens (two erc20), volnerabilities (10 papers), cost of opcodes, impacts of EIP-1559, xxx.

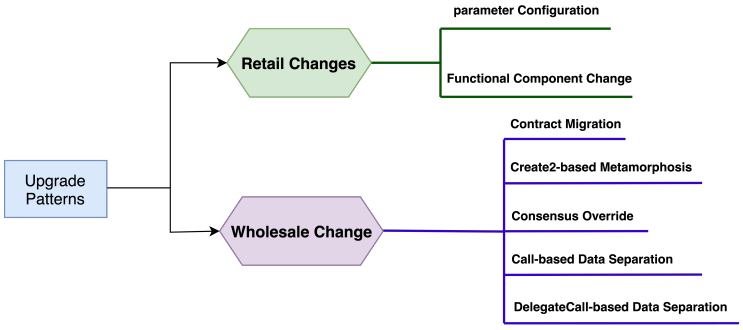


Fig. 1. Classification. Re-arrange to match ordering of text. Include section numbers of leaf nodes.

SELFDISTRUCT survey -¿ contract migration, clean up the old contract. Also useful for metamoprhisis but authors don't cover this

* we are measurement * no tools * broad set of upgrade pattern

TBD

3 Classification of Upgrade Patterns

A variety of upgradeability patterns have been proposed for smart contracts. Most leverage Ethereum-specific operations and memory layouts and are not applicable to other blockchain systems.

3.1 Parameter Configuration

We first categorize upgradeability patterns into two main classes: *retail changes* and *wholesale changes*. A pattern for retail change does not enable the replacement of the entire contract. Rather, a component of the contract is predetermined (before the contract is deployed on Ethereum) to allow future upgrades, and the code is adjusted to allow these changes.

The simplest upgrade pattern is to allow a system parameter, that is stored in a state variable, to be changed. This requires a *setter function* to overwrite (or otherwise adjust) the variable, and access control over who can invoke the function. For example, in decentralized finance (DeFi), many services have parameters that control fees, interest rates, liquidation levels, *etc.*. Adjustments to these parameters can initiate large changes in how the service is used (its 'tokenomics'). A DeFi provider can retain control over these parameters, democratize control to a set of token holders (*e.g.,* stability fees in the stablecoin project MakerDao), or lock the parameters from anyone's control. In Section 5, we dive deeper into the question who can upgrade a contract.

3.2 Functional Component Change

While a parameter change allows an authorized user to overwrite memory, a functional component change addresses modifications to the code of a function (and thus, the logic of the contract). In the EVM, code cannot be modified once written and so new code must be deployed to a new contract, but can be arranged to be called from the original contract.

One way to allow upgradable functions is deploying a helper contract that contains the code for the functions to be upgradeable. Users are given the address of the primary contract, and the address of this secondary (helper) contract is stored as a variable in the primary contract. Whenever thhis function is invoked at the primary contract, the primary contract is pre-programmed to forward the function call, using the opcode `Call`, to the address it has stored for the secondary contract. To modify the logic of the function, a new secondary contract is deployed at a new address, and an authorized set of individuals can then use a parameter change in the primary contract to update the address of the secondary contract.

The DeFi lending platform Compound uses this pattern for their interest rate models which are tailored specifically for each asset. The model for one asset can be changed without impacting the rest of the contract.

Upgradeable functional components need to be pre-determined before deploying the primary contract. Once the primary contract is deployed, it is not possible to add upgradeability to existing (non-upgradable) functions. It also cannot be directly used to add new functions to a contract. Finally, this pattern is most straightforward when the primary contract only uses the return value from the function to modify its own state. Thus, the function is either 'pure' (relies only on the parameters to determine the output) or 'view' (can read state from itself or other contracts, but cannot write state). If the function modifies the state of the primary contract, the primary contract must either expose its state variables to the secondary contract (by implementing setter functions), or it can run the function using `DelgateCall` if the secondary contract has no state of its own.

This upgrade pattern suggests a way forward for wholesale changes to the entire contract: create a generic 'proxy' contract that forwards all functions to a secondary contract. To work seamlessly, this requires some further engineering (sections 3.6 and 3.7).

3.3 Consensus Override

The two previous patterns enable portions of a smart contract to be modified. The remaining patterns strive to allow an entire contract to be modified or, more simply, replaced. The first wholesale pattern is not a tenable solution to upgradeability as it as only been used rarely under extraordinary circumstances, but we include it for completeness.

Immutability is enforced by the consensus of the blockchain network. If participating nodes (*e.g.,* miners) agreed to suspend immutability, they can in theory

allow changes to a contract's logic and/or state. If agreement is not unanimous, the blockchain can be forked into two systems—one with the change and one without. In 2016, a significant security breach of a decentralized application called 'the DAO' caused the Ethereum Foundation to propose overriding the immutability of this particular smart contract to reverse the impacts of attack. In the unusual circumstances of this case, it was possible to propose and deploy the fix before the stolen ETH could be extracted from the contract and circulated. Nodes with a philosophical objection to overriding immutability continued operating, without deploying the fix, under the name Ethereum Classic.

3.4 Contract Migration

The simplest wholesale upgrade pattern is to deploy a new version of the contract at a new address, and then inform users (and the web applications they use) to use the new version—called a 'social upgrade.' One example is Uniswap, which is on version 3 at the time of writing. Versions 1 and 2 are still operable at their original addresses.

Contract migration does not require developers to instrument their contracts with any new logic to support upgradability, as in many of the remaining patterns, which can ease auditability and gas costs for using the contract. However for most applications, there will be a need to transfer the data stored in the old contract to the new version. This is generally done in one of two ways. The first is to collate the state of the old contract off-chain and load it into the new contract (*e.g.,* via its constructor). If the old contract was instrumented with an ability to pause it, this can eliminate race-conditions that could otherwise be problematic during the data migration phase. The second method, specific to certain applications like tracking a user's balance of tokens, is to have the user initiate (and pay the gas) for a transfer of their balance from the old system to the new one.

3.5 `CREATE2`-based Metamorphosis

Is it possible to do contract migration, but deploy the new contract to the *same* address as the original contract, effectively overwriting it? If so, developers can dispense with the need for a social upgrade (but would still need to accomplish data migration). At first glance, this should not be possible on Ethereum, however a set of opcodes can be 'abused' to allow it: specifically, the controversial[1] `SELFDESTRUCT` opcode and the 2019-deployed `CREATE2`.

Consider a contract, called Factory, that has the bytecode of another contract, A, that Factory wants to deploy at A's own address. `CREATE2`, which supplements the original opcode `CREATE`, provides the ability for Factory to do this and know in advance what address will be assigned to contract A, invariant to when and how many other contracts that Factory might deploy. The address

[1] "Expectations for backwards-incompatible changes / removal of features that may come soon." V. Buterin, Reddit r/ethereum, Mar 2021.

is a structured hash of A's "initialization" bytecode, parameters passed to this code, the factory contract's address, and a salt value chosen by the factory contract.[2] Most often, A's initialization bytecode contains a copy of A's actual code ("runtime" bytecode) to be stored on the EVM, and the initialization code is prepended with a simple routine to copy the runtime code from the transaction data (calldata) into memory and return. Importantly, however, the initialization bytecode might not contain A's runtime bytecode at all, as long as it is able to fetch a copy of it from some location on the blockchain and load it into memory. In order for `CREATE2` to complete, the address must be empty, which means either (1) no contract has ever been deployed there, or (2) a contract was deployed but invoked `SELFDESTRUCT`.

Assume the developer wants to deploy contract A using metamorphosis and later update it to contract B.[3] The developer first deploys a factory contract with a function that accepts A's (runtime) bytecode as a parameter (which includes the ability to self destruct). The factory then deploys A at an arbitrary address and stores the address in a variable called codeLocation. The factory then deploys a simple 'transient' contract using `CREATE2` at address T. This contract performs a callback to the factory contract, asks for factory.codeLocation, and copies the code it finds there into its own storage for its runtime bytecode and returns. As a consequence, A's bytecode is now deployed at address T.

To upgrade to contract B, the developer calls `SELFDESTRUCT` on A. Mechanically, the consequences of `SELFDESTRUCT` on the EVM are only realized at the end of the transaction. In a followup transaction, the developer calls the factory with contract B's bytecode. The factory executes the same way placing a pointer to B in factory.codeLocation. Importantly, it generates the same address T when it invokes CREATE2 since the 'transient' contract is identical to what it was the first time—this contract does not contain contract A or B's runtime code, it just contains abstract instructions on how to load code. The result is contract B's runtime bytecode being deployed at address T where contact A was.

