BLOCKCHAIN (HTTPS://AUTIFYNETWORK.COM/CATEGORY/BLOCKCHAIN/)

Exploring ERC-2535: The Diamond Standard for Smart Contracts 💎

Soumitra Sen (https://autifynetwork.com/author/soumitra/)

April 11, 2023 (https://autifynetwork.com/exploring-erc-2535-the-diamond-standard-for-smart-contracts/)

Are you looking for a smarter, more efficient way to modularize and upgrade your Ethereum smart contracts? Look no further than the ERC-2535 Diamond Standard – the ultimate solution for developers is here! The core idea of the Diamond standard is similar to upgradeable smart contracts, such as the proxy pattern, but with the benefit that you can control many implementation contracts (i.e., logic contracts) from your single Diamond contract (i.e., proxy contract).

Features & Types of ERC-2535

The Diamond Standard boasts several crucial features, including:

- A single gateway for carrying out proxy calls to n number of implementation contracts
- The capability to upgrade single or multiple smart contracts atomically
- No limit on the number of implementation contracts that can be added to the Diamond
- A comprehensive record of all upgrades executed on the Diamond
- Reduced gas costs by decreasing the number of external function calls

There are different types of Diamonds, such as:

- **Upgradeable Diamond**: A mutable contract that can be upgraded
- **Finished Diamond**: An immutable contract due to the upgradeability feature being removed
- **Single Cut Diamond:** An immutable contract that can no longer be upgraded

Some protocols that have adopted the Diamond standard are Aavegotchi (https://www.aavegotchi.com/), BarnBridge (https://barnbridge.com/), DerivaDEX (https://derivadex.com/), and Oncyber (https://oncyber.io/).

Applications of ERC-2535, the Diamond Standard

1. **Upgradeable Contracts**

The Diamond Standard allows for the creation of upgradeable contracts, which means that developers can update their smart contracts without the need for manual intervention. This feature is especially useful for smart contracts that require ongoing maintenance or updates.

2. **Modularization**

The Diamond Standard simplifies the modularization of smart contracts, allowing developers to separate the contract's logic and data into separate implementation contracts. This makes it easier to maintain, upgrade, and deploy smart contracts in a more organized and scalable way.

3. **Gas Efficiency**

The Diamond Standard can reduce gas costs by decreasing the number of external function calls, which can lead to significant savings in gas fees. This makes smart contract development more cost-effective and efficient.

4. **Atomic Upgrades**

The Diamond Standard allows for atomic upgrades, which means that multiple smart contracts can be upgraded simultaneously, making it easier to keep multiple contracts in sync with one another.

5. **Interoperability**

The Diamond Standard is compatible with other Ethereum standards and can be used in conjunction with other smart contract frameworks, allowing for greater interoperability and flexibility in smart contract development.

History of Diamond Standard for Smart Contracts(ERC-2535)

Nick Mudge (@mudgen (https://twitter.com/mudgen)) began working on the ERC-2535 Standard for Smart Contracts to circumvent the 24KB maximum contract size limit. Interested to read more about ERC -2535? You can read the complete backstory of the diamond standard of the smart contract here (https://eip2535diamonds.substack.com/p/introduction-to-the-diamond-standard).

In short, ERC-2535 Diamonds have **one storage space** for all state variables and **one Ethereum address** from which all functions can be designed and implemented without bytecode size limitation. State variables may be read and written directly, simply, and uniformly by all functions. To top it all off, a seamless upgrade mechanism also exists that can be used to replace existing functions, remove them, and add new ones without needing to redeploy everything.

Sounds unreal, right? Well, ERC-2535 Diamonds (https://eips.ethereum.org/EIPS/eip-2535) exist and are here to change the way smart contracts were being written.

What's with the Diamond Terminology?

The Diamond terminology used in the ERC-2535 Diamond Standard refers to the multiple facets of a diamond, which represent the multiple implementation contracts that a single Diamond contract can control.

A real physical diamond has different sides and different facets. An Ethereum diamond also has different sides and facets as sets of related functions (smart contracts). Just as the facets of a physical diamond are linked to a central core, the Ethereum address and the state of a diamond serve as a common core. With experience in implementing diamonds, one's mind will categorize them into different function sets that are associated with this core.

Additionally, just like a real diamond, the ERC-2535 Diamond Standard is designed to be durable, valuable, and versatile, allowing developers to seamlessly upgrade and modify their smart contracts with ease.

