合约升级标准 ERC2535 的设计解析和不足

比特奇点
智能合约安全

合约升级标准 ERC2535 的设计解析和不足

Safful最近审计了钻石标准的一份实现代码，这一标准是一种新的可升级合约模式。撰写标准是一项值得赞许的事业，但钻石标准及其实现有许多引人担忧的地方。这份代码是过度工程的产物，附带了许多不必要的复杂性，所以现在我们不能推荐使用。 当然，钻石标准提议还处在草稿阶段，有成长和改进的空间。一套有用的可升级合约保证应该包含：

- **一个清晰的、简单的实现**。标准应该易于阅读，以化简与第三方应用的集成流程。
- **一个升级流程的通盘检查清单**。升级是有风险的，因此必须有透彻的解释。
- **对大多数常见升级错误（包括函数遮挡和函数碰撞）的链上缓解措施**。许多错误虽然易于检测出来，但都能导致服务出错。见“slither-check-upgradeability”一文了解许多可以化解的陷阱。
- **相关风险的清单**。合约可升级性不是简单的事，它可能遮掩了安全上应当考虑的问题，或者传递低估风险的暗示。EIP 是提升以太坊的提议，不是商业广告。
- **集成了常见测试平台的测试**。测试应该凸显出如何部署系统、如何升级一个新的实现，以及升级在哪些条件下会失败。

不幸的是，钻石提案没有满足所有这些要求。这实在太糟糕了，因为我们想看到的是一个可以解决、至少是减轻可升级合约的主要安全陷阱的标准。从根本上来说，标准的撰写人必须明确假设开发者会犯错，并且以开发出能缓解错误的标准为目标。 不过，我们 还是能从钻石提案中学到很多。请继续往下读：

- 钻石标准如何工作
- 我们的复查揭示了什么
- 我们的建议
- 可升级标准的最佳实践

钻石标准范式

钻石标准是由 EIP 2535 定义的、还在开发中的工作。提案的草稿声称要给予 delegatecall 方法提出合约升级的一种新范式。（我们曾撰写过一份关于合约如何升级的概述，仅供参考。）EIP 2535 提议使用：

1. 与实现合约适配的查找表（lookup table）
2. 任意的存储指针（arbitrary storage pointer）

查找表

基于 delegatecall 的升级方法主要使用两个组件：一个代理合约和一个实现合约

图 1. 单一实现合约的基于 delegatecall 的升级方法 用户与代理合约交互，代理合约向实现合约发送 delegatecall 调用实现合约内的函数。执行的是实现合约内的代码，但整套合约的 storage 保存在代理合约内。 使用了查找表，代理合约就可以向多个实现合约发起 delegatecall 调用，可根据要调用的函数来选择合适的实现合约：

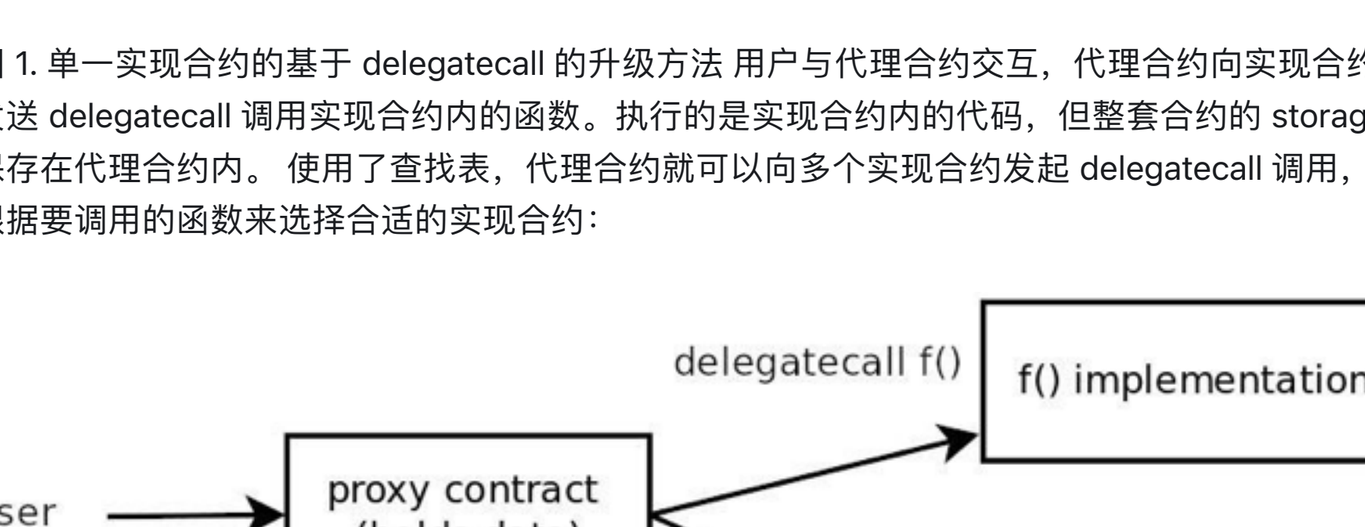

图 2. 多实现合约的基于 delegatecall 的升级方法 这种模式不是什么新东西。之前也有其他项目使用过这样的查找表来实现可升级性。ColonyNetwork 就是一个例子。

