OpenZeppelin

Stylus Rust SDK Audit

OPENZEPPELIN SECURITY | SEPTEMBER 9, 2024

Rust

Table of Contents

Summary

Type
Smart Contract Language
Timeline
From 2024-07-08
To 2024-08-09
Languages
Rust

Total Issues
36 (27 resolved, 1 partially resolved)
Critical Severity Issues
2 (1 resolved, 1 partially resolved)
High Severity Issues
2 (2 resolved)
Medium Severity Issues
10 (7 resolved)

Notes & Additional Information
10 (9 resolved)

Scope

We audited the OffchainLabs/stylus-sdk-rs repository at commit 62bd831.

In scope were the following files:

```
 stylus-proc
└── src
    ├── calls
    │   └── mod.rs
    ├── lib.rs
    ├── methods
    │   ├── entrypoint.rs
    │   ├── error.rs
    │   ├── external.rs
    │   └── mod.rs
    ├── storage
    │   ├── mod.rs
    │   └── proc.rs
    └── types.rs
stylus-sdk
└── src
    ├── abi
    │   ├── bytes.rs
    │   ├── const_string.rs
    │   ├── export
    │   │   ├── internal.rs
    │   │   └── mod.rs
    │   ├── impls.rs
    │
    ├── block.rs
    ├── call
    │   ├── context.rs
    │   ├── error.rs
    │   ├── mod.rs
    │   ├── raw.rs
    │   ├── traits.rs
    │   └── transfer.rs
    ├── contract.rs
    ├── crypto.rs
    ├── debug.rs
    ├── deploy
    │   ├── mod.rs
    │   └── raw.rs
    ├── evm.rs
    ├── hostio.rs
    ├── lib.rs
    ├── msg.rs
    ├── prelude.rs
    ├── storage
    │   ├── array.rs
    │   ├── bytes.rs
    │   ├── map.rs
    │   ├── mod.rs
    │   ├── traits.rs
    │   └── vec.rs
    ├── tx.rs
    ├── types.rs
    └── util.rs
examples
├── erc20
│   └── src
│         └── main.rs
├── erc721
│     └── src
│         ├── erc721.rs
│         ├── lib.rs
│         └── main.rs
└── single_call
      └── src
          ├── lib.rs
          └── main.rs
mini-alloc
├── src
│     ├── imp.rs
│     └── lib.rs
└── tests
      └── misc.rs
```

System Overview

Arbitrum Stylus introduces a paradigm shift in smart contract development by enabling programs to be compiled to WebAssembly (WASM) and deployed on-chain, seamlessly coexisting with traditional smart contracts written in common EVM languages like Solidity. This language-agnostic approach opens up new possibilities for developers, enabling them to use their preferred programming languages while maintaining full ABI compatibility with the Ethereum ecosystem.

One of the most remarkable aspects of Stylus programs is their exceptional performance and cost-effectiveness. These programs are orders of magnitude cheaper and faster to execute than traditional EVM-based smart contracts while also being fully EVM compatible. This breakthrough enables Stylus programs to interact seamlessly with existing Ethereum smart ecosystem.

Stylus SDK for Rust

This powerful toolkit enables developers to write programs for Arbitrum chains in Rust. Rust's combination of performance, safety, and modern features makes it ideal for developing robust and efficient smart contracts. The Stylus SDK for Rust provides a comprehensive set of tools and abstractions that simplify the process of creating Stylus programs. It offers a familiar development experience for Rust programmers while integrating seamlessly with the Arbitrum ecosystem. Some of the features available in the SDK include:

- Generic, storage-backed Rust types for programming Solidity-equivalent smart contracts with optimal storage caching.
- Simple macros for writing language-agnostic methods and entry points.
- Automatic export of Solidity interfaces for interoperability across programming languages.
- Powerful primitive types backed by the feature-rich Alloy.

Procedural Macros

The Stylus Rust SDK leverages several powerful procedural macros to streamline smart contract development and ensure seamless integration with the EVM ecosystem. These macros automate complex tasks such as trait implementation, method exposure, storage management, and inter-contract communication, allowing developers to write idiomatic Rust code while maintaining full compatibility with contracts made with EVM-compatible languages.

`entrypoint`

This macro defines the entry point for Stylus execution. It implements the `TopLevelStorage` trait and is typically used to annotate the top-level storage struct. This macro sets up the necessary boilerplate for handling incoming calls, parsing calldata, and execution model.

`external`

This macro is used to make methods "external" so that they can be called by other contracts. It implements the `Router` trait for the annotated `impl` block. The macro handles the complexities of ABI encoding and decoding, method selector generation, and integration with the Stylus VM's calling conventions. It also manages purity annotations (`pure` , `view` , `payable`) and can infer these based on the method signature if not explicitly specified. This macro can have the `#[inherit]` attribute to implement inheritance-like behavior, allowing a contract to include methods from parent contracts.