As it is concerning that a contract's code could completely change, we note that metamorphic upgrades can be ruled out for any contract where either: it was not created with `CREATE2`, it does not implement `SELFDESTRUCT`, and/or its constructor is not able to dynamically modify its runtime bytecode.

3.6 `CALL`-based Data Separation

To avoid migrating the stored data from an old contract to an upgraded contract, a contract could instead store all of its data in an external "storage" contract. In this pattern, calls are made to a "logic" contract which implements the function (or reverts if the function is not defined). Whenever the logic contract needs to read or write data, it will call the storage contract using setter/getter (aka accessor/mutator) functions. An upgrade consists of (1) deploying a new logic contract, (2) pausing the storage contract, (3) granting the new logic contract

[2] Specifically: $\mathsf{addr} \leftarrow \mathcal{H}(\texttt{0xff} \| \mathsf{factoryAddr} \| \mathsf{salt} \| \mathcal{H}(\mathsf{initBytecode} \| \mathsf{initBytecodeParams}))$

[3] "The Promise and the Peril of Metamorphic Contracts." 0age, Medium, Feb 2019.

access to the storage contract, (4) revoking access from the old contract, and (5) unpausing the storage contract.

An important consideration is that the layout of the storage contract cannot be changed after deployment (*e.g.,* we cannot add a new state variable). This can be side-stepped to some extent by implemented a mapping (key-value pair) for each primitive data type. For example, a new uint state variable can be a new entry in the mapping for uints. This is called the Eternal Storage pattern (ERC930). It however requires that every data type be known in advance, and is challenging to use with complex types (*e.g.,* structs and mappings themselves).

A variant of this pattern can introduce a third kind of contract, called a proxy contract, to address the social upgrade problem. In this variant, users permanently use the address of the proxy contract and always make function calls to it. The proxy contract stores a pointer (that can be updated) to the most current logic contract, and asks the logic contract to run the function using `CALL`. Unlike the functional component pattern (Section 3.2), the proxy will catch and forward *any* function (including new functions deployed in updated logic contracts) using its fallback function. With or without proxies, this pattern is very powerful, but instrumenting a contract to use it requires deep-seated changes to the contract code. As our measurements will show, it has fallen out of favour for the cleaner `DELEGATECALL`-based pattern (Section 3.7) that addresses the same issues with simpler instrumentation.

3.7 `DELEGATECALL`-based Data Separation

This pattern is a variant on the idea of chaining each function call through a sequence of three contracts: proxy, logic, and storage. The first modification is reversing the sequence of the logic and storage contracts: a function call is handled by the proxy which forwards it to the storage contract (instead of the logic contract). The storage contract then forwards it to the logic contract using `DELEGATECALL` which fetches the code of the function from the logic contract but (unlike `CALL`) runs it in the context of the contract making the call—*i.e.,* the storage contract. When upgrading, a new logic contract is deployed, the proxy still points to the same storage contract, and the storage contract points to the new logic contract. Since the proxy and storage contracts interact directly and are both permanent, the functionality of both can be combined into a single contract. It is common for developers to call this the 'proxy contract,' despite it being a combination of a proxy and a storage contract.

This pattern generally cleaner than using the previous `CALL`-based pattern because the logic contract does not need any instrumentation added to it. It is an exact copy of what the contract would look like if the upgrade pattern was not being used at all. However this does not mean the pattern in a turn-key solution. Each new logic contract needs to be programmed to respect the existing memory layout of the storage contract, which has evolved over the use of all the previous logic contracts. The logic contract also needs to be aware of any functions implemented by the storage contract itself—if the same function exists

in both the storage contract and the logic contract (called a function clash), the storage function will take precedence.

The main issue with function clashes is that the proxy contract needs, at the very least, to provide an admin (or set of authorized parties) the ability to change the address of the logic contract it delegates to. This can be address in four main ways:

1. Developers are diligent that no function signature in the logic contract is equal to the signature of this function in the proxy contract (note that signatures incorporate a truncated hash of the function name, along with the parameters types, so collisions are possible).
2. As found in the *universal upgradeable proxy standard (UUPS)* (EIP-1822): implement the upgrade function in the logic contract, which will run in the context of the proxy contract. Its exact function signature must be hard-coded into the proxy contract. Every logic contract update must include it or further updates are impossible.
3. As found in the *beacon proxy* pattern (EIP-1538): deploy another contract, called the beacon contract, to hold the address of the logic contract and implement the setter function for it. The proxy contract will get the logic contract address from the beacon every time it does a `DELEGATECALL`. The admin calls the beacon contract to upgrade the logic contract, while normal users call the proxy contract to use the DApp.
4. As found in the *transparent proxy* pattern (EIP 1538): inspect who is calling the proxy contract (using `msg.sender()`)—if it is the admin, the proxy contract catches the function call and if it is anyone else, it is passed to the proxy's fallback function for delegation to the logic contract.

A drawback of the entire `DELEGATECALL`-based pattern is that logic contracts need to be aware of the storage layout of the proxy contract. In a stand-alone contract, the compiler (*e.g.,* Solidity) will allocate state variables to storage locations, and using `DELEGATECALL` does not change that, however new logic contracts need to allocate the same variables in the same order as the old contract, even if the variables are not used anymore. This can be made easier with object-oriented patterns: each new logic contract extends the old contract (inheritance-based storage). Other options include mappings for each variable type (eternal storage) or hashing into unique memory slots (unstructured storage). The *Diamond Storage* pattern (EIP-2535) breaks the logic contract into smaller clusters of one or a few functions that can be updated independently, and each can request one or more storage slots in a storage space managed by the proxy contract itself.

3.8 Evaluation Framework

Table 1 summarizes the pros and cons of each upgradability pattern, omitting consensus override as it is only used in emergencies.

	Can replace entire logic (1)	Remove	Remove	Remove	No need to deploy a new contract to upgrade (2)	No need to migrate state from old contract	Remove	Function selector clashes risk	Storage clashes risk	No indirection	User endpoint address unchanged (3)	No downtime to upgrade	No need to instrument source (4)	Remove
Parameter Configuration				●	●	●	●			●	●		●	●
Component Change		●				●	●			○	●		○	●
Contract Migration	●		●				●			●			●	
Create2 metamorphosis	●		●				●			●	●	●	●	
Consensus Override	●		●				●			●	●	●	●	
Call-based	●					●								●
DelegateCall-based	●					●		●	●		●		○	●

Table 1. What does the square mean?

4 Finding Upgradeable Contracts

We now design a series of measurement studies to shed light on the prevalence of the various upgrade patterns. We exclude retail changes from our measurements as variable changes and external function calls are too commonplace to distinguish. We focus on wholesale patterns, and devote the most effort to finding contracts using the `DELEGATECALL`-based data seperatation pattern (Section 3.7) as these are the most widely used and there are various sub-types (UUPS, beacon, *etc.*). The other types of wholesale patterns are:

- **Consensus override:** Only 1 occurrence to date (the DAO attack []).
- **Contract migration:** Not detectable in code; relies on social communication of the new address.
- `CREATE2`**-based Metamorphosis.** This was measured by Frowis and Bohme [] in a broader study of all uses of `CREATE2`. They found 41 contracts between March 2019 and July 2021 that upgraded using this pattern.
- `CALL`**-based Data Separation.** We conducted a quick study of 93K contracts with disclosed source code [8]. We identified the Eternal Storage pattern using regular expressions and found 140 instances, the newest having been deployed over 3.5 years old. We conclude this pattern is too uncommon today to pursue a deeper bytecode-based on-chain measurement.

4.1 Methodology

As mentioned in classification section, the *DelegateCall-based* upgradeability approach consists of a storage contract (a.k.a proxy contract) and a logic contract (a.k.a implementation contract). The proxy contract is a simple type of smart

contract in which there is a *Fallback* function. Fallback is a function inside smart contracts that do not have a function name. If a user sends a transaction to a contract to call a function that does not exist, it will pass into the fallback function, and the logic inside the fallback function will be executed. Inside the fallback function of a proxy contract, there is a delegate call to the address of *implementation contract* (we call it *Target address* in the rest of the paper) which passes the whole data of the transaction to the implementation contract without altering it.