任意的存储指针

钻石提案还建议使用 Solidity 最近引入的一个功能：任意的存储指针（arbitrary storage pointer）。这个功能名副其实，就是允许你把一个 storage 的指针指向任意一个位置。 因为 storage 都存储在代理合约里，实现合约的 storage 布局必须与代理合约的 storage 布局保持一致。在升级的时候，很难跟踪这种布局（此处有一个例子）。 这个 EIP 提议，每个实现合约都要有一个相关联的结构体（structure）来保管实现合约的变量（variables），然后用一个指针指向存储该结构体的 storage 位置。这类似于“unstructured storage”模式，也是 Solidity 的一个新功能，支持使用一个结构体来替代一个单一的变量。 此处的假定是：来自两个不同实现的结构体不可能冲突，只要它们的基础指针（base pointer）不同。

```
bytes32 constant POSITION = keccak256(
     "some_string"
 );

 struct MyStruct {
     uint var1;
     uint var2;
 }

 function get_struct() internal pure returns(MyStruct storage ds) {
     bytes32 position = POSITION;
     assembly { ds_slot := position }
 }
```

图 3. storage 指针的例子

图 4. storage 指针的表示

插句话，什么是“钻石（diamond）”?

EIP 2535 提出了一套“钻石术语”，其中，“钻石”指的是代理合约，“雕琢面（facet）”指的是实现合约。等等。不太明白为什么要发明这套黑话，因为可升级性的标准术语都已经得到定义，而且众所周知了。我们这里做了一个列表来帮你翻译这套提案：

钻石标准术语	常见用名
Diamond	Proxy（代理合约）
Facet	Implementation（实现合约）
Cut	Upgrade（升级）
Loupe	List of delegated functions（delegated 函数的列表）
Finished diamond	Non-upgradeable（不可升级的合约）
Single cut diamond	Remove upgradeability functions（移除了可升级函数的合约）

图 5. 钻石提案使用新的术语来指称已有的观念。

审计结果及建议

我们复查了钻石标准的实现，成果如下：

- 代码是过度工程的产物，包含了许多颠三倒四（misplaced）的优化
- 使用存储指针带有风险
- 代码有函数隐藏（function shadowing）
- 合约缺少存在性检查（existence check）
- 钻石术语带来了不必要的复杂性

过度工程的代码

虽然 EIP2535 所提议的模式是直接了当的，但其实际实现却难以阅读，也难以审核，因此提高了出问题的概率。 举个例子，在链上保存许多数据是很麻烦的。虽然这个提案只需要用到查找表，从函数的签名映射到实现合约的地址，但 EIP 定义了许多接口，需要把额外的数据存为 storage：

```
interface IDiamondLoupe {
    /// These functions are expected to be called frequently
    /// by tools.

    struct Facet {
        address facetAddress;
        bytes4[] functionSelectors;
    }

    /// @notice Gets all facet addresses and their four byte function selectors
    /// @return facets_ Facet
    function facets() external view returns (Facet[] memory facets_);

    /// @notice Gets all the function selectors supported by a specific facet.
    /// @param _facet The facet address.
    /// @return facetFunctionSelectors_
    function facetFunctionSelectors(address _facet) external view returns (byte

    /// @notice Get all the facet addresses used by a diamond.
    /// @return facetAddresses_
    function facetAddresses() external view returns (address[] memory facetAddr

    /// @notice Gets the facet that supports the given selector.
    /// @dev If facet is not found return address(0).
    /// @param _functionSelector The function selector.
    /// @return facetAddress_ The facet address.
    function facetAddress(bytes4 _functionSelector) external view returns (addr
```