`sol_interface`

It transforms Solidity interface definitions into Rust structs and methods, enabling seamless interaction with other contracts. The macro handles method generation, type conversion, function selector calculation, and call context management. It supports various call types (`pure` , `view` , and `payable`) and accommodates reentrancy concerns. By automating the creation of ABI-compatible Rust code, it simplifies cross-contract communication while maintaining type safety and idiomatic Rust practices. This macro acts as a translator, enabling Rust contracts to integrate with the broader EVM ecosystem.

`solidity_storage`

This attribute macro is applied to Rust structs to enable their use in persistent storage within a smart contract. Each field in the struct must implement the `StorageType` trait which ensures EVM storage model compatibility. Applying this macro allows developers to define storage layouts directly in Rust, with the fields mapping to the corresponding storage slots in the EVM. This macro ensures that the storage layout of the Rust structs aligns with that of Solidity, facilitating seamless upgrades and interactions with existing Solidity contracts. It supports nested structs and various storage types like `StorageAddress` , `StorageBool` , and custom types implementing `StorageType` .

This macro enables the definition of Rust structs using Solidity-like syntax. It ensures that the storage layout of these structs is identical to their Solidity counterparts. This macro simplifies the transition from Solidity to Rust by allowing developers to reuse their Solidity type definitions directly in Rust, maintaining compatibility with existing storage layouts. This macro uses `solidity_storage` macro under the hood.

`derive_solidity_error`

This macro allows Rust enums to be used for error handling in contract methods. It enables enums to be automatically converted into Solidity-compatible error messages that can be returned by smart contract functions. Under the hood, the macro works by implementing `From<YOUR_ERROR>` for `Vec<u8>` along with printing code for `export-abi`.

`derive_erase`

This macro automatically implements the `Erase` trait for a struct. It generates an `erase()` method that calls `erase()` on each of the struct's fields. This allows for easy clearing of complex storage structures in Arbitrum Stylus smart contracts, ensuring that all fields are properly erased without manual implementation. The macro cannot implement Erase for types that do not support it, such as mappings.

Core Modules

The following modules are contained within the `stylus-sdk` folder to facilitate smart contract development and interaction with the Stylus WASM module:

`abi`

The `abi` module provides functionality for encoding and decoding data according to the Ethereum Application Binary Interface (ABI) specification. It enables a two-way mapping between Solidity and Rust types, allowing for interoperability between Rust and Solidity responses back into Rust types. The `abi` module also includes utilities to generate method selectors and export Solidity interfaces, treating `Vec<u8>` as `uint8[]` in Solidity and using the `Bytes` type for Solidity `bytes`. This functionality is essential for communication between Rust-based smart contracts and those written in Solidity.

`call`

The `call` module manages interactions with external contracts by handling the execution context and providing mechanisms for standard and raw contract calls. It allows developers to specify gas limits and call values, and access contract storage. The module includes caching strategies to optimize repeated state access and features for safe execution during re-entrant calls. By managing these aspects, the `call` module enables Rust-based contracts to interact with Ethereum contracts, facilitating contract communication and data exchange.

`deploy`

The `deploy` module facilitates the deployment of contracts on the Arbitrum network. It includes functionalities for both standard and raw deployments, offering flexibility and control over the deployment process. The `raw.rs` file provides lower-level deployment functions, allowing for more granular control and potentially unsafe operations. This module supports setting deployment parameters, handling the deployment process, and ensuring correct contract initialization.

`storage`

The `storage` module provides a comprehensive framework for managing smart contract storage, featuring abstractions for common data structures like arrays, bytes, maps, and vectors. It supports both basic and complex storage operations through a set of defined traits, ensuring proper data handling and access. The module allows developers to define custom storage logic by implementing the `StorageType` trait, enabling more advanced data manipulation while providing persistent storage access in the Rust-based contracts. Stylus contracts run on a virtual machine that shares the same EVM State Trie, allowing access to using Rust's borrow checker, preventing unsafe aliasing of storage.