In the diamond ecosystem, upgrades are referred to as "cuts," which may raise confusion among those who are not familiar with physical diamonds. The process of physically cutting diamonds to create or add facets is actually how retail diamonds are made.

A special magnifying glass known as a “loupe” is used to closely examine and inspect diamonds.

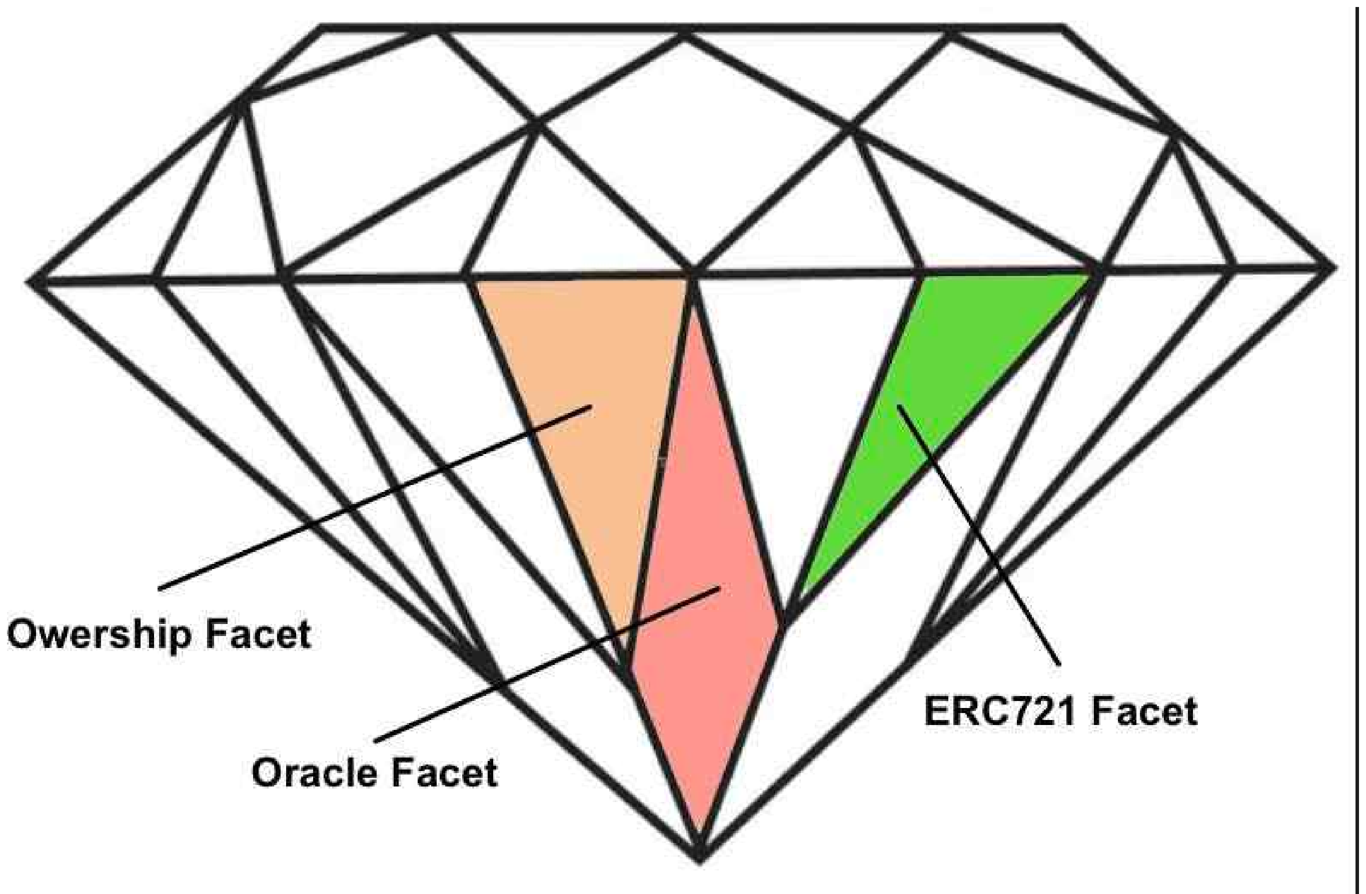
With this high-level knowledge of the standard, let us now explore some of the components that constitute the standard in more detail.