All proxy contracts have the above structure. However, *Upgradeable proxy contracts* should have another extra condition as well. The agent who is responsible for changes in the smart contract (a.k.a *admin*) must have the ability to change the target address. If a proxy does not have this condition, the contract delegates the data into a fixed implementation contract for the rest of its life. So, this type of proxies is not upgradeable. There are a bunch of patterns that follow this structure (e.g., Minimal Proxies [7], Delegate call forwarders [4], etc.), which we call *Forwarders* in the rest of the paper. So, for upgradeable proxies, the target address must be *changeable*.

To find the proxy contracts in Ethereum, we need to collect transactions and all information regarding those transactions. To collect the transaction details, we need to replay the transaction and collect the data of execution of the transaction. Ethereum full archival node has a method, *trace_transaction*, that gives the traces [4] of the transactions executed on the specific block. We used this method to have transaction traces and find the transactions in which a delegate call happened. Each transaction trace may consist of several sub-traces (a.k.a, actions). If the data of two consecutive sub-traces of a transaction are equal and a delegate call is in the second sub-trace, it shows that the transaction passes the fallback function. Because if any other function in the contract is called (other than fallback), then the first four bytes of the data will be changed. Also, a delegate call in the fallback transferred the whole data without altering it, which means the contract is a proxy contract.

As discussed above, these proxies can be forwarders or upgradeable proxies. To find upgradeable proxies, we should filter them by checking whether the target address variable is changeable or not. Three general standards are proposed to change the target address of a proxy: Beacon proxy, Regular proxy, and Universal Upgradeable proxy. As discussed in the classification part, the target address in beacon proxies comes from an external call to another contract named *Beacon Contract* (put reference to beacon). So, to find upgradeable beacon contracts, we first check if the target address comes from an external call to another contract. If yes, we should check the callee contract to find out if the target address inside the beacon contract is changeable. If the target address is changeable, the proxy contract is a *Beacon Proxy* contract, and the admin of the beacon contract can change the target address inside it to upgrade the proxy contract.

If the address does not come from an external call, we check if there is any function inside the proxy contract that the admin can call to change the target

[4] Parity VM transaction trace

address and upgrade the system. This is the most tricky part in our methodology to find out if a function inside the contract gives the admin the ability to change the target address, because there is no general pattern for upgrade function. The process of upgrading the target address can happen by having a single function for the change, having a chain of functions that leads to the change of target address, or even not changing target address by setting a new amount to the storage slot in which the target address is kept. If that function is found, we mark the proxy as an upgradeable proxy contract. The process is divided into two main parts; 1) finding the target storage variable regarding the target address, and 2) checking if there is an assignment to that specific storage variable inside the contract.

Finding storage variable (slot) of the target address. We use bytecode decompiler named *Panoramix decompiler* [5] to decompile the bytecode of the contract into well-formatted python language codes. Then check to find the line of the code in which the delegate call happened and pick the target of the delegate call. We find the variable name or a storage slot of the target address, which is our goal in this part.

Checking for assignment. Now that we have the decompiled code and variable name (or storage slot) of the target address, we parse the code and check if an assignment to that variable/slot happened in any function in the contract. If any assignment is found, we should be sure that the other variable assigned to the target address variable comes from the input of that function. If these conditions are satisfied, there is a function inside the contract that can change the target address and upgrade our system (upgrade function).

So by applying the first filter, we find the storage variable/slot of the target address and then check if it is changeable or not. If the target address is changeable, we mark the proxy as an upgradeable proxy contract. If there is no way to change the target address inside the proxy, we pass it to another final filter. There is another way of implementing upgradeable contracts named *Universal Upgradeable Proxy Standard (UUPS)* that is discussed in the classification section (add reference to the UUPS). In this method, the target address is changeable using the implementation contract. So to filter and find them, we check the implementation contract to find out if there is any function inside the implementation contract by which the admin can change the target address. If yes, then our proxy is a UUPS proxy contract. Otherwise, the proxy is not upgradeable. The first step here is to find the storage slot of the target address inside the proxy contract. Then we decompile the bytecode of the implementation contract and check to find if any assignment to that storage slot happened inside the implementation contract. The process of finding the assignment is very similar to how we checked the proxy contract to find the assignments. So, if a function is found that gives the admin a chance to write a new amount to the storage slot regarding the target address, the admin can call that function using the proxy to change the target address and upgrade the system. In this case, we marked the proxy as a UUPS proxy contract. All the remained proxy contracts

[5] `https://github.com/palkeo/panoramix`

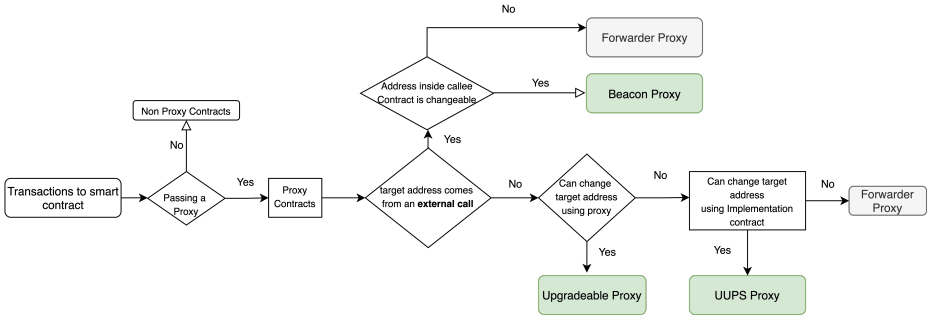


Fig. 2. Flowchart of the Process

are marked as non-upgradeable proxy contracts. The whole process is depicted in figure 4.1. For a detailed explanation of the methodology and implementation, check the appendix.

Results Having access to an Ethereum full archival node, we have collected transaction traces of transactions included in 2,064,595 blocks of Ethereum blockchain, starting from block number *#10800000* to *#12864595*. It covers transactions on the Ethereum blockchain from *Sep-05-2020* to *Jul-20-2021*.

Applying our methodology gives us *1,427,215* unique proxy contracts. However, a bunch of these proxies is using shared implementation contracts. We decide to weed out the proxy contracts that share the same implementation contract; however, two different Dapps may use the same implementation contract. The reason for this decision is that there are some Dapps like opensea [6] that create proxy contracts for each of their user, and these proxies share the same implementation contract (it will reduce the redundancy). So after filtering proxies with the same implementation contracts, we come up with *13,088* contracts. Afterwards we filter *Forwarder Contracts* from our dataset and then apply the first filter to check if the proxy contract has a method to change the target address. This filter finds *7,470* regular upgradeable proxy contracts. On the other hand, checking the remained proxy contracts (the proxies that neither forwarders nor regular upgradeable proxies) by applying the next filters explained in the methodology section, *403* upgradeable proxy contracts found that follow Universal Upgradeable Proxy Pattern and also *352* unique beacon proxy contracts.

At the end we find *8,225* unique upgradeable proxy contracts with unique implementation contract. We randomly sampled 150 contracts from these contracts and manually checked them, and all of them were upgradeable proxy contracts. On the other hand, we sampled 150 contracts from those marked as non-upgradeable and checked them manually. Between these 150 contracts, just two were upgradeable. Our model did not catch these contracts because a failure

[6] `opensea.io`

happened on the decompiler to decompile the implementation contract code, so our assignment checker detector could not catch them. The reason is that implementation contracts are much larger in contrast with the proxy contract itself.

5 Finding Admin of the Proxies

This section proposes a novel way to find the admin of the proxy contract (the agent responsible for upgrading the proxy contract) and classify them based on their account type. We apply the method to the dataset of proxy contracts we provided from the previous section 4. We also shed light on the risks regarding the number of decision-makers who have the authority to change the whole logic of the Dapps that are using a proxy contract.

The question we will answer in this part is who can upgrade the system? There should be an agent who decides on the upgrades of the system. Generally, there are three main types of admins for upgradeable smart contracts which we describe below; an Externally Owned Account (EOA), Multi-Signature wallets, and Governance schemes. EOA and multi-signature schemes add centralization risks to the system because a limited number of private keys can change the system's whole logic. Based on *The State of DeFi Security 2021* [5] report by Certik [7], **Centralization** is the most common attack vector of the hacked DeFi projects. In further part of this section, we will explain a real-world incident to show how using an EOA as the admin of an upgradeable proxy contract may lead to loss of funds.