`hostio`

The `hostio` module facilitates interactions between Rust-based smart contracts and the host environment on the Arbitrum blockchain. It provides a set of functions for managing contract state, executing calls, and handling I/O operations via a foreign-function interface to the Stylus WASM VM which ultimately communicates with the core blockchain. The `wrap_hostio` macro is a key component of this module, designed to simplify and streamline the process of defining host functions. It wraps low-level host operations in Rust-safe abstractions, automatically generating bindings to interact with the blockchain. Several single-file modules in the Stylus SDK provide typical blockchain interactions extensively using the `wrap_hostio` module, such as:

- `block` : Provides access to Ethereum block information, including properties like the block number, timestamp, and miner details. It serves as an interface for retrieving data about the current or past blocks on the blockchain.
- `contract` : Facilitates interactions with other contracts, enabling function calls and access to contract metadata. It allows contracts to perform operations such as balance checks and contract code retrieval.
- `crypto` : Offers cryptographic functions and utilities, such as hashing algorithms and signature verification.
- `evm` : Interfaces with the EVM, managing execution resources and logging. It includes utilities to query remaining gas and ink (Stylus-specific compute units), emit logs both in raw and alloy-typed forms, and manage memory growth.
- `msg` : Handles message-passing operations, providing information about the current transaction, such as the sender, value, and gas. It allows contracts to interact with transaction data and control flow.
- `tx` : Provides transaction-related functionality, including accessing transaction details like gas price and origin. It enables contracts to work with transaction-specific data and

Mini Allocator

This allocator is key to the Stylus ecosystem, optimized for wasm32 targets like Arbitrum Stylus. It uses a minimal bump allocator strategy, prioritizing simplicity and efficiency. Notably, mini-alloc never deallocates memory, which is ideal for scenarios with tight binary size constraints where it is acceptable to leak all allocations. This design choice enhances performance, aligning with Stylus' focus on optimizing for blockchain environments where traditional memory management can add unnecessary overhead.

Examples

The Arbitrum Stylus SDK repository contains three example crates of common smart contract designs: `erc20`, `erc721`, and `single_call`.

- **`erc20`**: Demonstrates an ERC-20 token contract with functionalities like minting, transferring, and checking balances. It includes methods for token transfers and managing allowances.
- **`erc721`**: Illustrates an ERC-721 NFT contract implementation, covering minting, transferring, and querying ownership of unique tokens. It also handles token metadata and ensures compatibility with the ERC-721 standard.
- **`single_cal`**: Showcases a simple contract for making a single call to another contract, demonstrating inter-contract communication within the Arbitrum environment using the Stylus SDK.

Trust Assumptions

- **SDK Integrity**: It is assumed that the Stylus SDK itself is free from malicious code, such as backdoors or other vulnerabilities that could circumvent the behavior of the underlying that developers can rely on its behavior as intended.

- **WASM VM and Stylus Module Compliance**: It is assumed that the WASM VM and Stylus module strictly follow the consensus rules as outlined by the core EVM part of Arbitrum. This ensures that the execution within the Stylus environment is consistent with the broader consensus rules governing the Arbitrum network, maintaining the integrity and reliability of smart contract execution.
- **Third-Party Dependencies**: Any third-party libraries or dependencies integrated with the Stylus SDK are trusted to be secure and regularly updated to mitigate known vulnerabilities.
- **Smart Contract Interfaces**: It is assumed that the ABIs generated by the Stylus SDK for contracts are correct and that smart contracts behave in an expected manner, similar to Solidity contracts. This means that from an external perspective, the contracts should exhibit predictable and standard behavior, ensuring compatibility and reliability for the users interacting with them.

Critical Severity

Storage Layout is Inconsistent with Solidity

The documentation asserts that struct fields in Stylus will map to the same storage slots as in EVM programming languages and that the layout will be identical to Solidity's. This suggests that upgrading from Solidity to Rust should not cause misalignment in storage slots, thereby implying an easy transfer of type definitions.

However, Stylus does not handle inherited storage in the same manner as Solidity. For example, consider the following Solidity code:

```
contract Parent {
    bool a = true;
    bool b = true;
}
contract Child is Parent {
    bool c = true;
    bool d = true;
}
```

In this case, storage slot 0 contains `0x01010101`. In contrast, the equivalent Stylus code uses a `borrow` clause that uses a new storage slot and does not pack the state variables, which results in a different storage layout:

```
// Snippet of parent.rs
sol_storage! {
    pub struct Parent {
        bool a;
        bool b;
    }
```

```
// Snippet of lib.rs
sol_storage! {
    #[entrypoint]
    struct Child {
        #[borrow]
        Parent parent;
        bool c;
        bool d;
    }
}

#[external]
#[inherit(Parent)]
impl Child {
// ...
}
```

Given the same value set, this results in `0x0101` being in slot 0 and `0x0101` being in slot 1.

This discrepancy is critical for projects using proxy patterns that are migrating from Solidity to Stylus as it could lead to storage layout misalignment, potentially overwriting state variables.

Consider refactoring the `solidity_storage` macro to mirror Solidity's behavior. Alternatively, consider updating the documentation to accurately reflect the current behavior and avoid misleading developers.