Anatomy of the Diamond Standard

1. **The Diamond**

A Diamond is a smart contract that uses other smart contracts (aka Facets) to make *delegatecall* calls. So, what exactly is a Facet and delegatecall?

In essence, a delegate call is a special form of an external function call in which a proxy contract uses code from another smart contract in its own context (i.e., its own state variables). Facets, which allow you to divide your implementation logic into multiple Solidity files and update them as required, make up the second critical aspect of a Diamond contract.



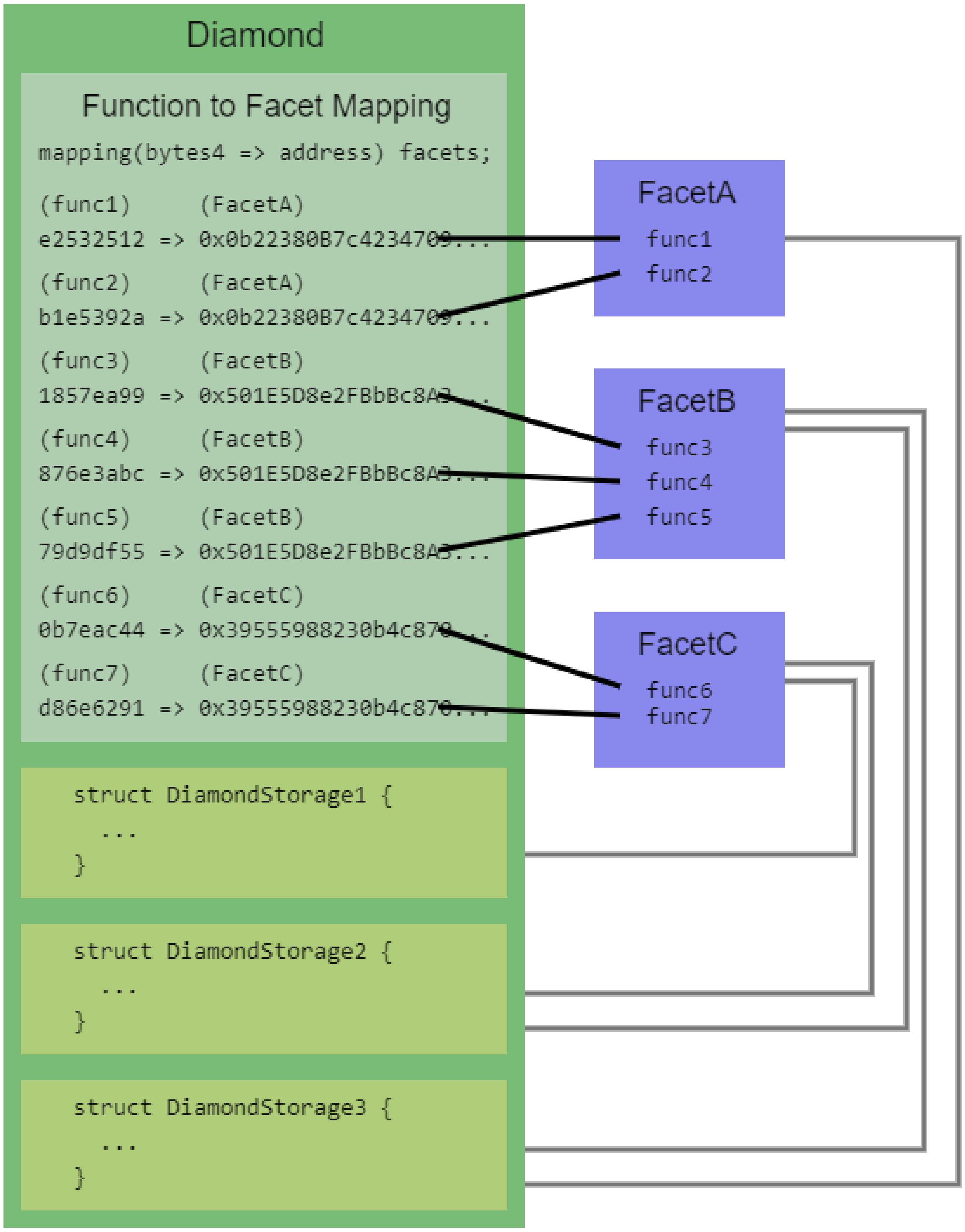
1. A diamond is a smart contract. The only address external software uses to communicate with it is its Ethereum address.
2. Facets, a collection of contracts, are utilized by the Diamond to handle its external functions.
3. The storage of state variable data is housed in the Diamond rather than in its facets.

