OpenBrush Contracts Library Security Review

OPENZEPPELIN SECURITY | NOVEMBER 14, 2023 Security Audits

Table of Contents

Summary

Type
: Ecosystem Library

Timeline
: From 2023-07-26
: To 2023-08-29

Languages
: Rust and ink!

Total Issues
: 34 (17 resolved, 5 partially resolved)

Critical Severity Issues
: 0 (0 resolved)

High Severity Issues
: 3 (2 resolved)

Medium Severity Issues
: 11 (5 resolved, 1 partially resolved)

Low Severity Issues
: 7 (3 resolved, 2 partially resolved)

Notes & Additional Information
: 13 (7 resolved, 2 partially resolved)

We reviewed the `Brushfam/openbrush-contracts` repository at the 5533473 commit.

In scope were the following files:

```
contracts
└── src
    ├── access
    │   ├── access_control
    │   │   ├── access_control.rs
    │   │   ├── extensions
    │   │   │   └── enumerable.rs
    │   │   └── mod.rs
    │   ├── mod.rs
    │   └── ownable
    │       └── mod.rs
    ├── finance
    │   ├── mod.rs
    │   └── payment_splitter
    │       └── mod.rs
    ├── governance
    │   ├── mod.rs
    │   └── timelock_controller
    │       └── mod.rs
    ├── lib.rs
    ├── security
    │   ├── mod.rs
    │   ├── pausable
    │   │   └── mod.rs
    │   └── reentrancy_guard
    │       └── mod.rs
    ├── token
    │   ├── mod.rs
    │   ├── psp22
    │   │   ├── extensions
    │   │   │   ├── burnable.rs
```

```
    │   │   │   ├── mintable.rs
    │   │   │   └── wrapper.rs
    │   │   ├── mod.rs
    │   │   ├── psp22.rs
    │   │   └── utils
    │   │       └── token_timelock.rs
    │   ├── psp22_pallet
    │   │   ├── extensions
    │   │   │   ├── burnable.rs
    │   │   │   ├── metadata.rs
    │   │   │   └── mintable.rs
    │   │   ├── mod.rs
    │   │   └── psp22_pallet.rs
    │   ├── psp34
    │   │   ├── extensions
    │   │   │   ├── burnable.rs
    │   │   │   ├── enumerable.rs
    │   │   │   ├── metadata.rs
    │   │   │   └── mintable.rs
    │   │   ├── mod.rs
    │   │   └── psp34.rs
    │   └── psp37
    │       ├── extensions
    │       │   ├── batch.rs
    │       │   ├── burnable.rs
    │       │   ├── enumerable.rs
    │       │   ├── metadata.rs
    │       │   └── mintable.rs
    │       ├── mod.rs
    │       └── psp37.rs
    ├── traits
    │   ├── access_control
    │   │   ├── access_control.rs
    │   │   ├── extensions
    │   │   │   └── enumerable.rs
    │   │   └── mod.rs
```

```
    │   │   │   └── diamond_loupe.rs
    │   │   └── mod.rs
    │   ├── errors
    │   │   ├── access_control.rs
    │   │   ├── diamond.rs
    │   │   ├── flashloan.rs
    │   │   ├── mod.rs
    │   │   ├── ownable.rs
    │   │   ├── pausable.rs
    │   │   ├── payment_splitter.rs
    │   │   ├── psp22.rs
    │   │   ├── psp34.rs
    │   │   ├── psp37.rs
    │   │   ├── reentrancy_guard.rs
    │   │   ├── timelock_controller.rs
    │   │   └── upgradeable.rs
    │   ├── flashloan
    │   │   └── mod.rs
    │   ├── mod.rs
    │   ├── ownable
    │   │   └── mod.rs
    │   ├── pausable
    │   │   └── mod.rs
    │   ├── payment_splitter
    │   │   └── mod.rs
    │   ├── proxy
    │   │   └── mod.rs
    │   ├── psp22
    │   │   ├── extensions
    │   │   │   ├── burnable.rs
    │   │   │   ├── capped.rs
    │   │   │   ├── metadata.rs
    │   │   │   ├── mintable.rs
    │   │   │   └── wrapper.rs
    │   │   ├── mod.rs
    │   │   ├── psp22.rs
```

```
    │   │   ├── extensions
    │   │   │   ├── burnable.rs
    │   │   │   ├── enumerable.rs
    │   │   │   ├── metadata.rs
    │   │   │   └── mintable.rs
    │   │   ├── mod.rs
    │   │   └── psp34.rs
    │   ├── psp37
    │   │   ├── extensions
    │   │   │   ├── batch.rs
    │   │   │   ├── burnable.rs
    │   │   │   ├── enumerable.rs
    │   │   │   ├── metadata.rs
    │   │   │   └── mintable.rs
    │   │   ├── mod.rs
    │   │   └── psp37.rs
    │   ├── timelock_controller
    │   │   └── mod.rs
    │   ├── types.rs
    │   └── upgradeable
    │       └── mod.rs
    └── upgradeability
        ├── diamond
        │   ├── diamond.rs
        │   ├── extensions
        │   │   └── diamond_loupe.rs
        │   └── mod.rs
        ├── mod.rs
        ├── proxy
        │   └── mod.rs
        └── upgradeable
            └── mod.rs
lang
├── codegen
│   └── src
│       ├── accessors.rs
```

```
│       ├── internal.rs
│       ├── lib.rs
│       ├── metadata.rs
│       ├── modifier_definition.rs
│       ├── modifiers.rs
│       ├── storage_derive.rs
│       ├── storage_item.rs
│       ├── trait_definition.rs
│       └── wrapper.rs
├── macro
│   └── src
│       └── lib.rs
└── src
    ├── lib.rs
    ├── macros.rs
    ├── storage
    │   ├── mapping.rs
    │   ├── mod.rs
    │   ├── multi_mapping.rs
    │   └── raw_mapping.rs
    ├── test_utils.rs
    ├── traits.rs
    └── utils.rs
```

Security Review Summary

This security review is not intended to be considered a security audit. Unlike an audit, a security review aims to assess the overall health of the project without conducting an exhaustive line-by-line review of the codebase. Instead, it focuses more on development best practices and potential design issues. However, during this review, many high-severity and medium-severity issues were discovered. Some of these issues are related to the aspects mentioned above, while others are connected to errors and bugs in the code.

rather than only within the OpenBrush library. This library introduces several features that were developed due to the historical lack of functionality in the ink! programming language.