Update: *Resolved at commits a99d8c5 and 6e3a62e. Inline documentation has been added to highlight the discrepancy with Solidity storage layout.*

Lack of Selector Collision Check in External Macro

a Stylus smart contract as Ethereum-compatible methods. To achieve this, it analyzes an implementation block, processes each method to generate its selectors, and then uses these selectors to route incoming calls to the appropriate Rust functions.

However, the macro does not validate the uniqueness of these selectors. This oversight can lead to selector collisions, resulting in methods becoming unreachable. In addition, developers can manually set a custom selector for a function using `#[selector(id = <NUMBER_THAT_GENERATES_THE_COLLISION>)]`, potentially duplicating existing selectors intentionally or unintentionally. This vulnerability can be exploited to create malicious contracts, such as honeypots, wherein methods are intentionally made unreachable.

Consider implementing a validation mechanism within this macro to ensure that all selectors are unique, preventing selector collisions and enhancing contract reliability and security. One approach could be to add a `#[deny(unreachable_patterns)]` statement on the `route` function to prevent unreachable methods.

***Update:** Partially resolved at commits a78decd and 0d50c1d. The suggested solution of using `#[deny(unreachable_patterns)]` is insufficient, as it only checks for unreachable patterns within the specific match statement where it is applied. It does not account for selector collisions with functions from inherited contracts, allowing these conflicts to go undetected. Offchain Labs has added inline documentation to highlight this potential issue, and warnings will also be included on the documentation website. The Offchain Labs team is exploring alternative approaches to implementing inheritance for Stylus.*

High Severity

Potential Misuse of `sol_interface` Macro

The `sol_interface` macro allows developers to seamlessly call Solidity contracts from Stylus smart contracts using their native interfaces. However, this macro can be easily manipulated to mislead users about the actual state mutability of functions. For example, a homoglyphs, or employ other tricks to deceive users into believing that a function is non-mutating even though the macro treats that function as state-changing. This fallback to treating the function as state-changing occurs if no valid mutability keyword is detected, thereby opening the doors to unintentional errors and hard-to-detect scams.

To mitigate this issue, consider implementing stricter validation within the macro to ensure that only the correct mutability keywords are permitted. This will help prevent both intentional misuse and accidental errors, enhancing the overall security and reliability of the macro.

Update: *Resolved at commit a474666.*

Custom Selectors Could Facilitate Proxy Selector Clashing Attack

Stylus allows developers to modify the selector for a given function using the `selector` attribute, which can accept either a `string name` or a `u32 ID`. This feature facilitates changing the name of a function while maintaining the same selector, simulating Solidity overloading capabilities and enabling the creation of language-agnostic contract standards.

However, providing an `ID` may result in a function selector that exists in both an implementation contract and its proxy contract. Thus, a user may call a proxy contract function with a selector matching the intended implementation contract function instead, causing unintended code execution and precluding the user from accessing the functionality of the implementation contract. This scenario can occur if an `ID` integer is defined in such a way that, when converted to hex, it matches the function selector in the other contract, whether it is the implementation or the proxy.

This vulnerability enables malicious projects to create hard-to-detect backdoors. In contrast to Stylus, Solidity requires finding function signatures with matching selectors before exploiting this vulnerability, which is not trivial. Even if such function signatures are found, they are likely to raise red flags due to nonsensical names in the codebase. In Stylus, this attack is harder to detect.

Custom selectors can also confuse third-party monitoring or indexing services that use function selectors to identify specific functions. These services may rely on standard selectors, which are part of the standards or belong to community databases such as the 4byte directory. If contracts use custom selectors, these services may fail to recognize and monitor transactions, leading to errors.

Given these risks, reconsider the `ID` option and consider accepting only the `name` instead. If the benefits of custom selectors do not outweigh the risks, consider removing them entirely.

Update: *Resolved at commit 795d376.*

Medium Severity

Function Overriding Does Not Enforce Mutability Rules

In Stylus, when an inheriting contract overrides a base function in its parent, there are no checks to enforce any mutability rules. For example:

- A function that does not modify the state in the parent (e.g., a `view` function) can be overridden by one that does so in the child.
- A function marked `payable` in the parent can be overridden by a non-payable function in the child, causing the child function to revert upon receiving ETH.

From a developer standpoint, this can lead to unexpected behavior and error-prone contract development since this does not match the rules enforced in Solidity, which mandate stricter mutability enforcement. To align with Solidity's mutability rules and avoid unexpected behavior, consider refactoring the logic in the `external` macro by implementing checks on the mutability attributes in parent functions.