图 6. 钻石合约的接口 在这里，*facetFunctionSelectors* 会返回一个实现合约的所有函数选择器。这个信息其实只对链下的部件有用，而链下的部件完全可以从合约的事件中抽取出这些信息。其实根本没必要在链上放置这个功能，尤其，它还会极大地增加代码的复杂性。 此外，大部分的代码复杂性，都是因为去优化了无需优化的位置。举个例子，用于更新实现合约的函数应该是直接了当的。获得一个新的地址和一个新的签名后，代理合约应该在查找表中更新对应的入口。但是，你看看，用来实现这个功能的部分函数，长下面这个样子：

```
// adding or replacing functions
 if (newFacet != 0) {
     // add and replace selectors
     for (uint selectorIndex; selectorIndex &amp;amp;amp;amp;amp;lt; numSelecto
         bytes4 selector;
         assembly {
             selector := mload(add(facetCut,position))
         }
         position += 4;
         bytes32 oldFacet = ds.facets[selector];
         // add
         if(oldFacet == 0) {
             // update the last slot at then end of the function
             slot.updateLastSlot = true;
             ds.facets[selector] = newFacet | bytes32(selectorSlotLength) &amp;
             // clear selector position in slot and add selector
             slot.selectorSlot = slot.selectorSlot &amp;amp;amp;amp;amp;amp; ~(
             selectorSlotLength++;
             // if slot is full then write it to storage
             if(selectorSlotLength == 8) {
                 ds.selectorSlots[selectorSlotsLength] = slot.selectorSlot;
                 slot.selectorSlot = 0;
                 selectorSlotLength = 0;
                 selectorSlotsLength++;
             }
         }
         // replace
         else {
             require(bytes20(oldFacet) != bytes20(newFacet), "Function cut to s
             // replace old facet address
             ds.facets[selector] = oldFacet &amp;amp;amp;amp;amp;amp; CLEAR_ADD
         }
     }
 }
```

图 7. 升级函数 许多的力气都花在优化这个函数的 gas 效率上。但是，升级操作是很少用到的，所以不管花多少 gas ，都不可能是 gas 的重度消耗者。 另一个多此一举的例子是用按位操作来替代结构体：

```
uint selectorSlotsLength = uint128(slot.originalSelectorSlotsLength);
uint selectorSlotLength = uint128(slot.originalSelectorSlotsLength &amp;amp;amp
// uint32 selectorSlotLength, uint32 selectorSlotsLength
// selectorSlotsLength is the number of 32-byte slots in selectorSlots.
// selectorSlotLength is the number of selectors in the last slot of
// selectorSlots.
uint selectorSlotsLength;
```

图 8. 使用按位操作来替代结构体 *2020 年 11 月 5 日更新：*我们审计之后，参考实现已经改变，但其底层的复杂性仍然保留着。现在有三个参考实现，让用户更加摸不着头脑，也让对该提议的进一步审核更加困难。 **我们的建议**：

- 始终追求简洁，并尽可能把代码放在链下。
- 撰写一套新标准时，保证代码可读且易于理解
- 在实现优化之前，先分析清楚需求

指针风险

虽然有人主张只要基础指针不同，就不会发生冲突，但是，一个恶意的合约可以用来自另一个实现合约的一个参数造成冲突。所以，冲突是有可能的，原因在于 Solidity 存储变量和影响映射和数组的方式。举个例子：

```
contract TestCollision{

    // The contract represents two implementations, A and B
    // A has a nested structure
    // A and B have different bases storage pointer
    // Yet writing in B, will lead to write in A variable
    // This is because the base storage pointer of B
    // collides with A.ds.my_items[0].elems

    bytes32 constant public A_STORAGE = keccak256(
        "A"
    );

    struct InnerStructure{
        uint[] elems;
    }

    struct St_A {
        InnerStructure[] my_items;
    }

    function pointer_to_A() internal pure returns(St_A storage s) {
        bytes32 position = A_STORAGE;
        assembly { s_slot := position }
    }

    bytes32 constant public B_STORAGE = keccak256(
        hex"78c8663007d5434a0acd246a3c741b54aecf2fefff4284f2d3604b72f2649114"
    );

    struct St_B {
        uint val;
    }

    function pointer_to_B() internal pure returns(St_B storage s) {
        bytes32 position = B_STORAGE;
        assembly { s_slot := position }
    }

    constructor() public{
        St_A storage ds = pointer_to_A();
        ds.my_items.push();
        ds.my_items[0].elems.push(100);
    }

    function get_balance() view public returns(uint){
        St_A storage ds = pointer_to_A();
        return ds.my_items[0].elems[0];
    }

    function exploit(uint new_val) public{
        St_B storage ds = pointer_to_B();
        ds.val = new_val;
    }

}
```

图 9. 存储指针冲突 在爆破中，写入 *B_STORAGE* 基础指针的内容实际上会写入 *my_items[0].elems[0]*，而这个位置又是从 *A_STORAGE* 基础指针中读取得到的。一个恶意的合约所有者可以推送一个看起来无害，但实际上包含后门的升级。 这份 EIP 没有为防止此类恶意冲突提供指导。此外，如果一个指针先被删除然后又被重用，这个重用会导致数据泄漏。 **我们的建议**：