Externally owned Address(EOA): This most centralized way to deal with admin is to have just one private key controlling the upgrades. Using EOA as admin is the fastest way to respond to incidents, but in case of a malicious admin or a private key compromise, the whole funds are at risk.

Multi-Signature Wallet: A *m out of n* Multi-signature wallet is a smart contract that can execute a transaction only if m number out of a specified n EOAs agree and sign the transaction. Using multi-sig as admin is a better way to the upgrade's decision-making than using an EOA. However, it may not be decentralized. The problem here is that the Ethereum accounts are pseudo-anonymous, and the identity behind addresses is not recognizable. So, the malicious developer team can keep at least m signatures out of n in their hands, and in the desired time, they can upgrade a system to a malicious version and steal the funds. Also, there are some types of governance voting which is known by *Off-chain Governance Schemes* in which users who hold governance tokens can signal their votes on proposals in an off-chain tool like *Discord* or *Snapshot* and then another agent which is a part of multi-signature schemes can put the results on-chain and execute the actions if needed. We consider the off-chain governance in the Multi-sig category because, in the end, these multi-signature wallet owners are the only on-chain agents responsible for changing the system, and there is no way to enforce the signers to reflect the off-chain agreement results to the smart contract.

[7] `certik.com`

On-Chain Governance Voting: The most decentralized way to decide on a system change is to use a decentralized voting scheme. 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 is some critique to this method. Governance by voting has an inherent time delay to the upgrading process. This raises a problem when the system needs an instant upgrade (e.g., responding to an incident) [8] It is also not cost-efficient for the voters because all token holders must send a transaction for voting and pay a network fee. The other problem with this method is a fair distribution of the tokens. If the governance tokens are not distributed fairly, and the majority of tokens are granted to a limited number of users, they can change the voting results to their desired outcome.

5.1 Exploring Admin Types

As described above, proxy contracts may have three types of admin: EOA, Multi-Sig, and Governance Contract. In EOA and Multi-sig types, a person or a limited number of persons may decide to take control of the system. The risks regarding each admin type discussed above bring us to find the admin types of all proxy contracts we found in our first analysis.

In the previous section 4, we gathered *7,470* regular upgradeable proxy contracts. In this part, we try to find the admin of each proxy contract and recognize the type of the admin (i.e., EOA, Multi-Sig, Governance). Here we describe our methodology of finding the admin addresses and their types. The process can be divided into two main parts: finding the admin account's address and finding the admin type (EOA, Multi-Sig, or decentralized governance).

Finding the Admin Account's Address. EIP-1967 [**?**] suggested specific arbitrary slots for upgradeable proxy contracts to store *Admin address*[9]. So, we first check this specific storage slot and if it is non-zero, the address that is saved inside it, is the admin address. However, not all proxy contracts use the EIP-1967 suggested storage slot. So, for non-EIP-1967 proxies, we propose a way to find the storage slot in which the admin address is stored. The process is very similar to how we found storage variable (slot) of the target address in 4. We first find the function in which the admin can change the *target address* (upgrade function). This function is critical and should only be called by the admin. It means there should be access control to check the caller of the function. We find this access control check, and conclude that the address that is checked inside, is the admin address.

Finding the admin type. Having the admin address, we can check if the account is an EOA by just checking if the account consists of a code or not[10].

[8] This arises the need for another mechanism to quickly fix bugs and upgrade the system in the event of incidents in conjunction with the voting process (e.g., Global shutdown in MakerDAO).

[9] Storage slot 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103

[10] using the *eth_getCode* method for the admin address

If the contract does not contain code, the admin is an EOA and if it contains code, the admin is another smart contract which can be a multi-signature wallet. The most widely used multi-signature wallet is Gnosis Safe[11] wallets. We automatically checked if the code of the admin address is the Gnosis multi-signature wallet, and if yes, we marked them as Multi-Signature admins. After picking Gnosis safe wallets, we manually checked 10% of the remaining addresses to find any other patterns for multi-signature wallets and found other patterns (e.g., MultiSignatureWalletWithDailyLimit, etc.) and added them to the dataset as well.

2,534 contracts between all *7,470* proxies we have, are using another proxy contract as their admin, which is known as *Admin Proxy*. Admin proxy contract adds another layer of indirection. In this case, the real admin (owner of the admin proxy) sends their desired transaction to the admin proxy, redirected to the primary proxy, and getting executed. So, we attempt to find the admin proxies among the admin addresses and then find the owner of these proxies. The owner's address is the real admin account that can upgrade these systems. Now that we have the admin address, we do the same processes as before to find the EOA and Multi-Signature types. The remaining proxy contracts which are not marked as EOA or Multi-Signatures are marked as decentralized governance or unknown. We add unknown tag, because some of the contracts were using undefined new patterns as their multi-signature contracts and our model has false negatives to detect the multi-signatures. For a detailed explanation of the methodology and implementation, check the appendix.

By applying the above methodology in our dataset, the results show that out of **7,470** proxy contracts, **3,558** are controlled by an EOA address, **988** are controlled by a multi-signature wallet, and **2,924** addresses are either decentralized governance based admins, or unknown type.

The results show that a single EOA account controls 48% of the proxy contracts and 61% by an EOA or Multi-Signature wallets control. This is a significant risk to the Ethereum ecosystem because, in these contracts, one or a limited number of persons can decide to change the whole logic of the contract and take control of the funds under the custody of the contract. Bent Finance incident [6] is a real-world example of what may happen to all these proxy contracts. Bent Finance[12] is a staking and farming platform. They are using *Transparent Upgradeable Proxy* pattern in their system. The admin of the proxy was an EOA at the time of the incident. The malicious developer deployed a new implementation contract[13] in which it provides a huge amount of token to the malicious actor's address [14]. Afterward, the attacker upgraded the proxy to the malicious implementation contract, and by doing that, a considerable amount of tokens were assigned to the attacker's address. Once the balance was transferred to the attacker, they upgraded the proxy to the latest non-backdoor version to hide the

[11] `https://gnosis-safe.io/`
[12] `https://app.bentfinance.com/`
[13] `https://etherscan.io/address/0xb45d6c0897721bb6ffa9451c2c80f99b24b573b9`
[14] 0xd23cfffa066f81c7640e3f0dc8bb2958f7686d1f

exploit. Having a massive amount of tokens, the attacker drained liquidity from Curve Finance protocol, a decentralized exchange. The same scenario may happen to all other upgradeable proxy contracts which use EOA or multi-signature wallets as their admin.

6 Concluding Remarks

* Which update pattern is the best? * Immutabilty is not a reality (cite Angela Welch?) * One approach is too find specifics of upgrade frameworks (libraries, implementations) that are commonly used, our approach is pattern-based and cross cuts different implementations. We demonstrated this to a leading blockchain firm in finding a broader set of contracts potentially volunerable to a bug in a commonly used UUPS libraries. * EOA -¿ the risk is underestimated -¿ stories -¿ that is a mess -¿ not centralized * Layer 2

References

1. Barros, G., Gallagher, P.: Eip-1822: Universal upgradeable proxy standard (uups) https://eips.ethereum.org/EIPS/eip-1822
2. of Bits Blog, T.: Breaking aave upgradeability https://blog.trailofbits.com/2020/12/16/breaking-aave-upgradeability/
3. of Bits Blog, T.: Contract upgrade anti-patterns https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/
4. Buterin, V.: Delegatecall forwarders: how to save 50-98contracts with the same code https://www.reddit.com/r/ethereum/comments/6c1jui/delegatecall_forwarders_how_to_save_5098_on/
5. Company, C.: The state of defi security 2021 https://www.businesswire.com/news/home/20220113005054/en/CertiK-Releases-2021-State-of-DeFi-Security-Report
6. Finance, B.: Bent update https://bentfi.medium.com/bent-update-12ae69a41dc6
7. Murray, P., Welch, N., Messerman, J.: Minimal proxy contract. EIP-1167 (2018)
8. Ortner, M., Eskandari, S.: Smart contract sanctuary https://github.com/tintinweb/smart-contract-sanctuary
9. Palladino, S.: Security advisory: Initialize uups implementation contracts https://forum.openzeppelin.com/t/security-advisory-initialize-uups-implementation-contracts/15301
10. Palladino, S.: Uupsupgradeable vulnerability post-mortem https://forum.openzeppelin.com/t/uupsupgradeable-vulnerability-post-mortem/15680

A Evaluation of different methods

In this section we compare and evaluate different methods discussed in previous section and explain the consequences regarding each method to the users and developers of Dapps.