Update: *Resolved at commit 1984d8a.*

If two or more interface definitions are present in the same `sol_interface!` block, subsequent interfaces will be expanded to include method definitions from the previous ones. For example:

```
 sol_interface! {
    interface IService {
        function makePayment(address user) payable returns (strin
        function getConstant() pure returns (bytes32);
    }
    interface ITree {}
}
```

The expanded Rust interface for `ITree` will include `make_payment` and `get_constant`, allowing calls to `ITree.make_payment`, for example, in the contract logic. This could allow malicious developers to hide the interface of `ITree`, including functions from previous interface definitions that could then be called on subsequent ones. Moreover, if `ITree` includes its own legitimate definition of `makePayment`, the code will fail to compile because the implementation block will include two function definitions of the same name.

Consider modifying the `sol_interface` macro by moving the `method_impls` declaration inside the first `for` loop so that it does not retain tokens from the previous methods during iteration.

Update: *Resolved at commit a64c058.*

Contracts Without at Least One Return Type Fail to Compile With `export-abi` Feature

The `export-abi` feature can be used in the Stylus SDK to generate a Solidity ABI for Stylus contracts using the `cargo stylus export-abi` command within the contract crate. With `fmt_abi` can be added to the returned `router` implementation. However, if the contract does not contain at least one function with an explicit return type (e.g., `U256`), the code will fail to compile due to an error in the `type_decls` token stream which is expanded in `fmt_abi`. This occurs when there are no return types because `types` is an empty array and `type_decls` attempts to loop over `[].iter()`. Since the compiler cannot infer the expected type from an empty iterator, attempting to access fields like `id` on an unknown type (`&_`) leads to errors.

Consider including `type_decls` within the `fmt_abi` generation process only when at least one type is available.

Update: *Resolved at commit 31995da.*

Unnecessary and Problematic Storage Types in Stylus

The Stylus language supports storage types such as `StorageU1` and `StorageI1`, which are not present in Solidity. These types do not provide any clear benefits within the SDK and fail to work properly with arrays and vectors. The `density` function, heavily utilized with arrays and vectors, triggers a division-by-zero error when interacting with these storage types. In addition, types like `StorageBlockHash` and `StorageBlockNumber` also lack clear utility.

Consider providing detailed explanations of the use cases and relevance of these storage types. If their benefits cannot be demonstrated, it is advisable to remove them to avoid confusion and potential interoperability errors.

Update: *Resolved at commit 3f511fa.*

Inefficient Storage of Strings and Bytes

Stylus allows for the use of both strings and dynamically-sized bytes in contract storage. However, the current implementation is notably inefficient for such types. Specifically, the bytes operate byte by byte, which results in a high number of `SLOAD` and `SSTORE` operations. This inefficiency is especially noticeable with longer strings or byte arrays, leading to significantly increased gas costs.

Consider refactoring storage operations for strings and dynamic bytes to optimize gas usage. Possible approaches include pre-allocating storage space when appropriate; writing data in full, 32-byte words where possible; and minimizing the number of storage operations.

Update: *Acknowledged, not resolved. The Offchain Labs team stated:*

> Work is ongoing for this issue but needs some additional testing.

Verification Challenges in Contracts May Facilitate Scams

The Stylus SDK allows developers to create smart contracts for Arbitrum chains using the Rust programming language which is then compiled to WASM and deployed alongside Solidity contracts. However, the final WASM output is influenced by several factors, including the Rust version, enabled features, dependencies, and more. These variations make the build process nearly non-deterministic across different operating systems and architectures. This non-determinism complicates contract verification, which is crucial for establishing trust and reliability in the contract. Without consistent build outputs, it becomes challenging to ensure that the deployed WASM accurately reflects the intended contract's code. A malicious actor could exploit this by altering the SDK to compile WASM files that do not function as expected, even if the smart contract code appears to be secure.

Consider standardizing the build process by specifying and enforcing clear guidelines and issuing notifications to developers early in the development cycle (instead of doing it post-deployment). This will streamline contract verification and enhance user trust in the contract's integrity.

repository.

Insufficient Test Coverage

The workspace currently contains only a limited number of unit tests for the `abi` module and the `mini-alloc` crate. The remainder of the codebase lacks unit tests entirely along with any integration tests. This limited test coverage may lead to undetected issues and hinder the verification of code functionality across various modules.

Consider adding a robust test suite that includes comprehensive unit and integration tests for all modules. This will help ensure proper interaction between different parts of the system.

Update: *Acknowledged, not resolved. The Offchain Labs team stated:*

> We are currently working on adding a full suite of unit tests for the SDK as well as other items outlined in issue #148.