We recommend that the BrushFam team actively collaborates with other teams dedicated to improving the ink! ecosystem, as well as with the ink! core developers. Through collective efforts, a safer, more standardized, and user-friendly language and set of tools can be developed.

OpenZeppelin views this engagement with the BrushFam team as an opportunity to continue working and providing value from both security and user experience perspectives to the Polkadot ecosystem. We believe that it is crucial to contribute to making the transition from other ecosystems where smart contracts are the standard tool to facilitate the development of decentralized applications on Polkadot.

By improving overall security and developers' experience, we hope that more auditors, core developers, and external contributors will consider using ink! and OpenBrush as a viable option for building smart contracts in the Polkadot network.

Caveats of Reviewing Libraries

In libraries, vulnerabilities can take on different forms. First, there are explicit flaws where specific parts of the library, such as contracts or functions, do not perform as intended. These issues can directly impact the security of the library and, by extension, the protocols built upon it.

However, challenges arise with implicit vulnerabilities. While the individual components of the library may work properly in isolation, combining multiple contracts or engaging in certain actions, such as overriding functions, can unexpectedly introduce security risks. This suggests that the vulnerability lies not with the developer's implementation but rather with the design of the library itself.

Severity Classification

- *Impact*: Represents the technical and business damage of a successful attack.
- *Severity*: Evaluates the vulnerability's overall criticality.

The likelihood and impact of potential risks are categorized into three ratings: High, Medium, and Low. The severity of a risk is determined by the combination of its likelihood and impact, and can be classified into four categories: Critical, High, Medium, and Low, as shown in the table. Additionally, if we find other suggestions that are not worth reporting as vulnerabilities, they will be reported under the Notes & Additional Information section.

Severity Level Descriptions

Critical

The issue significantly jeopardizes the security of the protocol built upon the library, leading to a high risk of compromising sensitive information, causing substantial financial losses, or severely damaging the protocol's reputation. This includes explicit flaws in the library's code, implicit vulnerabilities arising from the combination of multiple contracts of the library, or overriding the provided functions, which have a direct impact on the security of the protocol.

High

The issue poses a substantial risk to the security of the protocol built upon the library, potentially compromising sensitive information, causing temporary disruptions, or resulting in moderate financial losses. This encompasses explicit flaws in the library's code or implicit vulnerabilities that may arise when combining multiple contracts of the library or overriding the provided functions, introducing security risks to the protocol.

Medium

The issue presents a moderate risk to the security of the protocol built upon the library, potentially affecting a subset of users or having a moderate financial impact. This includes explicit flaws in the library's code that may not have a significant impact on the overall system but could introduce vulnerabilities to the protocol when combined with other contracts in the library or by overriding existing functions

Low

design of the library and its impact on the protocol's security.

Notes and/or Additional Information

This category encompasses non-security-relevant issues that are worth noting to improve the codebase's overall quality, irrespective of their direct impact on the security of the protocol.

OpenBrush is an open-source smart contract library written in the Rust programming language and the ink! domain-specific language (DSL). It provides developers with a suite of standardized contracts to kickstart their projects in the Polkadot ecosystem.

OpenBrush is based on the `openzeppelin-contracts` Solidity library, maintained by the OpenZeppelin team and originally developed for Ethereum and its derivatives. It implements some of the contracts present in this library to enable developers in the Ethereum ecosystem to transition smoothly into the Polkadot ecosystem, offering a faster and more straightforward development process.

This security review focused on two different areas:

- The smart contracts themselves, in the `contracts` directory
- The `lang` directory, which provides functionality to be used within the smart contracts, such as Rust macros, improving the overall usability of the library

Smart Contracts

The set of contracts implemented by the BrushFam team up to the date of this security review includes:

- `PSP22`: A standard for fungible tokens, similar to the ERC-20 standard in Ethereum. The BrushFam team has added the following extensions to the basic implementation:
  - `Burnable`: Allows PSP22 tokens to be burned.
  - `Mintable`: Allows PSP22 tokens to be minted.
  - `Flashmint`: Enables PSP22 tokens to be flash-borrowed by users or contracts in a single transaction, based on a mint cap.
  - `Metadata`: Provides the ability to add extra information to the token, such as a symbol and a name.
  - `Wrapper`: Allows PSP22 tokens to be wrapped into another PSP22 token, typically to add additional functionality to already-deployed PSP22 contracts.
- `PSP34`: A standard for non-fungible tokens, similar to the ERC-721 standard in Ethereum. The BrushFam team has introduced the following extensions:
  - `Burnable`: Enables the burning of NFTs.

