Smart Libraries – Libraries and Proxy Contracts

Smart Contract Libraries: Reusable and Secure Code Modules

In the Solidity programming language, libraries are a special type of contract that contains reusable code. They are deployed only once to the blockchain at a specific address, and their code can be reused by multiple other contracts. This promotes code reuse, reduces deployment costs, and enhances security by centralizing the logic for critical operations.

The primary way libraries are used is through delegatecall, the same low-level EVM instruction that powers proxy contracts. When a contract calls a library function using delegatecall, the library's code is executed in the context of the calling contract. This means:

- The library code operates on the storage (state variables) of the calling contract.
- The msg.sender and msg.value are preserved.
- The library itself is stateless; it does not have its own storage. It merely provides functions that other contracts can use to manipulate their *own* storage.

The paper implicitly references libraries when discussing the **Eternal Storage** pattern. In this pattern, to avoid storage layout clashes, a contract doesn't declare state variables in the traditional way. Instead, it uses a set of functions (often housed in a library) that store and retrieve data from specific, pre-defined locations in the contract's storage using mappings. For example, instead of having a state variable uint256 public totalSupply;, a contract would use a library function like StorageLib.getUint(keccak256("totalSupply")). This library function would access a mapping mapping(bytes32 => uint256) internal uints; at a known storage slot. This decouples the storage structure from the logic, making it easier to manage and less prone to clashes during upgrades.

Proxy Contracts: The Backbone of Upgradeability

The paper's central focus is on **Proxy Contracts**, which are a specific architectural pattern to achieve upgradeability. A proxy contract is a shell or a gateway that users interact with directly. Its core components, as detailed in the document, are:

1. **Storage for Logic Address (**addr_logic**)**: A slot in the proxy's storage that holds the address of the contract containing the current business logic.
2. **Fallback Function**: A special function that executes when a function called on the proxy does not exist in the proxy itself. This function contains a delegatecall to the address stored in addr_logic.
3. **Setter Function**: A function (often restricted to an admin) that allows updating the addr_logic address to a new version.
4. **Getter Function**: A function to retrieve the current logic address.

The magic happens in the fallback function. When a user calls myProxy.doSomething(), the proxy contract doesn't have a doSomething function. This triggers the fallback function, which uses delegatecall to execute the doSomething() function from the logic contract, but within the proxy's own context. This means all state changes (storage updates) happen in the proxy's storage, not the logic contract's. The logic contract is effectively a library of executable code, and the proxy is the persistent storage layer.

The Crucial Intersection: Libraries vs. Logic Contracts in a Proxy System

While both libraries and logic contracts in a proxy pattern use delegatecall, they serve different hierarchical purposes:

- **Logic Contracts are "Versioned Implementations"**: In a proxy system, the logic contract is the complete, versioned set of business logic for the application (e.g., V1, V2, V3). It contains state variable definitions (the storage layout it expects) and the functions that operate on them. Upgrading involves pointing the proxy to a new logic contract address.
- **Libraries are "Shared Utilities"**: Libraries are used *within* a logic contract (or within the proxy itself) to perform common tasks. A logic contract might use a safe math library for arithmetic operations, a signature verification library for permissions, or a complex data structure library. The library is a reusable component that the logic contract relies on.

Therefore, a typical architecture looks like this:

1. **Proxy Contract**: Holds the state and delegates calls to the Logic Contract.
2. **Logic Contract (V1)**: Contains the main business logic and uses delegatecall to execute functions from various Libraries for specific tasks.
3. **Libraries**: Deployed, immutable contracts providing reusable functions.

This creates a layered system where the proxy handles upgradeability, the logic contract defines versioned business rules, and libraries provide secure, audited, and gas-efficient low-level operations.

Security Implications and Design Patterns

The paper's taxonomy and findings heavily revolve around the security implications of how proxies manage their interaction with logic contracts, which can be seen as a formalized use of delegatecall.

1. Storage Layout Clashes:
This is one of the most critical risks identified. It occurs when the storage layout of the proxy and the logic contract, or two versions of a logic contract, are incompatible. Since delegatecall executes logic in the caller's context, if the logic contract expects its variable x to be at storage slot 0, but the proxy has already stored its addr_logic at slot 0, the logic contract will overwrite the proxy's critical data. The paper documents real-world instances of this, such as in Synthetix and a DexProxy, where a storage clash made a proxy impossible to upgrade.

The study identifies several patterns to mitigate this:

- **Inherited Storage**: The proxy and logic contract inherit from the same base storage contract, ensuring they agree on the initial storage layout.
- **Unstructured Storage**: The proxy stores its addr_logic at a pseudo-random, high-numbered storage slot (e.g., keccak256("eip1967.proxy.implementation") - 1), far away from the slots a logic contract would typically use. This is the approach of standards like EIP-1967.
- **Eternal Storage**: As mentioned earlier, this uses a library-like approach with generic mappings for all data types, completely decoupling the storage schema from the logic.

2. Function Selector Clashes:
This risk arises when a function in the proxy (like the setter) has the same 4-byte function selector as a function in the logic contract. A user calling the logic function could inadvertently trigger the proxy's

function, with potentially disastrous consequences (e.g., changing the addr_logic to a malicious address). The paper describes patterns designed to avoid this:

- **Transparent Proxy Pattern**: This pattern introduces an admin address. Calls from the admin are routed to the proxy's own functions (like the setter), while calls from any other address are routed through the fallback to the logic contract. This perfectly isolates the namespaces.
- **UUPS (EIP-1822)**: This pattern moves the setter function *into the logic contract itself*. Since the proxy has almost no functions of its own, the risk of collision is minimal.

3. Upgradeability Policy and Safety:
The research uncovers "policy issues" related to how upgradeability is managed. A key finding is that when the setter is in the logic contract (as in UUPS), upgradeability can be permanently removed—either accidentally or intentionally. The paper notes that while some see this as a dangerous safety issue, others (like the EIP-2535 Diamond standard proponents) view the ability to "freeze" a contract and make it immutable as a feature.

Furthermore, the study found instances of **time-delayed upgrades**, where an upgrade proposal must wait for a period (e.g., days or weeks) before being executed. While this enhances trust and decentralization, it creates a security risk where critical vulnerabilities cannot be patched immediately, as tragically demonstrated by the $100 million Compound incident described in the paper.

Conclusion

In summary, while **Libraries** are a foundational tool for code reuse and gas optimization within a single contract or logic implementation, **Proxy Contracts** are a high-level architectural pattern that uses the same underlying delegatecall mechanism to enable a system's evolution over time. The research paper "Proxy Hunting" systematically reveals that the intersection of these concepts is where both immense flexibility and significant risk reside. By developing a taxonomy and the UseHunt tool, the authors provide a framework for understanding the diverse design patterns (like Transparent Proxy, UUPS, and Diamond) that have emerged to manage this complexity. Their large-scale study proves that while USCs have become indispensable, holding billions of dollars in value, they introduce a new class of security concerns—from low-level storage clashes to high-level upgrade policies—that the blockchain community must rigorously address.