A.1 Criteria

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. In this part we pencil out these criteria and evaluate different methods based on these criteria. In this part we describe and specify what it means that each row of our table receives a check mark (✓), double check marks (✓✓), square (⊠) 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.

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.

Replace entire state An upgradeability method in which the upgrader can replace the entire state in 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.

No need to deploy a new contract In 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 for the process of upgrade receive a check mark (✓).

No need to migrate state from old contract In some patterns, there is no need to collect data from the old version and push it to the new contract which receive a check mark (✓). On the other hand, patterns which required to migrate data from old version receive nothing.

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.

DelegateCall opcode Risks Some of the upgradeability methods utilize *Delegate call* opcode. Using this opcode bring complexity to the system and needs more security considerations. These security considerations are categorized into two main risks which we explained before; function selector clashes and storage clashes.

Function Selector Clashes Risk Upgradeability methods in which the developer should take care of function selector risks receive check mark (✓), otherwise receive nothing.

Storage Clashes Risk Upgradeability methods in which the developer should take care of storage clashes risks in two contracts receive check mark (✓), and the methods that the developers must consider this risk and deal with it in more than two contracts receive double check marks (✓✓) otherwise receive nothing.

No indirection Indirection happens if the first external message need be forwarded from a contract to another. Upgradeability methods that do not need any indirections receive a check mark (✓). An upgradeability pattern that contains indirections which 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 (and not all transactions to the contract) need indirection receive square (⊠).

User endpoint address not changed In some upgradeability methods, after the upgrade process, users must call a new contract address to use the Dapp. It is equivalent to having 2 different Dapps at the end of the upgrade. Alice uses a DApp X which uses one of the upgradeability patterns at address A before the upgrade. After upgrade, she may be unaware that upgrade happened and use the previous address (receive check mark (✓)) or she may need to use address B instead which receive nothing.

Downtime in upgrade events Patterns which we will have a downtime of the Dapp in the upgrade event receive check mark (✓) otherwise it receives nothing.

No need to change code to add the upgrade pattern Upgradeability patterns in which the developers do not need to change any part of the original code to add the upgrade method receives nothing. The methods in which the developers do not need to change the whole code but should add a proxy contract or change just one component of the system receive square (⊠) and patterns in which the devs should change the whole code to add upgradeability receive check mark (✓).

Need to change a state variable Upgradeability patterns in which the upgrader should change a state variable on the upgrade process receive a check mark (✓) otherwise it receives nothing. Two scenario could happen in this case, changing a variable as a upgrade parameter or changing an address variable which is a pointer address in the system.

A.2 Consequences

In this section we discuss about the consequence of each upgrade methods regarding the criteria we mentioned in the previous part in users and developers that want to use the upgradeability pattern or uses a Dapp that uses one of the mentioned patterns.

A.3 Speed of an Upgrade

Upgrade events of a Dapp consists of two different processes. First a way to come to an agreement to a change, and then a way to implement and execute the change. The First part depends on the reason behind the upgrade. If the upgrade is to patch a bug, then the process to come into agreement is very fast but if the goal behind upgrade is to add new functionality or change a logic, it usually starts with a proposal and after the discussion if the agent that responsible for the decision agree with the proposal, the execution part will be started. We won't discuss about the first process because it depends on the type of agent that is responsible for the upgrades. The types of agents will be discussed in details in further sections which are EOA, Multi-sig and Decentralized Governance Voting system.

After coming into agreement about the change, the speed that the upgrader can implement and execute the upgrade depends on three main criteria discussed above; *Need to migrate state from old contract* , *No need to migrate state from old contract*, and *having a downtime in the upgrade process*.

Parameter change method is the fastest way to execute the upgrade because there is no need to deploy a new contract, and no need to migrate state and no downtime in the system. *Component change* method change is not as fast as Parameter change method but faster than other types because the upgrader needs to deploy a specific smart contract which is a small component of the system and also update an address variable inside the main contract that points to that specific component and change it to the address of the new version of that component. But there is no need to migrate data and there is no downtime needed for this upgrade method.

Migration method has a slow upgrade process. The reason is that the upgrader needs to deploy a new contract and also the upgrader or users should transfer the data from old version to the newer version. In most Migration processes the developer team deploy a *Migrator* contract and users should use this Migrator contract to withdraw their funds/data from the previous version and move it to the newer version. But, there is no downtime in the Dapp and no need to change a state variable.

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 implement and deploy the *whole logic* contract to the blockchain and then change the pointer addresses inside the storage/proxy contract to the newer version. *Diamonds* is very similar to these two but because Diamonds is a modularized pattern, in the event of upgrade we need to just implement and deploy one module which is related to the functions we want to change. So, the speed in Diamonds is similar to Component change methods. On the other hand these two approaches are faster than *Migration* because as mentioned before, there is no need to migrate data. There is no downtime in these methods.

Metamorphic method is the slowest way to upgrade a system which uses this method because similar to the Migration plan there is a need to deploy a contract and migrate the state to the newer version but there is a difference between these two. In Metamorphic method the upgrader first should *Self-Destruct* the previous version in a single transaction and after that transaction send a contract creation transaction to deploy the newer version. Because self destruct happened at the end of the transaction, the process of upgrade happens on two different transactions which is a downtime to the system. This downtime could be a gap between order of the two transaction in a single block or could be gap between blocks that these two transactions included into blockchain.

A.4 Cost of Upgrade

One of the main differences between upgradeability approaches is how much does the upgrade process costs for the upgrader and users. The cost of upgrade mostly depends on three criteria explained above; need to deploy a new contract, need to migrate a the state to newer version, and need to change a state variable.

Parameter change method is the cheapest method in the upgrade event because there is no need to deploy a new contract or migrate data but just need to change a state variable.

Component change is in the middle because there is a need to deploy a new contract (however it is cheaper comparing to methods in which we should deploy the whole logic), and change an address pointer variable but there is no need for data migration.

Migration plan is very expensive in the upgrade event because we need to deploy a new contract and migrate the data from the old version which is very expensive. But no need to change any state variables.

Call-based and *DelegateCall-based* are very similar to each other in the cost of upgrade which is more expensive than component change but cheaper than migration. In both the upgrader must deploy the a contract containing the whole logic and change an address pointer inside storage/proxy contract. But there is no need to migrate the whole data.

Diamond's cost of upgrade depends on the upgraded needed for the system. It is very similar to Delegate-based pattern but if there is a need to change some functions that are not in one module (faucet) of the system then we need to deploy more than one smart contract in the event of upgrade and so it is more

expensive than doing upgrade comparing to Delegatecall-based pattern (however we do not need to deploy the whole logic but deploying a contract to ethereum blockchain is the most expensive action we have in EVM).

Metamorphic method is the most expensive method we have because we need to deploy a new contract, migrate data to the newer version and also we need to self destruct the previous version before the upgrade event which adds cost to the upgrade process.

A.5 Gas overhead for users

Sometimes in upgradeability patterns, we have a tradeoff between adding a feature to the pattern to improve it and increasing the cost for users that want to interact with our Dapp.

In patterns that needs indirection, such as *Call-based*, *Delegatecall-based*, *Diamonds* and *Component change* pattern we are adding a cost to the users because for all or some of the transactions to the Dapp, our system needs to forward the calls to another contract using Call or Delegatecall opcode to the users. Also in *Delegatecall-based* and *Diamond* pattern to mitigate the function selector clashes or storage clashes we need to add some checks to our code which also increases the cost of interacting with the Dapp. We can compare all patterns like UUPS or transparent proxies in term of gas overhead here.

Also there are some other ideas that addresses some limitations of a upgradeability pattern but increases the cost for users. For instance in *Call-based* approach one of the problems is that after upgrade users should use a new address for using the Dapp but adding a *Registry* contract can help to mitigate this. Using Registry contract, all other contracts should ask the registry to find out the latest version of the contract and then calls to the newer version which adds a gas cost to the users.

A.6 Useability

Upgradeability patterns differ in term of Useability and it depends on three criteria explained above; *User endpoint address changed*, *Need to migrate state from old contract* and *Downtime in upgrade events*.