- `PSP37`: A standard for multitokens, akin to the ERC-1155 standard in Ethereum. This standard allows the creation of both fungible and non-fungible tokens within a single contract. The BrushFam team has extended this standard with the following features:
  - `Burnable`: Allows the burning of both fungible and non-fungible tokens.
  - `Mintable`: Allows the minting of new fungible and non-fungible tokens.
  - `Metadata`: Provides the ability to add new attributes to tokens.
  - `Enumerable`: Enables the enumeration of tokens on the chain.
  - `Batch`: Allows users to transfer tokens in batches.
- `Pausable`: A contract that enables users to implement an emergency stop mechanism, which can be triggered at any time by authorized accounts (when used with the Ownable contract mentioned below).
- `ReentrancyGuard`: A contract that helps prevent reentrancy attacks in specific functions.
- `Ownable`: This contract provides a basic implementation of access control through the concept of ownership. The unique owner account of a contract can perform certain administrative tasks.
- `Access control`: A more intricate implementation of access control that allows developers to define distinct privileged roles. Accounts can belong to one or more of these roles to interact with restricted functionality in a contract.
- `PaymentSplitter`: A contract designed to distribute payments in the native token (support for PSP22 tokens is not yet implemented in the library) among a group of accounts.
- `TimelockController`: A contract that introduces a delay in function calls to another smart contract after a predefined time period.
- `Upgradeability`: Using the `set_code_hash` function implemented in ink!, this feature allows users to upgrade the code of a contract at a specific address.
- `Diamond Proxy Pattern`: Based on Ethereum's EIP-2535, this pattern allows users to construct modular smart contract systems that can be upgraded post-deployment.

Macros

Macros from the OpenBrush library streamline the development process of ink! smart contracts. They facilitate contract creation, provide storage utilities, enforce function signatures, and offer custom implementations among other features. Here is a closer look at each macro:

points to use OpenBrush's macros in ink! smart contracts.

The macro's main task is to look for `impl` blocks implementing contract traits. if the trait is recognized (verified by `LockedTrait`), it produces new `impl` items using the `impl_external_trait` function from the `internal` helper file. Afterwards, it pastes the ink! code, allowing further processing by ink!'s macros.

`trait_definition`

This macro is utilized on top of trait blocks. It defines extensible traits for ink! smart contracts. `trait_definition` recognizes specific ink! attributes, manages the segregation of methods based on these attributes, and generates the necessary trait transformations for contract execution.

`modifier_definition`

A procedural macro that primarily acts as a compile-time checker for functions intended for use as modifiers. It expects a specific function signature and enforces a set of rules on the function's parameters. If the function doesn't meet the requirements, a compile-time error is raised.

`modifiers`

Positioned right above the function block, this macro is responsible for applying one or more modifiers to a method. Each modifier's code is nested inside another in reverse order, with the original method body residing in the innermost tier. Modifiers are designed to be used for methods in `impl` sections.

`wrapper`

The macro facilitates the creation of a wrapper type for traits that are defined via the `#[openbrush::trait_definition]` attribute macro. This is particularly useful for scenarios involving cross-contract calls to another contract in a blockchain context. The wrapper essentially modifies traits in such a way that they call external contracts rather than acting on local ones. It is a wrapper for `AccountId` that knows how to do cross-contract calls to another contract.

`storage_derive`

A derive macro that implements the OpenBrush Storage trait for each field within the contract storage struct marked by the `#[storage_field]` attribute. This facilitates accessing these fields via the `self.data::<Type>()` method. It is primarily used for OpenBrush to determine fields accessed by traits.

Direct usage of these methods is discouraged. Helper traits like `StorageAsRef` and `StorageAsMut` are available and provide a user-friendly API (e.g., `self.data().min_delay.get_or_default()` or using turbofish syntax when the compiler can't infer the type `self.data::<Data>().min_delay.get_or_default()`).

`accessors`

The purpose of the macro is to simplify the creation of accessor methods (both getters and setters) for struct fields. Use `#[get]` to indicate fields requiring getters and `#[set]` for those needing setters. The data struct where the macro will be applied should be positioned outside the contract, as the macro needs to expand prior to the expansion of the `openbrush::contract` macro.

`implementation`

The macro implements the OpenBrush library traits that the end users require for their ink! smart contracts. Users are also allowed to override any of the method implementations with `#[overrider]` or `#[default_impl]` attributes.

The `#[overrider]` attribute is used when users want to change a method's behavior with a custom implementation. The `#[default_impl]` attribute is chosen when retaining the default OpenBrush implementation but with added modifiers to the function.

The `implementations.rs` file contains the implementations for every contract in the library.

`storage_item`

The macro implements the `ink::storage_item` macro for the struct, which prepares the type to be fully compatible and usable with the storage. It implements all necessary traits and calculates the storage key for types.

Additionally, it also generates constant storage keys for every mapping or lazy field and inserts them into a type definition.

General Suggestions

The following are general suggestions of potential features and refactors that BrushFam could implement to improve the overall user experience of OpenBrush.

Simplify the Upgradeability Module by Deprecating the Proxy Pattern Approach

The ink! programming language implements a `set_code_hash` function that allows users to replace the contract code at the specified address with new code while preserving the storage layout of the contract as well as the contract address (`AccountId`).

This native feature of the language renders the Proxy Pattern designed by OpenZeppelin unnecessary for implementing upgradeability features, making the upgrade process less error-prone. The Proxy Pattern is not straightforward to implement as it requires an understanding of how proxies work and what delegate calls are.

Consider deprecating the Proxy Pattern approach and instead focusing on improving the `set_code_hash` approach.

Upgradeability Code Improvements and Features

For the sake of upgradeability, it is recommended to follow the `set_code_hash` pattern instead of the Proxy pattern. However, the `UpgradeableImpl` trait currently only implements the permissioned `set_code_hash` function, which can update the code of a given contract.

For completeness, we suggest implementing the following features:

- `AdminUpgradeableImpl`: This contract will manage who can upgrade a contract, allowing roles to be separated between the owner of the contract and the administrator of the upgradeability features. Unlike Solidity and the Proxy Pattern, where the storage layouts of

implementation contract. By implementing an `AdminUpgradeableImpl` contract, it becomes possible to easily separate the roles of the contract owner and the administrator of upgradeability.

- An `upgradeAndCall` function: This permissioned function allows the caller to both upgrade the code of the contract and call a function and/or the constructor to initialize new variables introduced in the newest implementation in the same transaction.
- An `Initialize` contract: This contract sets the initializing function mentioned above to be callable only once.