Missing `receive` and `fallback` Functions

The absence of `receive` and `fallback` functions in Stylus, a language designed to be interoperable with Solidity, can have several significant implications. In Solidity, the `receive` function handles direct transfers of ETH to a contract. Without this function, contracts cannot accept plain ETH transfers, thereby limiting ETH transfers to those including data which triggers a specific function call. For example, contracts like `PaymentSplitter` will only work for externally owned accounts (EOAs) due to the lack of a `receive` function. In addition, many proxy patterns rely on the `fallback` function to forward calls to another contract. Without a `fallback` function, implementing upgradable contracts or beacon proxy patterns becomes much more complex, requiring alternative mechanisms to delegate calls to other contracts.

Consider implementing `receive` and `fallback` functions to ensure full compatibility with Solidity contracts. Doing this will help increase code flexibility and support different proxy patterns.

> Implementation underway. Needs further testing. Progress can be tracked on issue #150.

Solidity Interfaces in Stylus Might Mislead Users into Thinking They Match Solidity's Features

The `sol_interface` macro in Stylus smart contracts allows developers to nearly copy and paste Solidity interfaces for seamless contract interactions. However, the current implementation has some potential pitfalls. While it permits elements such as interface inheritance, events, and errors within the interfaces, these are silently ignored, with only the functions being processed. This behavior can lead to confusion and errors, as the contract compiles without issue despite these unsupported elements.

To address this issue, consider documenting the `sol_interface` macro's current limitations to set appropriate user expectations and implementing checks to revert when unsupported syntax is detected. If feasible, consider including these additional features in future updates as it could enhance the overall code functionality.

***Update:** Resolved at commits 821b7f6 and be6306c.*

Potential Misuse of Purity Attributes

In Stylus, contract methods can be marked with attributes such as `#[view]`, `#[write]`, and `#[pure]` to explicitly define how they interact with the contract state. However, there are two ways malicious users can mislead users or third-party services regarding these attributes:

- Malicious users can use colons in the attribute names, like `#[::pure]`, `#[stylus::view]`, or any other name with colons, to bypass the checks enforced by the `external` macro. This allows them to misrepresent the function's intention.
- Functions that do not modify the state, such as `#[pure]` and `#[view]` methods, can be incorrectly marked with the `#[write]` attribute without even requiring colons. The inline documentation of the macro contains this information, but it should be fixed.

enforce correct usage of these attributes.

Update: *Resolved at commit d44d94f. The Offchain Labs team removed the mutability specifiers (except `#[payable]`), as they can be inferred from the `&self`/`&mut self` or the absence of `self`. This change simplifies the code by leveraging Rust's syntax. Since methods using `&self` or those without `self` can still modify the state of other contracts—or even their own state if reentrancy is enabled—through external calls, the team has thoroughly documented this behavior in the codebase to inform users.*

Low Severity

Unclear Documentation Concerning `Call`

There are several instances in the documentation and code comments that are unclear or contradictory with respect to the difference between `Call::new` and `Call:new_in`. For example, `Call::new` is given as a simple example to show how to configure gas and value. However, `new` is only available with the reentrant flag enabled. The reasoning behind this is unclear since this comment says that `new_in` should be used for re-entrant calls. Other confusing comments in the documentation include:

> Note too that Call::new_in should be used instead of Call::new since the former provides access to storage. Code that previously compiled with reentrancy disabled may require modification in order to type-check. This is done to ensure storage changes are persisted and that the storage cache is properly managed before calls.

Consider clearly documenting the difference between `Call::new` and `Call::new_in`, and what storage access patterns they represent, and modify code comments accordingly.

Update: *Resolved at commit ba3472f.*

Unclear Usage and Documentation For Storage Context During Calls

`TopLevelStorage`. The usual pattern to create this implementation is to use the `entrypoint` macro, but this is not always desired if there are multiple contracts within a crate. Without `TopLevelStorage`, `&self` and `&mut self` are no longer available, and cumbersome workarounds are required, such as adding an empty implementation for `TopLevelStorage`. For example:

```
unsafe impl TopLevelStorage for Contract {}
```

Furthermore, it is unclear what the purpose of the `storage` argument is when using `Call::new_in` since the `call` function never actually uses this attribute of the call context.

Consider updating the code comments and documentation to clearly describe the purpose of the `storage` argument. In addition, consider alternative implementations of the `Call::new_in` function to accommodate contracts that do not implement the `TopLevelStorage` trait.

Update: *Resolved at commit ba3472f.*

Misleading Documentation

Throughout the codebase, multiple instances of inaccurate or misleading documentation were identified:

- The inline documentation for the `set` function in `StorageBlockNumber` is identical to that of the `get` method, causing confusion about their distinct functionalities.
- The documentation for the `raw_log` function advises users to prefer the alloy-typed `raw_log`, but it should actually recommend using the `log` function instead.

Consider correcting the documentation to align with the code's behavior. This will help improve the clarity and readability of the codebase.