Patterns in which the endpoint address is changing after upgrade event, *Migration* and *Call-based* is not user friendly because each time that the upgrade happened, the user must use the newer address. So make awareness about theis change is a hard action and need to socially interact with the users and make them aware of the change. We have two main type of users in the Dapp ecosystem, normal user or another smart contract (Dapp) that uses our system. Regular users which uses the official interface (website) of the project may do not sense any changes but users that work with the smart contract directly or via their own interface or other Dapps that uses the smart contract must have a way to upgrade the address they uses to use the newer version and if they did not implement a way to upgrade this address then their Dapp will face problems. So these patterns are make problems for composablility of the ecosystem.

In *Migration* plan, in most cases of upgrade events the users are responsible for the migration of data. For instance, the user must withdraw the fund and use a *Migrator* contract to push the data into the newer version which add costs to the user and it is not user friendly. This is one reason that make the Migration plans very hard because some users are not doing the process of migration and stay on the previous version which is like having a fork for the Dapp in side of the Dapp team. We see this happened on Uniswap V2 and V3.

In *Metamorphic* pattern as mentioned before there is a downtime during the upgrade. So users cannot work with the Dapp on that exact time which is not user friendly.

A.7 Dealing with two different new versions

In *Migration* and *Call-based* pattern we will come up with two different Dapps. So a decision must be made for the previous version. One possible choice could be shutting down the old version. It can be done by self-destructing the old version, or by pausing mechanism which will be explained in further sections. In migration plan it is not regular to stop the previous version because in most migration plans, users are responsible to move their funds and data from the previous version to the new one and we cannot force them to do that, so we cannot stop the smart contract.

The other option could be having a mechanism that after the upgrade, all calls to the previous version just be forwarded to the newer version which add costs and have some limitations like we cannot call the new functions defined in the newer version using the old version. This option is doable in Call based patterns. The other problem of this option is that if we upgrade a system more than one time then the calls to the first version should be indirected through lots of contracts to reach to the newer version. Also it adds complexity because developers must maintain more than one contract [3].

A.8 System Complexity

Using upgradeability patterns will add to complexity of our system but the degree of complexity varies and depends on the pattern. *Parameter Change* method does not change the system in general but just adding a mechanism to change pre-specified variables in the system. The most important issue about the Parameter change method is that the developer team must limit the boundary of these parameter for the security of the system. For instance in MakerDao platform, Stability fee is changeable but if this variable be changed to %100 then the whole system will be halted.

Component Change pattern is very similar to the Parameter change but here a whole component could be changed and finding the safe boundary of changes and limiting this boundary is a bit harder.

Migration plans for upgradeability does not change any complexity to the system because we do not need to change any part of system to add this type of upgradeability to it. The only important issue regarding this pattern is that we

must be sure that there is a way to collect data from the old version like having getter functions for reading data and also having a withdraw function for users to collect data and funds from previous version and push or deposit it to the newer version.

Using *Call-based* patterns adds higher degree of complexity to the system compared to previous patterns. As discussed before in this pattern we must be sure that the storage and logic contract is divided and there is not any storage variable inside the logic contract. This is one of the main security issues that found in the Dapps using this pattern regarding Trail of Bits company reports [3]. As mentioned before to add a way to storage contract to define new variables, developers uses the Eternal Storage pattern for their storage contract which is very hard to apply for complex data structures in Ethereum such as mappings or structures. This is another source of complexity using Call-based pattern.

Delegate-call pattern adds complexity to the code because of using *Delegate-call* opcode in its logic. As mentioned above because of using this opcode, the developer should take care of storage clashes and also function selector clashes. Other than these two there are some other limitations and risks of using this patterns. For instance, we cannot have a *Constructor* function on the logic contract (implementation contract) because constructor functions is used to initialize specific variables at deployment time and if we have a constructor inside the logic, then storage of implementation contract will be changed and not storage of proxy contract. To mitigate this problem we can add a regular function named *Initialize* function inside the implementation and make sure that this function can be called *once* to act just like a constructor function. Using initialize function brings some security risks that we will explain in further sections.

Diamonds pattern is very similar to the Delegate-call based patterns and have the same risks but this pattern is more complex because here we have different implementation contract for a single proxy and we should be sure that all of these implementation contracts share the same storage layout otherwise we will have a storage clash problem.

Metamorphic pattern is proposed recently and not well-tested yet. There are some risks to this pattern as well. We should be sure that we have a mechanism to self-destruct the contract. Otherwise if we cannot self-destruct the current contract we cannot redeploy a new version and so our contract won't be upgradeable. The other important issue related to Metamorphic pattern is that the developer must know that each time they want to upgrade the system the whole storage will be wiped out and need to re-initiate the whole state after re-deployment.

B Implementation detail of Finding The proxy contracts

Implementation In this section we describe the processes of finding upgradeable proxy contracts explained in the previous section in detail of implementation and execution. Each block of Ethereum blockchain consists of transactions that processed in that exact block. To get all transaction details we need to replay the transaction and collect the data of execution of the transaction. Ethereum full archival node has a method, *trace_transaction*, that gives the *Parity VM transaction traces* of the transactions executed on the specific block. This transaction traces are composed of *actions*. Each action gives data about each specific part of the execution of the transaction and consists of the opcode instruction that is executed on that action and the input data relevant to the that instruction and also information about how it manipulates the state, memory or stack of the Ethereum Virtual Machine (EVM).

Having the transaction traces for each block, we will search to find actions that consists of *callType*. When an action has callType field it means that there is a call happened in that specific action, which means one of *Call,Static Call* or *Delegatecall* happened in that action. we picked actions that have *callType* and this call type is *delegate call*. Also each action has an *input* element for messages that shows the call data (the data that is used for the calling the callee address). If the input element of the selected actions are equal to the input element of their previous action with the same transaction hash, it means the transaction passes a fallback function which delegate calls it to another contract. Because the input is not changed after the call it means that it passes a fallback function because the function selector (first four byes of the input) has not been changed which means the called function does not exists in the contract, so the fallback is called. Also the delegate call in the action shows that inside that fallback function there is a delegatecall that passes the whole message data to the target address contract. So, these contracts meet first condition described in the methodology part and these contracts are proxy contracts. We collect *From address*,*To address* and the *transaction hash* of these picked actions and pick the unique from addresses of this data. From address is the address of sender of transaction (proxy contract addresses), to address is the address which the proxy sends the transaction into (implementation contract addresses) and transaction hash is the transaction identifier that is used in Ethereum blockchain.

As discussed before, these selected contracts are proxy contracts and not necessarily upgradeable proxy contracts. So, we need to filter the *forwarders* from proxy contracts to find upgradeable proxy contracts. As mentioned in methodology section, the upgradeable proxy contracts must satisfy a condition; the admin of the contract should be able to change the target address. In other words, we should check if the target address inside the proxy contract is changeable or not.

There are five regular ways for the situation that the target address is fixed and not changeable on the contract:

1. The target address is hardcoded in the contract without assignment to any variables

2. The target address is saved in a constant variable type
3. The target address is saved in an immutable variable type and the deployer sets it via a constructor function
4. The target address is defined in a storage variable, but there is no function or a way to change this address after contract deployment (neither in proxy contract nor in implementation contract)
5. The proxy contract grabs the target address each time by calling another contract and there is no way on the callee contract to change this address

In the first three situations, the target address amount will be appeared on the runtime bytecode (the bytecode that is saved in the blockchain after deployment) of the smart contract. We can get the bytecode of each proxy contract that are collected from previous part using the *eth_getCode* method of a Ethereum archival node and check whether the target address is appeared on the bytecode or not. To have the target addresses we just need to collect The *To addresses* that we collected on the previous part which are supposed to be the implementation addresses because the proxy contract delegate calls into these addresses. So we check if the target address is appeared on the bytecode of the proxy contract. If yes, these proxies are not upgradeable proxies and they are forwarder contracts because the address is hardcoded on the bytecode and cannot be changed after deployment. Finding the forwarders, we removed them from the list of the proxy 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 if there is a way to change the target address after deployment or not. We will describe the filter in the later part of the paper, but for now let's assume that we have a module named *Assignment Checker* that checks the code and tells us that whether there is a function inside the contract that the caller can change the *target address* using the function. If yes we mark the contract as an upgradeable proxy contract because there is a way in the proxy contract to change the target address and point it to the new implementation contract and upgrade the system.

For the situation 5, we check the proxy contract to find whether each time before the delegate call, the target address inside the proxy contract is coming from an external call to another contract. If yes the we check the callee smart contract (a.k.a Beacon contract) that if the target address is changeable inside that contract or not. If it is changeable we mark the proxy as *Beacon Proxy* otherwise it is a *Forwarder*.