Upgradeability Tooling Improvements

The upgradeability module lacks the necessary safety checks that can be performed with tools, essential for avoiding any potential issues caused by users.

Storage Layout Check

A CLI/tool is required to verify that there have been no changes that could potentially break the storage compatibility between different implementations. To accomplish this, it is necessary to save the storage layout of each previous implementation and then compare it, variable by variable, with the newest implementation. For more information, refer to our Storage Collisions prevention documentation.

Function Selectors Check

Since a contract might not be aware of the functions (and selectors) implemented in previous versions, it is advisable to implement a tool that checks, during upgrades, whether any function has been removed and if any new one has been added with the same selector. This is crucial to prevent confusion for users interacting with the contract and to guard against the possibility of malicious developers introducing backdoors.

Macro Definitions Could Be at Language-Level Rather than at Library Level

Modifiers Macros

In OpenBrush, the equivalent of the `_` keyword in Solidity is the `body` function. OpenBrush executes code before or after a modifier by calling the `body` function and passing an

other ink! developers will not have access to this functionality unless they also use the OpenBrush library.

Implementation Macros

To inherit functionality from the library contracts into the user's own contracts, the `implementation` macro is used. For example:

```
#[openbrush::implementation(Pausable)]
#[openbrush::contract]
pub mod my_pausable {
    use openbrush::traits::Storage;
    // ...
}
```

To enhance the ink! ecosystem's versatility, these features should be ink! features rather than solely OpenBrush's. In Solidity, the modifier feature is an inherent part of the language, and it is not limited to the OpenZeppelin contracts library; they utilize this feature to implement their own modifiers. Similarly, by moving the implementation macro to ink!, developers can easily inherit functionality from the library contracts, enhancing the overall development experience within the ink! ecosystem. To achieve this goal, the ink! and OpenBrush teams could collaborate and work together to make this feature accessible, ensuring that developers can benefit from it irrespective of whether they are using OpenBrush.

New Contracts

Based on the current functionality in OpenBrush, we think that the following list of features could be relatively straightforward to implement:

- `Ownable2Step`: An extension of the already implemented Ownable contract that adds the ability to change the owner of a contract in two steps, using the `transfer_ownership` and `accept_ownership` functions (see OpenZeppelin's implementation here).
- `VestingWallet`: This contract handles the vesting of funds, whether the native token or PSP22, for a given beneficiary. Its implementation shares similarities with the

PSP22 Capped Amount Is Not Enforced Upon Minting

The `PSP22Capped` extension provides various functions to initialize a cap value, check the cap amount, and verify if a given amount exceeds the cap for PSP22-compliant contracts to limit the number of tokens that can be created.

However, it currently lacks enforcement to ensure that the cap amount is not exceeded during a mint operation. This omission can lead to confusion and potential errors. Users might assume that merely implementing `PSP22Capped` in their contracts will automatically incorporate this feature, similar to other extensions such as `Burnable` or `Mintable`, since the contracts will compile and be deployable because the `_before_token_transfer` function in the PSP22 contract is defined as an empty function.

Additionally, note that the absence of enforcement for this behavior can potentially create issues when using other extensions. For instance, if the cap is not adequately enforced and the user's PSP22 implementation also incorporates the FlashLender extension, it could lead to situations where flash borrowers exceed the cap during flash loans.

To address this, consider overriding the `_before_token_transfer` function defined in the PSP22 implementation within the `PSP22Capped` implementation. By doing so, when the `mint` function is called, the hook will be triggered, and it will guarantee that the current supply plus the minted amount does not surpass the cap.

Update: *Resolved in pull request #141 at commit 4e6e93d.*

PSP22 Is Vulnerable to Double-Spending

The PSP22 contract defines the `approve` function, which permits a user to specify the number of tokens that another user can expend on their behalf.

However, this function is susceptible to the well-known double-spending attack, functioning as follows:

- Alice grants Bob the authority to expend 1000 tokens on her behalf.

- Bob anticipates this action and spends 1000 tokens on Alice's behalf, diminishing the allowance from 1000 tokens to 0.
- Alice's action is executed, establishing the new allowance amount as 500 tokens.
- Bob spends 500 tokens on Alice's behalf, resulting in a total of 1500 tokens spent.

Since the PSP22 implementation introduced the `increaseAllowance` and `decreaseAllowance` functions to modify token allowances for user expenditure, and considering the discussions with the OpenBrush team about potentially redefining the PSP22 or proposing a new fungible token contract standard, consider removing the `approve` function from the PSP22 implementation.

Update: *Acknowledged, not resolved. The Brushfam team stated:*

> *It is not worth to break the standard to incorporate this change.*

Zero Address Management Inconsistencies

In the Substrate ecosystem, public addresses adopt the SS58 encoding. The Base-58 alphabet omits characters prone to confusion when printed, such as the number 0.

Given this, referring to the "zero address" from EVM blockchains can be confusing. In this context, what could potentially use zeros is the public key, which is seldom referenced by regular users. The very idea of a "zero address" does not seamlessly translate from EVM chains to Substrate chains.

Additionally, the address derived from a hex public key composed entirely of zeros varies across different Substrate chains.

Throughout the codebase, the term "zero address" appears numerous times. However, in many instances, it actually refers to Options without an associated `AccountId` ("None") which can lead to potentially harmful misunderstandings.

When using `AccountId` without an associated address, it might be preferable to use "None". It is also recommended to rename or refactor any "zero address" references throughout the codebase for clarity. For instance, consider changing

Update: *Resolved in pull request #136 at commit 36e2a49.*

Medium Severity

Native Ink! Transfers Are Not Traceable in `PaymentSplitter` Contract

The `PaymentSplitter` contract implements a `receive` function used to add funds to the contract. When calling this function, an event is emitted to keep track of all funds received in the contract and the sender's identity.