4. The external functions of facets can directly read and write data stored in a diamond. This makes facets easy to write and gas efficient.
5. The Diamond operates via a fallback function that leverages delegatecall (https://eip2535diamonds.substack.com/p/understanding-delegatecall-and-how) to route external function calls to its facets.
6. To manage its data, a Diamond frequently leverages facets for its external functions rather than relying on any external operations of its own.

2. **Facets (The logic keepers)**

A Facet holds the external function logic that a proxy contract, such as a Diamond, calls upon and can be compared to an implementation contract or library. Unlike the conventional proxy pattern, there are no restrictions on the number of Facets that can be added to a Diamond. The use of Facets in a Diamond contract allows for code modularization and targeted updates, unlike the common proxy pattern that requires the redeployment of a new implementation contract. With Facets, multiple Facets or an individual Facet can be updated since there can be many implementation contracts.

Here's an image to help visualize a diamond. The image below shows how functions are mapped to the facets that hold the function code:


Diamond
Function to Facet Mapping
mapping(bytes4 => address) facets;
(func1) (FacetA)
e2532512 => 0x0b22380B7c423470...
(func2) (FacetA)
b1e5392a => 0x0b22380B7c423470...
(func3) (FacetB)
1857ea99 => 0x501E5D8e2FBbBc8A3...
(func4) (FacetB)
876e3abc => 0x501E5D8e2FBbBc8A3...
(func5) (FacetB)
79d9df55 => 0x501E5D8e2FBbBc8A3...
(func6) (FacetC)
0b7eac44 => 0x39555988230b4c870...
(func7) (FacetC)
d86e6291 => 0x39555988230b4c870...
struct DiamondStorage1 { ... }
struct DiamondStorage2 { ... }
struct DiamondStorage3 { ... }
FacetA
func1
func2
FacetB
func3
func4
func5
FacetC
func6
func7

3. **DiamondCut (The managing interface)**

A Diamond contract must include the standard function known as DiamondCut. It enables you to add, remove, or replace functions within your contract. Through the "upgrade" feature of DiamondCut, you can alter the mapping (i.e., selectorToFacet) stored in your

Diamond contract to reference a different function signature and/or address. Whenever the DiamondCut function is executed, an event is emitted, which can be used to track the upgrade history of your contract.

4. **AppStorage and DiamondStorage (The data keeper)**

There are multiple ways of incorporating storage in a Diamond contract, but the two most prevalent approaches are through AppStorage and DiamondStorage. When using DiamondStorage, you define your state variables within structs, and each of these structs is allocated a specific location in the contract's storage.

Another way of managing your state variables and storage hierarchy is through AppStorage, which is more efficient since it allows you to share state variables across different facets. With AppStorage, you create a smart contract that contains a struct representing all of your state variables. Once deployed, you can import the AppStorage into multiple facets.

5. **DiamondLoupe (Finding out which functions are supported)**

EIP-2535 Diamonds describe four fundamental functions, also known as loupe functions, that showcase the facets and capabilities of a diamond. More details about these functions can be found in this article: Diamond Loupe Functions (https://dev.to/mudgen/why-loupe-functions-for-diamonds-1kc3). For displaying information and executing the functions of diamonds, you can refer to louper.dev (https://louper.dev/)

Organizing Facets of a Diamond

Organizing facets in a Diamond contract can be compared to organizing a file system. Similar to how a file system is commonly organized, facets in a Diamond contract can be arranged as follows:

1. Functions that are related and similar can be placed in the same facet.
2. Similar and related facets can be grouped in the same folder.
3. Similar and related folders can be grouped in the same parent folder.

For instance, while implementing an ERC-721 token, the external functions from the ERC-721 standard can be grouped in the same 'ERC721Facet,' and custom functionality can be placed in another facet.

Even if a contract's 24KB maximum size limit is removed, ERC-2535 Diamonds still provide a systematic approach to organizing, extending, and upgrading a smart contract system. The ERC-2535 Diamonds were initially designed to overcome the 24KB contract limit, but it turns out they are beneficial beyond that. They provide a framework for developing larger smart contract systems and contract systems that can grow in production.