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 [1] that described in the previous sessions. In this type of proxy contracts, the target address can be changed using the implementation contract and not the proxy itself. As mentioned before the code of the implementation contract is executed 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 target address, we can upgrade the system by calling that function of implementation contract through proxy itself. We couldn't catch this type of upgradeable contracts by the previous processes because we just checked if there is a function inside the proxy contract itself that can change the target address.

We can tackle this problem by first finding the storage slot of the target address inside proxy contract. Then by using the *Assignment Module* to check if there is a function inside implementation contract which gives the admin right to change the target address. So first we should find the storage slot of the proxy contract in which the target address is saved. There are two EIPs out there that makes it easy to find these storage slots that suggested to be used to save target address for upgradeable proxies; EIP-1967 [15] and EIP-1822 [16], suggested randomly selected storage slots for target address to mitigate the overwriting on these slots. The UUPS contracts are usually using one of these two storage slots to save their target address.

The more general way to do this is not using these proposed slots from these EIPs for this process. The general way is to find the storage slot from the proxy contract and then check if admin can change this specific storage slot inside the implementation contract. But this way is not doable in large scale because in *Assignment checker* module we need to decompile the bytecode which is time consuming for contracts that have a large bytecode. Implementation contracts are usually large pieces of codes. So, it is not possible to do this process in a large-scale for all proxy contracts that are found from the previous parts. This is the reason that we limit our filter to specific storage slots proposed in EIP 1967 and EIP 1822.

So for this part we use the remained proxy contract addresses from the previous part. First we select the contracts that their target address is saved on the mentioned storage slots and if the target address is saved on the slot, check the implementation contract to find variable regarding that storage slot defined in the implementation contract. Afterwards check if the variable is changeable in a function in implementation contract. If it is changeable, it means that the admin can upgrade the system using a function on the implementation contract, and so the proxy is an UUPS contract.

The whole process is depicted on figure B.

B.1 Assignment Checker Module

As mentioned in the previous part, we need a module to check whether the admin can change target address on the proxy contract,using a function in the proxy contract, implementation contract or beacon contract. For this purpose the module must get the *Bytecode* of the proxy,implementation or beacon address as input and find the variable name and also its storage slot of the target address. Then checks to find out is there any function inside the contract that gives the admin the ability to change the target address.

[15] 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc
[16] 0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7

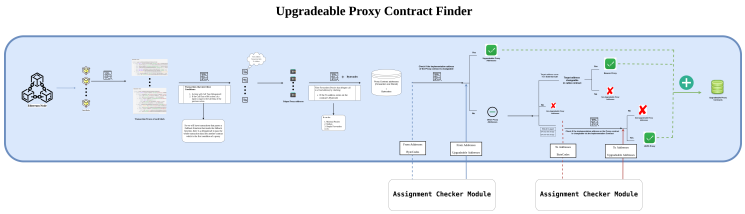


Fig. 3. Upgradeability Proxy Contract Finder

We use bytecode decompiler named *Panoramix decompiler* [17] to decompile the bytecode into well-formatted python language codes. The decompiled code gives us all storage variables of the related contract and the storage slots of those variables 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 its name is fallback is the *fallback* function of the contract. So we will try to find the line of code that *Delegate Call* happened on it and collect these lines. Now that we have storage variable names and storage slots of these variables and also the line of code inside fallback that have the delegatecall, we will check to find the target address variables. We are doing that by checking if one of the storage variables inside Storage function is used in the line of code that contains delegate call. We will add them to an array of implementation addresses.

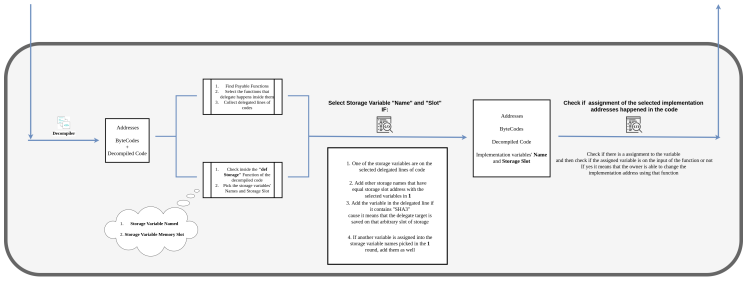
There is two other steps here. First finding other variable names with the same storage slots as the implementation addresses we found from the first step by checking the Storage function and also finding another variables that being assigned to those implementation variables in some other part of the code. We will add these two type of variables to the implementation addresses as well.

Now that we have a list for implementation addresses, we will search through the code to find if any assignment happened to one of them. If yes we will pick the variables that is assigned to target variable and then check if this assignment happened in a specific function and to one of the inputs of that function. In this case this function will be the upgrade function because the caller of this function can upgrade the target address by calling this function with desired input.

To summarize what we did, we find all possible variables in the code that can change the target address inside the contract and check if there is any function inside them that can assign new address to the target address variable.

The whole process is depicted on figure B.1.

[17] `https://github.com/palkeo/panoramix`



Assignment Checker Module

Fig. 4. Assignment CheckerModule

C Detailed Explanation of finding Admin types

The whole process is depicted in the figureC.

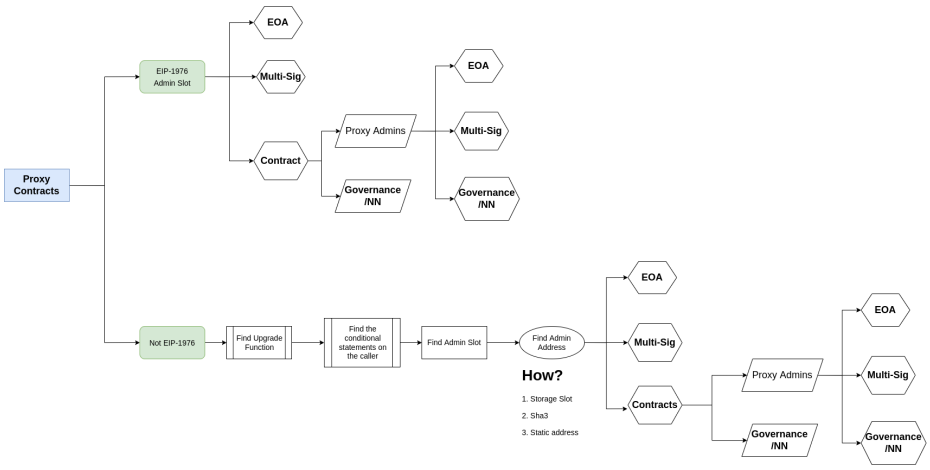


Fig. 5. Admin Types

EIP-1967. As mentioned above EIP-1967 [**?**] suggested specific arbitrary slots for upgradeable proxy contracts to store implementation contract's address and *Admin address*[18].

In first step we use *eth_getStorageAt* method of an Ethereum full archival node to search the EIP-1967 specified storage slot for admins on our 7.3k proxy contracts. If the result of this method is non-zero it means that the proxy uses EIP-1967 standard because the specified storage slot is an arbitrary slot and one can store variable in this slot just by defining this slot which means that they used EIP-1967.

So, for non zero results, we capture the address which is the address of admin of the proxy. Now we try to find the type of these admin addresses. Having the address of the admin we use *eth_getCode* method to check the code of the admin account. If the code is empty, it means that this account is not a smart contract so it is an EOA. we find *900* EOA admins that their proxy uses EIP-1967 standard.

The remained admin addresses are contract because their account keeps code. This contract can be multi signature smart contract wallets. The most widely used multi signature wallet is Gnosis Safe[19] wallets. We automatically checked if the code of the admin address is the Gnosis wallet multi signature patterns.

[18] Storage slot 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103 for admin

[19] `https://gnosis-safe.io/`

After picking Gnosis safe wallets we manually checked %10 of the remained addresses to find if they used other patterns for their multi signature wallet and we found some other patterns (e.g. MultiSignatureWalletWithDailyLimit, etc.). After Finding all these types we checked the admin codes to see whether they are multi signature wallets. We find *255* admin accounts that uses multi signature wallets as their admin.