- 操纵底层的存储是风险很大的，所以在设计依赖于底层存储的系统时必须格外小心。
- 使用带有结构体的非结构化存储来实现合约升级，是一个很有趣的思路，但需要详细的文档和指导来说明要在一个基础指针中检查什么东西。

函数遮挡

可升级合约组合的代理合约中通常会有一些函数，遮挡掉应该被 delegate 调用的函数。直接调用（call）这些函数无法被 delegate 传递至实现合约，因为只会使它们在代理合约内执行。此外，相关的代码也是不可升级的。

```
contract Proxy {

    constructor(...) public{
          // add my_targeted_function()
          // as a delegated function
    }

    function my_targeted_function() public{
    }

    fallback () external payable{
          // delegate to implementations
    }
}
```

图 10. 函数遮挡问题的简化例子 虽然这个问题是众所周知的，而且代码也经过 EIP 作者的审核，但我们还是在合约中发现了两个这样的函数遮挡的实例。 **我们的建议**：

- 在开发可升级的合约时，slither-check-upgradeability 来捕捉函数遮挡。
- 这一问题也凸显了重要的一点：开发者也会犯错。任何新的标准都应该包含对常见错误的缓解措施，只要你想干得比定制化的解决方案更好。

如果您对自己的升级策略有任何疑问，请联系我们。我们将竭诚为您服务。

没有合约的存在性检查

合约代码的另一个常见失误是缺乏存在性检查。如果代理合约 delegate 到了一个不正确的地址，或者一个已经被毁弃的实现合约，即使没有执行任何代码，这次调用也会返回成功（见 Solidity 文档）。结果是，调用者不会注意到这个问题，但这种行为可能会破坏掉第三方合约集成。

```
fallback() external payable {
     DiamondStorage storage ds;
     bytes32 position = DiamondStorageContract.DIAMOND_STORAGE_POSITION;
     assembly { ds_slot := position }
     address facet = address(bytes20(ds.facets[msg.sig]));
     require(facet != address(0), "Function does not exist.");
     assembly {
         calldatacopy(0, 0, calldatasize())
         let result := delegatecall(gas(), facet, 0, calldatasize(), 0, 0)
         let size := returndatasize()
         returndatacopy(0, 0, size)
         switch result
         case 0 {revert(0, size)}
         default {return (0, size)}
     }
 }
```

图 11. 不设合约存在性检查的 fallback 函数 **我们的建议**：

- 不管调用什么合约，都要检查合约的存在性。
- 如果担心 gas 的消耗，那就仅在调用不返回数据时执行这种检查，因为反过来（如果会返回一些数据）就表明确实执行了某些代码。

不必要的钻石术语

如前所述，钻石提案的阐述高度依赖于这些新发明的术语。这很容易出错，让审核变得困难重重，而且对开发者也没有任何好处。

1. “钻石（diamond）”是指使用其“雕琢面（facet）”中的函数来执行函数调用的合约。一个钻石可能有一个或多个雕琢面。
2. “雕琢面”一词来自钻石行业，指的是钻石的一个面，或者说平坦的表面。一个钻石有很多个雕琢面。在本标准中，雕琢面指的是带有一个或多个函数、执行一个钻石的功能的合约。
3. “透镜（loupe）”指的是用来观察钻石的放大镜。在本标准中，透镜指的是一个提供函数来观察钻石及其雕琢面的合约。

图 12. 该 EIP 定义的标准术语都是软件工程不相关的东西。 **我们的建议**：

- 使用通用、普及度高的词语，如果没实际用途，就不要去发明黑话。

钻石提案是一条死胡同吗?

如上所述，我们依然相信，社区能从一种标准化的可升级性方案中受益。但当前的钻石提案并不满足我们期待的安全性要求，相比定制化的实现也并没有带来足够多的好处。 不过，这个提案还只是一个草稿，可以变得更简洁、更优秀。即使并没有，它所使用的一些技术，比如查找表和任意存储指针，也值得继续研究。

所以，可升级性是否普遍可行?

这几年来，我们已经审核过许多可升级的合约，也发表了很多分分析。可升级性是困难的、容易出错的，也会带来风险，所以我们总的来说仍然不推荐大家把它当成一种解决方案。但话说回来，如果你决心给合约增加可升级性，你应该：

- 考虑不需要 delegatecall 的升级模式（见 Gemini implementation）
- 透彻地了解现有的解决方案及其局限性：
- Contract upgrade anti-patterns
- How contract migration works
- Upgradeability with OpenZeppelin

如果您对自己的升级策略有任何疑问，请联系我们。我们将竭诚为您服务。

发布于 2023-09-24 11:08 · IP 属地浙江