How can you create your own ERC-2535 Diamond?

Currently, there is no available reference implementation of EIP-2535 via OpenZeppelin contracts, but it may be available soon, as mentioned in this **GitHub issue** (https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2793).

However, the EIP author Nick Mudge (http://www.perfectabstractions.com/) has created three reference implementations (https://github.com/mudgen/diamond) that are all audited (https://eip2535diamonds.substack.com/p/smart-contract-security-audits-for) and have different advantages and disadvantages in terms of code complexity and gas costs. These implementations are available on GitHub:

- diamond-1-hardhat (https://github.com/mudgen/diamond-1-hardhat) (Simple implementation)
- diamond-2-hardhat (https://github.com/mudgen/diamond-2-hardhat) (Gas-optimized)
- diamond-3-hardhat (https://github.com/mudgen/diamond-3-hardhat) (Simple loupe functions)

All three implementations serve the same purpose, so the choice depends on your specific needs. If you plan on doing multiple upgrades and are concerned about gas costs, diamond-2 might be the best option due to its use of complicated bitwise operations to reduce storage space and save roughly 80,000 gas for every 20 functions added. Otherwise, diamond-1 might be the better choice since its code is more readable.

How to store state variables in Diamonds?

Solidity stores data in contracts using a numeric address space, where the first state variable is stored at position 0, the second at position 1, and so on. However, facets of a diamond share the same storage address space because they operate on the same diamond and only read and write state variables in the diamond, not in themselves. Understanding how **delegatecall** (https://eip2535diamonds.substack.com/p/understanding-delegatecall-and-how) works are crucial for this concept because all external functions of facets are called with delegatecall from the diamond.

But sharing the same storage address space can lead to problems if not handled properly. For instance, if a diamond has two facets, FacetA and FacetB, and let's say FacetA declares state variables **uint first** and **bytes32 second**, and FacetB declares state variables **uint first** and **string name**. Both facets are storing **uint first** at position 0, so both facets can read and write that variable without any issue. However, both facets declare state variables with different names but share the same storage location, so they clobber or overwrite each other's data. Here, FacetA declares a state variable named **bytes32 second,** and FacetB declares a state variable named **string name** , both are written and interpreted differently at position 1, resulting in data corruption.

To prevent such issues, facets of the same diamond need to declare the same state variables in the same order if they are reading and writing to the same locations. There are three common strategies to organize data in facets:

1. Inherited Storage
2. Diamond Storage
3. AppStorage

A detailed explanation of these strategies can be found here (https://eip2535diamonds.substack.com/p/introduction-to-the-diamond-standard#%C2%A7inherited-storage).

Maintaining Integrity

Maintaining the integrity of state variables is crucial during an upgrade process.

To ensure this, the following steps can be taken:

1. When adding new state variables to an **AppStorage** struct or a **Diamond Storage** struct, always add them at the end of the struct. This is because existing facets cannot overwrite state variables at new storage locations.
2. For structs used in mappings, new state variables can be added at the end of the struct.
3. Renaming state variables is possible, but it can be confusing if different facets are using different names for the same storage locations.

Action steps to prevent corrupting state variables during upgrades

1. When using AppStorage, avoid declaring and using state variables outside of the AppStorage struct, except when using Diamond Storage.
2. When using Diamond Storage, ensure that different structs are assigned unique namespace strings to prevent overwriting of different structs at the same location.
3. Do not add new state variables to the beginning or middle of structs. This will overwrite existing state variable data, causing all state variables after the new state variable to reference the wrong storage location.
4. Avoid adding new state variables to structs that are used in arrays.
5. Avoid placing structs directly inside other structs, unless you have no intention of adding more state variables to the inner structs in the future. Adding new state variables to inner structs during upgrades will overwrite the next state variable after the inner struct. However, structs within mappings can be extended during upgrades because they are stored in random locations based on keccak256 hashing.

6. Do not allow any facet to call **selfdestruct**. To prevent this, avoid using the **selfdestruct** command in any facet source code or calling it through delegatecall, as it can delete a facet used by a diamond or even the diamond proxy contract itself.

Understanding how Solidity assigns storage locations to state variables is crucial to making sense of these rules. An example of adding new state variables in a diamond upgrade can be found here (https://eip2535diamonds.substack.com/p/introduction-to-the-diamond-standard#%C2%A7inherited-storage).