Information Leakage in WASM Build

The WASM output of a Stylus contract includes sensitive metadata, such as the username of the individual who compiled the contract and partial paths from the home directory. Although this information does not pose a direct security risk, it can be leveraged by attackers for social engineering or other targeted attacks.

Consider removing or obfuscating such information from production builds to maintain privacy and reduce the potential attack surface. Stripping metadata from the WASM output can help mitigate these risks without impacting the functionality of the deployed contract.

***Update:** Resolved at commit 00abf34. The fix has been made on the `cargo-stylus` repository.*

Inefficient Allocator Fallback in Stylus Contracts

In Stylus contracts, the `mini-alloc` allocator module from the SDK is intended to be the standard allocator due to its performance. This module is included in every example and default template provided by the SDK, ensuring that developers are guided towards using the most efficient option. However, the declaration of the global allocator, which specifies the use of `mini-alloc`, can be removed inadvertently. If this happens, the contract defaults to using the allocator from the standard library, which is significantly less efficient. Consequently, contracts deployed using this allocator will be very gas-inefficient.

Consider enforcing `mini-alloc` as the default allocator across all Stylus contracts. Changes to the allocator should only be permitted through a deliberate and evident action to avoid unintentional fallback to the less-efficient standard library allocator.

***Update:** Resolved at commit 2354799.*

Macro Implementations Missing Proper Docstrings

missing proper docstrings.

Given the complexity and length of the code, consider adding detailed docstrings. This will make it easier for readers and developers to better understand the inner workings of the codebase while improving the overall code maintainability as well.

Update: *Acknowledged, not resolved. The Offchain Labs team stated:*

> We are looking to refactor our procedural macro implementations to make them more easily testable and easier to understand. This will include better internal documentation of their implementations. Progress may be tracked on issue #151

Misleading Methods in `RawDeploy`

The `limit_revert_data` and `skip_revert_data` methods set the `offset` and `size` fields of the `RawDeploy` instance. However, these fields are never used in the `deploy` function, rendering these methods redundant. This issue is significant because it misleads developers into believing that they can control the amount of revert data returned, whereas, in reality, these methods have no impact on the deployment's outcome.

Consider removing these methods and fields or modifying the `deploy` function to utilize these fields.

Update: *Resolved at commit 6e21166.*

Potential Misuse of `#[borrow]` Attribute in Storage Fields

The `#[borrow]` attribute is used on storage fields to implement the `Borrow` and `BorrowMut` traits for specific types, facilitating inheritance in Stylus contracts. However, its application can extend to state variables that do not represent the storage of the parent contract, leading to issues.

types. This attribute is designed for complex types that represent a subset of the contract's storage, not for individual storage slots. Such usage misrepresents the attribute's purpose, misleading developers about the storage layout or contract composition. Furthermore, it generates additional trait implementations that are unnecessary for simple storage types, causing unnecessary code bloat and potentially increasing the contract size without any benefit.

Consider allowing the use of the `#[borrow]` attribute exclusively for fields that genuinely represent a subset of a contract's storage. This ensures accurate semantic representation while avoiding misleading code and unnecessary trait implementations.

Update: *Acknowledged, not resolved. The Offchain Labs team stated:*

> Work is ongoing on this issue, and we are searching for a solution that works in Rust. Progress may be tracked on issue #149

Deprecate `constant` State Mutability in `sol_interface` Macro

The `sol_interface` macro currently allows users to define Solidity interfaces with methods marked as `constant` for state mutability. However, starting from Solidity version 0.5.0, the `constant` keyword for functions is no longer supported and has been replaced by `view` and `pure`. While the macro internally converts functions with the `constant` keyword to `pure`, retaining the `constant` keyword can be misleading.

To align with the later Solidity versions and avoid incorrect mutability assumptions, consider updating the `sol_interface` macro to disallow the use of the `constant` keyword for function state mutability. This change will help ensure compatibility with modern Solidity versions and enhance code correctness.

Update: *Resolved at commit e173182.*

`sol_interface` Improper Handling of Function Visibility

always be `external`. The `sol_interface` macro in Stylus, which processes these Solidity interfaces, currently allows users to include methods with incorrect visibility attributes, such as `public`, or omit the visibility attribute altogether. This non-compliance with Solidity standards could lead to confusion among developers.

Consider adding validation to ensure that the `external` keyword is present in function definitions. This would align with Solidity's interface requirements and prevent potential misunderstanding.

Update: *Resolved at commit 2330ac7.*

`sol_interface` Lacks Support for Struct and Enum Types

The `sol_interface` macro is designed to enable developers to seamlessly call Solidity contracts from Stylus smart contracts using their native interfaces. However, it currently does not support struct and enum types, which are commonly used in Solidity. This limitation forces developers to use less readable workarounds, potentially leading to accidental errors and reduced code maintainability.