There is another class of admin contracts named *Admin Proxy* contracts. These admin proxy contracts are another layer of re-direction between the real admin and the Dapp's proxy contract. The admin proxy contracts are proxy contracts that redirect the messages from the real admin into the Dapp's proxy. The only person who can use admin proxy is the admin (a.k.a owner) of the admin proxy. So we first filter the admin proxy contracts using the codes we get from the previous part and then try to find the owner of the admin proxy contracts. The owner of admin proxy contract (the real admin) also can be EOA, Multi-sig or governance contract. Finding the owner of the admin proxy contract, we used *eth_getCode* method to check the code of these account and find out if they are EOAs or Multi-signatures or governance schemes. Doing this we find *1202* EOA admin accounts and *567* multi signature admins. We marked the remained proxy admin addresses as Governance/Not Known admin types and we have *462* of them. There were also non admin proxy contracts which use EIP-1967 but they were not EOA or Multi signatures. We marked them as Governance/Not Known admin types and we have *53* of them.

Non EIP-1967. For proxy contracts which not use EIP-1967, the problem is we don't know where the admin address is saved in the proxy contract's storage (what is the storage slot of the admin address). It can be saved in a storage slot of the contract or be hardcoded in the smart contract[20].

So there are two ways that the admin address is saved in the proxy contract. It can be saved as a storage variables or it can be hardcoded as a fixed address.

In storage variable case, the first question is in which storage slot the admin address is stored. So, the first step is to find the storage slot of the admin address variable. Also for the fixed address we should find the fixed address of the admin directly.

To find the slot of the storage variable in which admin address is saved, we first find the function in which the proxy can be upgraded. For finding the upgrade function we exactly do what we did in B part. We first find the storage variable in which we saved the implementation address and then we find a function in which the implementation address can be changed using the inputs of that specific function.

The upgrade function of a proxy contract is a critical function and the only account that can call this function should be the admin of the proxy contract. So, there should be an access control check inside the upgrade function to check

[20] There are some other possible ways to store the admin address for instance saving it in another contract and each time make an external call to get the address but to our knowledge this pattern is not widely used as a standard

whether transaction sender is equal to the admin address or not. So, after finding the upgrade function we search for conditionals that checks the caller of the transaction and by doing that we can find the admin address or the storage variable in which the admin address is stored.

If the admin address is stored in a storage variable, then we should find the storage slot of that specific storage variable. For finding the storage slot we do what we did in B part by using *def storage* function of the decompiler and check the storage slot of the storage variable we found, and the admin address is saved on it. Now we have the storage slot of the admin address and we should start doing all the things we did for EIP-1967 in the previous part. In the EIP-1967 the storage slot for admin address was pre-specified and we do not need to find the slot but in this case we use the above methodology to find the slot but the further steps are the same as EIP-1967. So, by using the *eth_getCode* method for admin address inside the storage slot we find above, we can check wether the admin is EOA, Multi-sig, Governance, Proxy admin or not known. In this part we find *1313* EOA addresses and *104* multi-sig admins. Also by checking proxy admins we find *92* EOA addresses and *16* Multi signatures that uses proxy admin as a level of indirection.

In another case the admin address may be stored directly in a specific arbitrary storage slots. In this type the compiler will specify the address using the *sha3* hash notation. In this case same as above we find the conditional check on the transaction sender and then find the storage slot in that line and hash of that pre-specified string. By finding this arbitrary storage slot and doing the same processes we did in the previous part we find *2* EOA addresses and *10* Multi-sig addresses.

The only case that is left is proxy contracts, in which the address of the admin is hardcoded inside them. It very straight forward. We find the upgrade function and the access control check on the caller of the transaction and then pick the fixed admin address and do the same processes mentioned above to find the admin types. There are *49* EOAs, *36* multi-signature admins and *160* governance and not known admin addresses.

So, totally out of 7.3k proxy contract, **3558** are controlled by an EOA address, **988** are controlled by a multi signature wallet and **2924** addresses are governance controlled or our methodology could not find their type.

D Attacking Universal Upgradeable Proxy Standard (UUPS) contracts

In the previous sections A.8 (in Delegate-call section) we discussed that the implementation function should not contain *constructor* function and instead of that there should be regular function namely *Initialize* function that can be called just once and has the same functionality as constructor function. So the contract creator must call initialize function through proxy contract quickly after deploying the proxy contract. The Initialize function does not have any access control because it is considered to be called once and this function is responsible

to define the owner so before calling this function we don't have the owner's address to check it for access control of the function. This is the main reason that there should be a check to be sure that this function can just be called once and not more. Regularly, the deployer (the dev team) will define and initialize the owner of the contract (address which have the control of upgrading the system) via initialize function. So if the initialize function can be called more than once, or the deployer do not initialize the contract, then any external address can call initialize function and change the owner of the contract and take control of the contract. But this is not the case and less likely to happens because this is the first row of checklist of this type of upgradeability pattern.

There is another important issue that should take care of. We talked about calling the initialize function through proxy contract. But what about calling this function directly from the implementation contract itself? If the deployer do not call the initialize function directly from implementation contract to initialize it once (and to lock it), then any malicious address can call this function from implementation contract and change the owner inside the implementation contract and take control of the implementation contract. But one can ask what is the issue if another person takes control of the implementation contract. The answer depends on the functions inside the implementation contract and what executions can be done by the owner. If the functions inside implementation contract just change the state, then the attacker cannot attack the system because he/she just can change the state inside the implementation contract and our Dapp relies on the data that are saved inside proxy and not the data inside the implementation contract.

But it is not end of story, if the owner of the implementation contract has ability to self-destruct the contract, then the attacker can use initialize function inside the implementation contract to take control of the contract, and self-destruct it. This type of attack is a Denial of Service attack because transaction which are sent to proxy will be delegate called into a self-destructed contract and cannot peruse the desired logic. There is another way for attacker to self-destruct the implementation contract even if the contract itself does not have a self-destruct inside it. In case that the implementation contract has a function in which it delegate calls to another contract, and the target address of this delegate call can be changed by owner, then the attacker can take control of the owner's address using initialize function and then change the target address in the implementation contract and then call the function that delegate calls to the malicious address. The attacker can implement self-destruction inside the malicious contract and so, when the implementation contract delegate calls the malicious contract which executes self-destruct the implementation itself will be self-destructed like the previous scenario.

If the Dapp has an upgrade function inside its proxy contract, then the owner of proxy can just upgrade the proxy into a new version of implementation contract, This attack is explained in December 2020 by Trail of Bits team when they audit the code of Aave, a lending project [2].

There is another scenario which is more detrimental than the explained scenario. As mentioned in the previous sections in UUPS upgradeable contracts, the upgrade function resides in the implementation contract and so there is no way to upgrade the system by proxy itself. So if an attacker can take control of the implementation contract by calling initialize function directly from implementation contract, and then self-destruct it, there is no way to upgrade the system and consequently the proxy will be locked forever. All UUPS contracts that used Openzepplin UUPS library and their implementation contact do not get initialized is susceptible to this attack because there is a function in implementation contract of this library *upgradeToAndCall* in which the owner can change a target address and then delegate call into the newly changed target address. This attack vector was found in September 2021 and announced by OpenZepplin team [9][10]. There is an easy way to mitigate this attack just by calling initialize function directly from implementation contract.

We try to check all UUPS contracts that we have to find if any of them can be exploited in this way. We check all of them manually and the method of checking them is described bellow:

1. Find initialize function inside the implementation contract
2. check if anybody can call this initialize function directly from implementation contract and change the owner of the contract
   - Filter ones that are already initialized and so locked (which means that the initialize function is not callable)
   - Filter those ones that have a modifier that blocks direct calls from the implementation contract (there is modifier that lets just transactions that come from proxy contract and blocks direct calls from implementation)
3. Check if there is a way inside the implementation to self-destruct
4. Check if there is a function inside the implementation contract in which there is a delegate call to another target address
5. Check if the target address is changeable

Checking the list above we find 15 contracts in our data set which were exploitable till September 9, 2021 and openzeppelin team patched them by initializing the contract. It means that an attacker could deploy a new malicious contract which executes self-destruct on any calls to it. Then take control of the implementation contract by calling initialize function of them. Afterwards, the attacker should find the function inside the implementation contract that have a delegate call inside it and find the target address. There should be a function inside implementation contract to change the address to the malicious contract that the attacker deployed recently and just after that the attacker call the function to execute a delegate call into the malicious contract and then self-destruct the implementation contract.

We find 61 UUPS contracts that are not initialized and anybody can take control of these implementation contracts but because these contracts do not use delegate call or self-destruct, they are not exploitable by this type of attack.