Diamond Deployment

To deploy a diamond, at least one facet needs to be added in the constructor of the diamond to add the 'diamondCut' or other upgrade function. After deployment, more facets can be added using the upgrade function.

It is possible to create a 'Single Cut Diamond' by adding all the facets that the diamond will ever have in the constructor of the diamond and leaving out any kind of upgrade function. However, this type of diamond is not upgradeable.

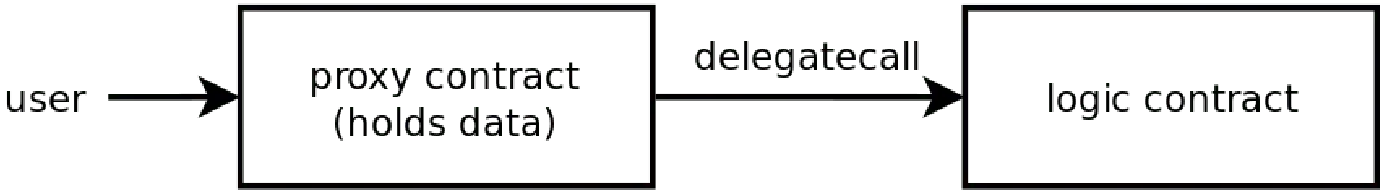
Is ERC -2535 Secure?

The ERC-2535 Diamond Standard has gained widespread adoption in the Ethereum ecosystem, with many protocols leveraging its powerful features to build more flexible and upgradable smart contracts. However, with great power comes great responsibility, and it's crucial to ensure that these implementations are thoroughly audited for security vulnerabilities.

Several security audit firms, including Omniscia (https://omniscia.io/alliance-block-multitoken-bridge/), Certik (https://www.certik.org/projects/aavegotchi), Quantstamp (https://raw.githubusercontent.com/BarnBridge/BarnBridge-PM/master/audits/BarnBridge%20DAO%20audit%20by%20Quanstamp.pdf), MixBytes (https://github.com/pie-dao/audits/blob/main/Mixbytes%20-%20ExperiPie_Smart_Contrac%202020-12-11.pdf), and Haechi Audit (https://raw.githubusercontent.com/BarnBridge/BarnBridge-PM/master/audits/BarnBridge%20DAO%20audit%20by%20Quanstamp.pdf), have conducted smart contract audits of various diamond implementations to identify any potential security risks. These audits cover different implementations of the Diamond Standard, including diamond-1, diamond-2, and diamond-3, and were conducted for protocols such as Aavegotchi, BarnBridge, DerivaDEX, DOTC, and PieDAO.

By conducting smart contract security audits of Diamond Standard implementations, developers can ensure that their smart contracts are secure and robust. This is critical for building trust with users and avoiding potential security breaches.

Why should you use ERC-2535 Diamonds instead of the proxy upgrade pattern?