However, there is no way to keep track of deposits made through the ink! `transfer` function. Consequently, other contracts will need to be aware of the `PaymentSplitter`'s implementation and its functions, as they must know about the aforementioned `receive` function. In general, transfers between protocols should occur in a standardized manner (for instance, using the `transfer` function mentioned above), which enhances interoperability between systems.

For instance, if one of the beneficiaries of the `PaymentSplitter` contract is another `PaymentSplitter` contract when releasing funds to this beneficiary, the funds sent will not be tracked since the contract is using the aforementioned `transfer` function instead of the `receive` function.

Currently, there appears to be no direct solution for initiating an action within a contract when funds are sent to it. This limitation stems from ink!, not OpenBrush. Consider encouraging the ink! team to incorporate a standard `receive` function, which could then be triggered upon someone executing a native `transfer` to a contract.

Update: *Acknowledged, not resolved. The Brushfam team stated:*

> *We will communicate with ink! team to fix the issue, for now will just update the documentation about this issue.*

`PSP22Wrapper`'s `deposit_for`/`withdraw_to` Implementations Might Lead to Stuck Tokens

the latter will own 10 `Token`s, and Alice will own 10 `WToken`s.

Additionally, the contract implements the `_recover` function, which enables anyone to sweep `Token`s mistakenly sent to the `WToken` contract. For example, if Bob sends 5 `Token`s to `WToken`, then the `_recover` function will allow the caller to sweep an amount of:

```
underlying_balance(self) - self.total_supply() = 15 - 10 = 5
```

The `WToken`-to-`Token` ratio will return to 1:1.

However, there are no validations verifying whether the `account` parameter of the `deposit_for` function is not the wrapper itself. This means that if a user sends the wrapper contract's address as a parameter, the `_recover` function will account for these tokens as minted, and they will not be recoverable. For instance, let's say that Bob deposits his 5 Tokens through the `deposit_for` function instead, but sends the wrapper address as a parameter. Then, the `_recover` function will calculate the number of `Token`s to sweep as follows:

```
underlying_balance(self) - self.total_supply() = 15 - 15 = 0
```

Consequently, those tokens will be stuck in the wrapper token forever.

Note that a similar behavior can occur when calling the `withdraw_to` function sending the wrapper contract address as the `account` parameter.

Consider adding a check in the `deposit_for` and `withdraw_to` functions to ensure that the `account` parameter is not the wrapper contract's address, preventing such tokens from getting trapped in the wrapper token indefinitely.

Update: *Resolved in pull request #140 at commit 57d90f8.*

Incorrect `max_flashloan` Amount When Using `FlashLender` and `PSP22Capped` Together

However, when using both the `FlashLender` and `PSP22Capped` extensions, the `max_flashloan` function does not take into account the cap. This means that when users call the function, an incorrect amount will be returned. If users then consider this amount as valid and attempt to perform a flash-borrow operation, the transaction can revert.

Consider adding a function that checks the cap (by calling the `_cap` function) and calling it within `max_flashloan` to obtain the correct flash loan amount.

Update: *Resolved in pull request #142 at commit c1c1fe1.*

Lack of Standard Interface Detection Mechanism Implemented in Contracts

In the Openbrush contracts, there is no standard way to check whether a contract supports specific interfaces or features. In the Ethereum ecosystem, this can be accomplished if a contract is compliant with EIP-165. An interface is a collection of functions that define a set of behaviors. By implementing a standard, contracts can expose a method to query whether they support specific interfaces, making it easier for other contracts to understand their capabilities.

Some advantages of having standardized interface detection are:

1. **Interoperability**: It ensures that contracts can communicate and interact efficiently by checking for compatible interfaces. This is crucial in the decentralized ecosystem, where smart contracts interact with each other without relying on a central authority.

2. **Efficient Function Calls**: Provide a computationally efficient and standardized method to check for function signature support, which reduces the risk of unexpected errors and makes function calls more reliable.

3. **Security**: Prevents erroneous interactions between contracts by allowing them to check whether an interface is implemented before invoking specific functions. This can help prevent costly mistakes and potential security vulnerabilities.

4. **Upgradeability**: Since in the ink! programming language smart contracts can be upgraded, and new functionality can be added to existing contracts, this standardization would allow

Consider defining and implementing a standard interface detection mechanism in the OpenBrush smart contracts to improve their overall usability, safety, and efficiency by providing a consistent way for contracts to query and identify the features they support.

Update: *Resolved in pull request #112 at commit d145cfd. The Brushfam team stated:*

> *PSP61 was already added in the latest versions of OpenBrush but was not in the scope of audit.*

Wrong Event Emitted In `PaymentSplitter::receive` Function

The `receive` function in the `PaymentSplitter` contract is responsible for receiving funds from accounts. However, instead of emitting the `_emit_payment_received_event`, it is emitting the `_emit_payee_added_event`. Although this appears to be intentional based on the comment in the payment splitter trait, this could lead to confusion since added payees will get mixed with accounts that contribute to the total balance of the contract, potentially hindering off-chain services' task of searching and filtering for added payees and funds within the system.

Consider emitting the `_emit_payment_received_event` in the `receive` function event instead.

Update: *Resolved in pull request #139 at commit 7482cfd.*

Lack of Test Coverage Tracking

Test coverage provides an essential metric in software development, reflecting the proportion of the codebase verified by tests. Without a system to monitor this, there is a heightened risk of undetected bugs and a decline in code quality.

To ensure software reliability, consider integrating a test coverage tracking tool into the development process. This tool highlights untested areas, promoting more thorough testing and a stronger end library.

Update: *Acknowledged, will resolve. The Brushfam team expressed that they will resolve the issue.*

Wrong Argument Validation in Modifier Generation

flawed methods to be accepted as valid modifiers.

Additionally, when matching on additional arguments, the error message implies a necessary check that the argument is passed by value and implements the `Clone` trait. However, the check only ensures that the argument is not a reference.

Consider correcting the argument validation in the `generate` function to correspond to the intended behavior.

Update: *Partially resolved in pull request #144 at commit b48968d. The client did not take any action on the inaccurate error message. They expressed that checking if `Clone` trait is implemented, it is not realistic to do in Rust. Nonetheless, they should modify the error message to more accurately reflect what the code is actually checking.*

Macro Implementations Missing Adequate Docstrings

In the `codegen` folder, the files that hold the implementations for each macro are missing proper docstrings. Given the complexity and length of the code, adding detailed docstrings is strongly recommended. This will make it easier for readers and developers to understand the inner workings of the codebase.

Update: *Acknowledged, will resolve. The Brushfam team expressed that they will resolve the issue. Progress can be tracked on issue #155 in the repository's issue tracker.*

Error-Prone and Unnecessary Push-Payment Mechanism in `PaymentSplitter` Contract

The `PaymentSplitter` contract implements a push-payment mechanism through the internal `_release_all` function. This function iterates through all the payees added to the contract and transfers the corresponding amount to each of them.

However, this function presents the following problems:

- If one of the beneficiaries has already claimed their part and their releasable amount is 0, then even if other beneficiaries have yet to receive their part, the whole transaction will

Even though this function is not public and therefore not accessible by default, library users might want to call this function from custom public functions, potentially triggering the aforementioned errors.

Consider removing the `_release_all` function, documenting why there is no push-payment implementation provided by the library, and encouraging the use of the `release` function's pull-payment mechanism instead.

Update: *Resolved in pull request #145 at commit e5de183.*

No Access Control on Setters Produced by `accessors` Macro

The `accessors` macro implements getters and setters for struct fields, which can be called externally in a smart contract context since they are annotated with `#[ink(message)]`. It is vital to ensure that any sensitive or critical state variables cannot be trivially accessible or modifiable using these generated functions, as it could expose vulnerabilities in the contract.

Currently, modifiers cannot be attached to these setters, and there is inadequate documentation cautioning users about this macro's use.

Consider modifying the macro setter attribute to incorporate access control capabilities. Furthermore, it would be ideal for the setters to come with access control mechanisms enabled by default and only allow them to be disabled if users intentionally choose to do so.

Update: *Acknowledged, will resolve. The Brushfam team stated that they will resolve the issue:*

> *The macro is going to be reworked soon, the suggestion will be implemented there, progress can be tracked here: https://github.com/Brushfam/openbrush-contracts/issues/135.*

Fetching Code Directly From Repository

The openbrush-contracts repository makes its public crates available only through direct fetching from the repository. This introduces potential risks of instability and security breaches. Using a

Consider using a crates registry for all the public crates within the codebase. Crates.io, for example, offers these benefits while also providing continuous access to crates, protecting against the pitfalls of repository deletions.

Update: *Acknowledged, will resolve. The Brushfam expressed that they will resolve the issue:*

> *We will be uploaded to registry in a new release, could not be uploaded before because OpenBrush was using* `pallet-assets-chain-extension`*, which is a git dependency. In new versions we are planning to move* `psp22_pallet` *in another repo and publish OpenBrush. Issue: https://github.com/Brushfam/openbrush-contracts/issues/154.*

Low Severity

Flash Fee Burned Instead of Sent to a Beneficiary

The `flashloan` function within the `FlashLenderImpl` trait is responsible for sending the flash-minted tokens to the flash receiver and calculating the fee that the borrower must pay. However, despite the fee being used for calculations, the `FlashLender` implementation never sends it to a flash fee receiver. Instead, it burns the fee without an apparent reason.

Consider defining a `flash_fee_receiver` function that returns the `AccountId` of the fee beneficiary, and subsequently sends the fee amount to that account. Alternatively, consider documenting the rationale behind burning the fees if there is a specific reason for this approach.

Update: *Resolved in pull request #157 at commit dcf5843.*

`PaymentSplitter` Beneficiary Amounts to Release Are Not Retrievable

The `PaymentSplitter` implementation lacks a public function for beneficiaries to perform a read-only query of the current amount of funds they can release. As a result, they may be uncertain about whether the transaction cost will exceed the amount to be released, particularly when the releasable amount is low. Additionally, if there are no funds available for release, the transaction will revert, causing users to incur costs unnecessarily.

they can release, even when they have no funds available for release.

Update: *Resolved in pull request #146 at commit 4e81921.*

Missing Documentation for End-to-End Tests

Simulating on-chain interactions is an essential part of the testing process during development for blockchain applications. In ink!, E2E tests are used for this purpose. The OpenBrush examples use the E2E tests frequently, where functions are annotated with the `#[ink_e2e::test]` attribute. Due to ongoing issues with `cargo-contract` and `substrate-contracts-node`, particular versions of toolchain components are currently required to run these types of tests, which may fail when using the latest versions. Consider adding documentation for the recommended setup to run these tests.

Update: *Resolved in pull request #148 at commit de87bf6.*

Ownership Can Be Transferred to "None"

This comment on the `Ownable` trait states that the `transfer_ownership` function should produce a `NewOwnerIsZero` error if the new owner is "None".

However, the implementation of `transfer_ownership` never produces this error, allowing any address to be used as the transfer recipient. Consider modifying the ownership transfer logic to produce this error if the provided `new_owner` argument is "None".

Update: *Partially resolved in pull request #137 at commit 5088162. The inline comment in the* `ownable` *trait still references the* `NewOwnerIsZero` *error.*

Checking for Unused Attribute in Contract Macro

In the `consume_traits` function of the `contract` macro, `ItemTrait`s are matched against the provided input and checked for the `trait_definition` attribute. However, the structure of the contracts do not have `TraitItem`s with this attribute inside the contract module due to the way that the `implementation` is generated. Hence, the code block is never entered. Consider refactoring the contract macro by removing unused code to improve efficiency and readability.