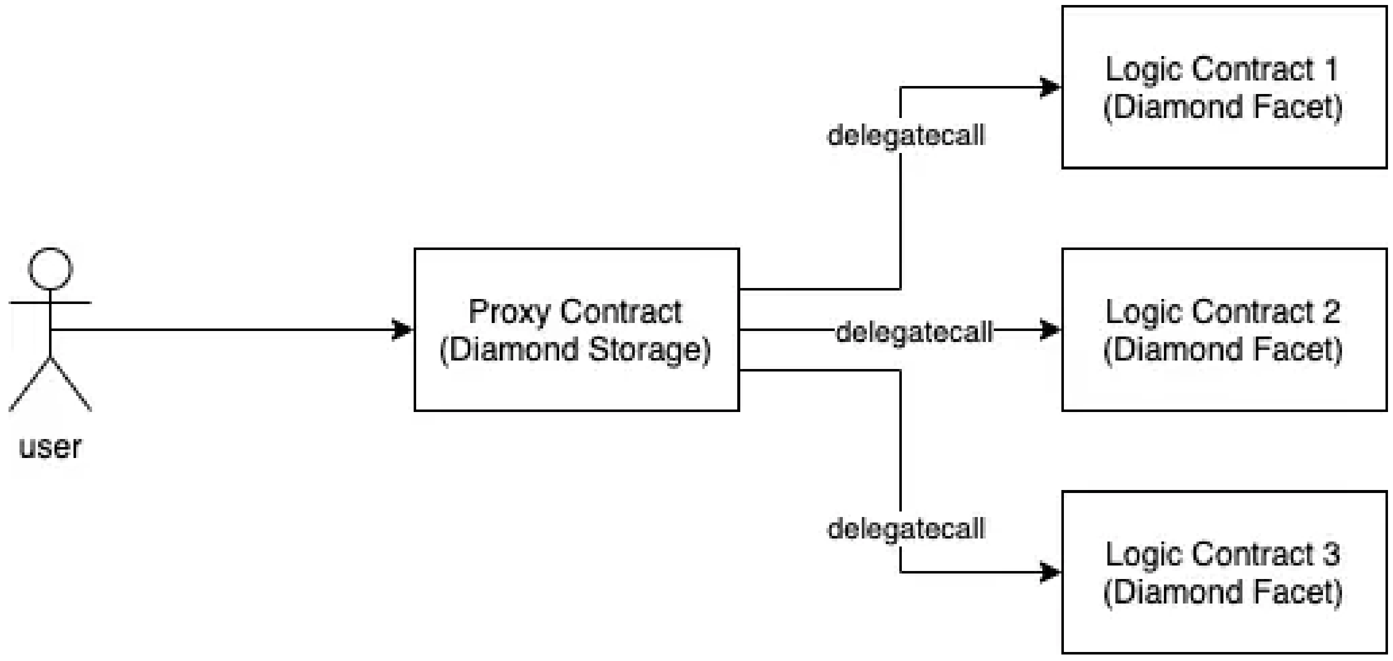
The user interacts with the proxy contract, which holds the data and is immutable. The contract contains a **fallback function** (https://docs.soliditylang.org/en/v0.8.9/contracts.html#fallback-function) that catches any function call and uses **delegatecall** to forward it to a separate logic contract. This means that the logic can be defined separately from the data.

This design enables us to update the logic contract without changing the underlying data. To update the logic, we only need to change the address of the logic contract defined in our proxy. This can be done through governance or, at a minimum, controlled by a multi-sig wallet.

The proxy pattern provides upgradability for smart contracts, but it has its limitations:

- Upgrading a small part of a complex contract is not possible. You have to upgrade to an entirely new logic contract, which makes it challenging to see the exact changes.
- Reusing logic contracts for multiple proxies is possible but not practical. Only identical proxy instances can be created using one logic contract. Combining logic contracts or using parts of one is not feasible.
- Modular permission systems cannot be implemented. It's an all-or-nothing approach, where anyone with permission can upgrade all existing functions.
- Data access within the logic contract requires special care after upgrades because the data in the proxy contract remains unchanged. Even if you stop using a state variable in the logic contract, you still need to keep it in the variable definitions. When adding state variables, you can only append them to the end of the definitions.
- Logic contracts can easily exceed the 24kb max contract size limit.

This is where diamonds come in. Diamonds are similar to the proxy pattern but with a key difference: they allow for multiple logic contracts to be used together.


user
Proxy Contract (Diamond Storage)
delegatecall
delegatecall
delegatecall
Logic Contract 1 (Diamond Facet)
Logic Contract 2 (Diamond Facet)
Logic Contract 3 (Diamond Facet)

Can’t solidity libraries achieve the same thing?

EIP-2535 Diamonds is a general smart contract architecture that addresses the need for designing extensible smart contract systems. While Solidity libraries are a useful tool for creating reusable on-chain code, they are not a contract architecture in and of themselves.

Diamonds natively support and solve some of the same problems that Solidity libraries do. Internal functions of Solidity libraries can be used to share functions between facets, while external functions from Solidity libraries can also be used with diamonds. Although Solidity libraries can be used to alleviate the 24kb contract size limit, their ability to do so is limited. Adding too many external functions to a contract can still result in hitting the contract size limit. In contrast, a diamond can have practically an unlimited number of external functions, enabling a deployed contract system to be extended without the risk of hitting a technical limit in the future.

While Solidity libraries can be used to reduce bytecode size, reorganizing code solely to avoid hitting a technical limitation may not be ideal. Instead, contract architectures can be designed to use Solidity libraries in a way that is both efficient and unlikely to hit the bytecode size limit. ERC-2535 Diamonds is a solution that uses delegatecall, as Solidity libraries do, and provides a way to easily extend smart contract systems without limit.