> *described inside contract module, so it is better to leave it there.*

All Traits Loaded Into End-User Projects

The OpenBrush library uses Rust `features` to allow users to selectively integrate contracts and modules into their codebases, thereby only importing the contracts they need. However, the system operates differently for traits. Traits are not governed by features and are loaded automatically. This means that regardless of the features chosen by the end user, all traits get loaded. This leads to unnecessary overhead at compilation time, especially for trait-related macros (`trait_definition` and `wrapper`). Moreover, metadata files are produced for all traits, even for contracts that are not utilized in the user's project.

Consider reorganizing the `traits` folder so that the traits within also use Rust `features`. This ensures that only relevant traits, those corresponding to the contracts users have integrated from the library, are loaded into user codebases.

Update: *Acknowledged, will resolve. The Brushfam team expressed that they will resolve the issue:*

> _ We will think about appropriate way for resolving this issue._

Code Repetition

Throughout the codebase, there are instances of similar or identical code being used that could be consolidated. For example:

In the `PSP22PalletMetadataImpl` Trait

The `token_name`, `token_symbol`, and `token_decimals` functions access the pallet assets in the following manner:

```
fn X(&self) -> Option<String> {
        let self_ = self.data();
        let name = self_
            .pallet_assets
            .get_or_default()
```

This code is repeated in each function. Consider centralizing this logic by creating `metadata_symbol`, `metadata_name`, and `metadata_decimals` functions and calling them in each corresponding function.

In `accessors.rs`

- The `generate_struct` is nearly identical to the one used in `storage_item`. Consider reusing it as much as possible.
- The only difference between the `extract_get_fields` and `extract_set_fields` functions is the parameter passed to `is_ident` ("get" for the former and "set" for the latter). Consider consolidating both functions in one single function and adding one parameter to distinguish between "set" and "get".

In `implementation.rs` / `contract.rs`

Both `implementation` and `contract` macros check that the module is not an out-of-line module declaration, which will be passed twice during code generation. Consider doing this check a single time.

Update: *Partially resolved in pull request #150 at commit dc91955. The client resolved code repetition in `PSP22PalletMetadataImpl` and part of `accessors.rs` but did not take any action on `implementation.rs` / `contract.rs`. The Brushfam team stated:*

> *Out of module declaration is more extracting values, not a check. Generating structs differs for working with fields, so we left them as they are for now.*

Notes & Additional Information

Using the Full License in Each File Adds Unnecessary Bulk

Throughout the OpenBrush repository, most files use the complete license, which can be redundant and potentially distracting for users reviewing the code.

Consider adopting an SPDX license identifier as an alternative to embedding the entire license in every file.

Lack of License Identifier

Some files within the `lang` folder do not have a license identifier. For example:

- `accessors.rs`
- `implementations.rs`

To avoid legal issues regarding copyright and follow best practices, consider adding the license identifier at the beginning of each file.

Undocumented Metadata Usage

When compiling a project that employs the Openbrush library, a directory named `__openbrush_metadata_folder` is produced, housing metadata files for each trait in the library.

These metadata files lack extensions and essentially provide an unformatted copy of the corresponding trait. The intent or utility of these metadata files remains unclear, and their current format does not align with conventional expectations for metadata - which is data describing other data.

It would be beneficial to introduce documentation clarifying the intention of these metadata files. Additionally, if feasible, it is recommended to revise their format to better match users' typical semantic expectations for metadata files.

Update: *Acknowledged, will resolve. The Brushfam team expressed that they will resolve the issue. Progress can be tracked on issue #159 in the repository's issue tracker.*

TODO Comments

There are "TODO" comments in the codebase that should be tracked in the project's issues backlog. See for example line 41 of the `PaymentSplitter` trait.

During development, having well-described "TODO" comments will make the process of tracking and solving them easier. They should include a brief description of the pending task, and a link to

Update: *Acknowledged, will resolve. The Brushfam team expressed that they will resolve the issue:*

> _ We will add a PSP22 Payment Splitter contract soon and remove the TODO._

`non_reentrant` Flag Should Be Boolean

The `non_reentrant` modifier stores the reentrancy status of the contract in the `status` variable, which is defined as a `u8`. When a function of the contract is accessed for the first time, the initial value of `status` is set to 0, and the modifier updates it to 1. Subsequently, if the function is re-entered, the modifier checks whether the `status` variable is 1 or 0, and reverts if it is 1.

In Solidity, choosing `u8` over boolean makes sense since transitioning from `false` to `true` incurs higher costs than going from `1` to `2`. Turning `false` to `true` in Solidity involves changing `0` to `1`, which requires writing to a cold storage slot, incurring additional expenses compared to writing to a warm storage slot when going from `1` to `2`. However, in Rust, there is no concept of "cold storage" and "warm storage" slots, and writing to an empty variable does not impose higher costs than writing to a non-empty variable.

To improve readability, expressiveness, and memory alignment, consider changing the variable type of the `status`, `NOT_ENTERED`, and `ENTERED` constants to boolean.

_**Update:** Resolved in pull request #143 at commit 6f94147.

Outdated Or Wrong Website References

- On the OpenBrush hompage, the "How to Use" section allows users to download an example crate using either psp22, psp37, or psp34 features from the OpenBrush library. In the generated `Cargo.toml` file, the OpenBrush dependency is included with a "v" prefix in the tag name, which is not being used in the OpenBrush repository. Consider updating the tool to prevent users from encountering errors when building contracts.
- Within the section on the OpenBrush website that pertains to upgradeability, broken links are present which direct to the ink! Solidity repository. Consider checking that all links on the

`873d0ad` *and pull request #12 of the* `learn-page` *repository at commit* `c27d26f`.

Outdated Copyright

Throughout the codebase, the copyright dates currently cover the period from 2012 to 2022. They should be updated to include the current year.

Update: *Partially resolved in pull request #147 at commit a0f48bc. The copyright is still outdated in the files within the* `lang` *and* `tests` *folders.*

Inaccurate Directory Name

The Diamond contracts reside in the `upgradeability` directory, even though the diamond proxy might not be inherently upgradeable. Consider moving these contracts to a different directory with a more precise name, bearing in mind that renaming the `upgradeability` directory to `proxy` might not be entirely accurate, as one of the upgradeability approaches employs `set_code_hash` instead of a proxy pattern.

Update: *Resolved in pull request #156 at commit fbff63e2.*

Typographical Error

In line 52 in the PSP34 Burnable example: "burn_wokrs" should be "burn_works".

Consider addressing this typographical error and thoroughly examining the codebase to rectify any additional ones.

Update: *Resolved in pull request #151 at commit 36b4c5c.*

Inconsistent Adoption of Underscore Prefix

In the Rust community, the underscore prefix is commonly used to signify intentionally unused variables. This prevents the compiler from issuing warnings for such variables.

While some functions in the codebase adhere to this convention, others do not. Some examples include the `_attrs` parameter in the `generate` function of the `contract` macro and the

For clarity, consider refactoring these examples and any other instances in the code to adhere to the established convention consistently.

Update: *Resolved in pull request #152 at commit 7128b63.*

Use of Non-Explicit Imports

Macros such as `implementation` and `contract` utilize the `*` symbol to import modules from certain crates. This approach might result in warnings for unused imports and potential ambiguities. For clarity and precision, consider importing only the necessary components explicitly and avoid using the `*` symbol.

Update: *Acknowledged, will resolve. The Brushfam team expressed that they will resolve the issue:*

> _ We will think about appropriate way to resolve, especially whether it is really needed, since some imports are generated by macros and it is hard to keep track on all of them. Also, it could be hard to read the code with lots of imports._

Redundant Bindings in the Codebase

Throughout the codebase, there are variable bindings that seem unnecessary. Instead of creating new variables, the original variables can be used directly. For example:

- The `ink_module` variable within the `implementation` macro can be directly used, eliminating the need for the additional binding `input`.
- In `contract.rs`, the `_attrs` and `ink_module` parameters are moved to `attrs` and `input`, respectively.
- The `result` variable at the end of the `accessors` method is an unnecessary binding. Consider using the `quote!` macro directly.

Consider removing any redundant bindings within the codebase to improve its overall clarity.

Update: *Resolved in pull request #153 at commit f73a53e.*

Misleading Error Message

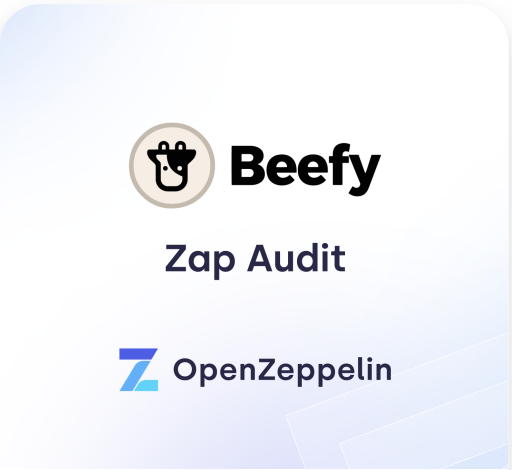
single path or a `MetaList` struct instance (or more precisely, something that matches `NestedMeta::Path`). Consider modifying the error message to more accurately describe the cause of panic.

Update: *Resolved in pull request #158 at commit 64b5647.*

The OpenZeppelin team has thoroughly reviewed the Openbrush contracts library. We present the implemented functionality and introduce macros. We also provide suggestions regarding design choices, tooling, upgradeability, and macro improvements.

During our review, we found three high-severity issues, as well as several lower-severity issues. We have provided recommendations to resolve and mitigate these issues, as well as general improvements to the codebase to improve maturity and documentation with the goal of enhancing developer experience.

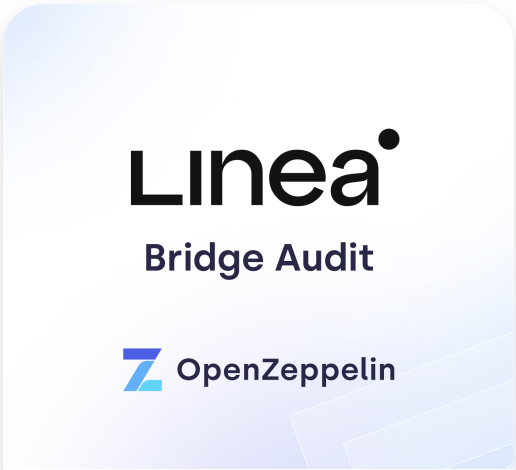
Related Posts



Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

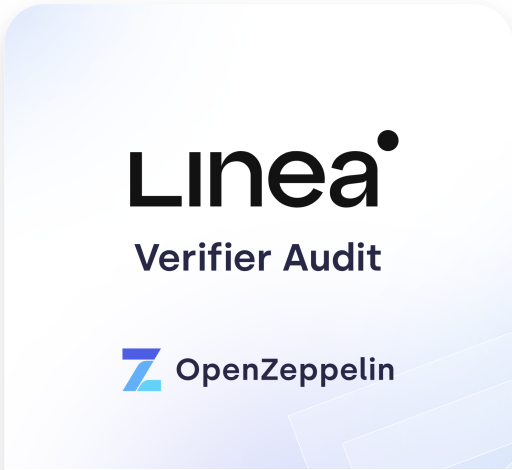
Security Audits



Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits



Linea Verifier Audit

The Linea blockchain (L2) is a ZK-rollup on top of the Ethereum mainnet (L1). The L1 periodically...

Security Audits

OpenZeppelin