Why are we using the Diamond Standard at Autify Network?

There are several compelling reasons why we are using the Diamond Standard at Autify Network (https://autifynetwork.com/):

- Facet reusability
- Modular upgradability
- Reduced gas costs
- Modular permission system
- Storage slot management
- Avoiding max contract size issues
- A single gateway to make calls to the blockchain

Like every other technology, using Diamonds also has some drawbacks that need to be considered:

- More Complexity
- Harder to maintain
- Not many big real-life projects yet
- Not supported by tools like Etherscan, but an alternative for at least Etherscan exists: Louper (https://louper.dev/)

In conclusion, it should be noted that diamonds are a relatively new and emerging standard that has yet to gain widespread adoption.

However, at Autify Network, we are utilizing this cutting-edge technology to tackle real-world issues. If you are interested in staying up-to-date with our innovative approach to solving problems in the supply chain industry, we invite you to sign up for our waitlist here (https://autifynetwork.com/#typeform).

Looking for more information on Diamonds? 💎

Recommended Readings

- EIP-2535 Diamonds (https://eips.ethereum.org/EIPS/eip-2535)
- EIP-2535 Diamonds Substack (https://eip2535diamonds.substack.com)
- How Diamond Storage Works (https://dev.to/mudgen/how-diamond-storage-works-90e)
- Understanding Diamonds on Ethereum (https://dev.to/mudgen/understanding-diamonds-on-ethereum-1fb)

SHARE ON

(https://www.facebook.com/sharer/sharer.php?u=https://autifynetwork.com/exploring-erc-2535-the-diamond-standard-for-smart-contracts/)
(https://twitter.com/intent/tweet?text=Exploring%20ERC-2535:%20The%20Diamond%20Standard%20for%20Smart%20Contracts%20💎&url=https://autifynetwork.com/exploring-erc-2535-the-diamond-standard-for-smart-contracts/)
(https://pinterest.com/pin/create/button/?url=&media=https://autifynetwork.com/wp-content/uploads/2023/04/Diamond-Contract2@3x.png&description=Exploring+ERC-2535%3A+The+Diamond+Standard+for+Smart+Contracts+%F0%9F%92%8E)
(https://www.linkedin.com/shareArticle?mini=true&url=https://autifynetwork.com/exploring-erc-2535-the-diamond-standard-for-smart-contracts/&title=Exploring%20ERC-2535:%20The%20Diamond%20Standard%20for%20Smart%20Contracts%20💎&source=Autify%20Network)

NEXT ARTICLE →

← PREVIOUS ARTICLE

Unpacking ERC-4337: A Deep Dive Into Account Abstraction (https://autifynetwork.com/unpacking-erc-4337-a-deep-dive-into-account-abstraction/)

The Definitive Guide to Traceability (2023) | The Future of E-Commerce (https://autifynetwork.com/the-definitive-guide-to-traceability-2023-the-future-of-e-commerce/)

You may also like

(https://autifynetwork.com/what-is-supply-chain-management-and-why-is-it-important/)

1 month ago Blockchain (https://autifynetwork.com/category/blockchain/)

What is Supply Chain Management, and Why is it Important? (https://autifynetwork.com/what-is-supply-chain-management-and-why-is-it-important/)

Navigating the Supply Chain Maze: Autify Network takes you on a journey from raw materials to satisfied customers. Discover the role of blockchain in reshaping Supply Chain Management.

(https://autifynetwork.com/the-ultimate-guide-to-ui-ux-in-web3-unlock-the-secrets-to-exceptional-user-experiences/)

2 months ago Blockchain (https://autifynetwork.com/category/blockchain/)

The Ultimate Guide to UI/UX in Web3: Unlock the Secrets to Exceptional User Experiences (https://autifynetwork.com/the-ultimate-guide-to-ui-ux-in-web3-unlock-the-secrets-to-exceptional-user-experiences/)

AUTIFY NETWORK

NEED HELP?

EMAIL US AT
help@autify.network

COMPANY

Frequently Asked Questions (https://autifynetwork.com/faq/)

Terms & Conditions (https://autifynetwork.com/terms-and-conditions/)

The Blog(https://autifynetwork.com/the-blog/)

Privacy Policy (https://autifynetwork.com/privacy-policy/)