Consider implementing support for structs and enums. This would significantly enhance developer experience and ensure full compatibility with Solidity interfaces.

Update: *Acknowledged, not resolved. The Offchain Labs team stated:*

> Work has begun on struct support in `sol_interface!`. More testing is required for release, and enums should be implemented as well. Progress may be tracked on issue #74.

`sol_storage!` Macro Does Not Support Private State Variables

Private state variables help enforce encapsulation by restricting direct access, ensuring that variables are only modified through controlled functions. This approach minimizes the attack surface and prevents unintended side effects or inconsistencies.

allowing child contracts to access and modify these variables directly. Developers must instead use the `#[solidity_storage]` macro, which supports private state variables.

To maintain consistency with `#[solidity_storage]`, consider enhancing the `sol_storage!` macro to support private state variables. Alternatively, this limitation should be clearly documented in the official documentation.

***Update:** Acknowledged, not resolved. The Offchain Labs team stated:*

> We will consider private state variables for the `sol_storage!` macro as part of the work for N-04, tracked by issue #147.

Notes & Additional Information

Naming Issues

Throughout the codebase, multiple instances of elements that could be renamed to better reflect their purpose were identified:

- The `topics` parameter in the `emit_log` function should be renamed to `number_topics` to better reflect the nature of the value it represents.
- The `#[external]` macro, which allows methods to be callable by other methods within the contract or external accounts, should be renamed to `#[public]`. The current name might be confused with Solidity's `external` visibility, which implies that the function cannot be called from within the contract.
- The `#[solidity_storage]` macro should be renamed to `#[persistent_storage]`, `#[storage]`, or `#[state]`. Instead, the wrapper macro currently named `sol_storage` should adopt the name `solidity_storage`, as it relates more to Solidity's syntax.
- The `types` module name suggests the existence of multiple types. However, if only `Address` is defined, the module should be renamed to `AddressType`. Alternatively, if

conclusion.

Consider addressing these naming issues to improve the readability of the codebase.

Update: *Resolved at commit 8d1699f. The Offchain Labs team decided to keep both the `sol_storage!` macro and the `types` module with the same name. The former aligns with the `sol!` macro from alloy, while the latter will include additional types in an upcoming release.*

`wee_alloc` Crate is Unmaintained and Vulnerable

The `wee_alloc` crate, a minimal allocator for WebAssembly, is no longer actively maintained, with the last release being over three years ago. Moreover, two of its maintainers have indicated that they do not plan to continue supporting the crate. As a result, several open issues, including memory leaks, remain unresolved.

Even though the crate is not currently used for `wasm32` targets, consider switching to a more actively maintained and safer alternative such as `lol_alloc` or the default Rust standard allocator.

Update: *Resolved at commit 80bfcba.*

Unstable License URL Reference

The license URL comment at the top of nearly every file in the scope points to a specific branch (`stylus`) in the GitHub repository. This branch is not the main branch and may change or be deleted in the future. If the branch name changes or the license is relocated, the current URL references will become invalid, leading to broken links.

Consider updating the license URL to point to a more stable reference, such as a specific commit or tag. Alternatively, consider including a note that the branch name may change. These measures will help ensure that the URL remains functional over time.

Limited Functionality in `sol_storage` Macro

The `sol_storage` macro currently allows users to define state variables in a Solidity-like syntax within their smart contracts. However, it does not fully replicate Solidity's syntax, notably lacking support for specifying visibility, as well as defining constants and immutables. This limitation prevents things like the automatic generation of getters using the `public` keyword and makes it difficult to define constants and immutables as seamlessly as in Solidity.

Consider extending the functionality of the `sol_storage` macro to support these features. Utilizing the `syn_solidity` crate could facilitate this enhancement by providing a more comprehensive parsing and handling of Solidity-like syntax.

Update: *Acknowledged, not resolved. The Offchain Labs team stated:*

> We are considering these additional features for the `sol_storage!` macro. Progress can be tracked in issue #147.

Lack of Length Accessor for Fixed-Size Arrays

In Solidity, both fixed-size and dynamic arrays support the `.length` property, allowing developers to easily determine the size of an array. However, in Stylus, there is no built-in method to access the length of fixed-size arrays. This functionality is available only for dynamic arrays (referred to as vectors in Stylus). This absence of a length accessor for fixed-size arrays may lead to inconsistencies and additional complexity in array management.

Consider implementing a built-in method to access the length of fixed-size arrays in Stylus, similar to the `.length` property in Solidity. This enhancement would simplify array handling and reduce the risk of errors.

Update: *Resolved at commit 8ab7650.*

Unresolved Link to `EagerStorage`

`EagerStorage` exists in the `storage` module.

Consider updating or removing the broken link to ensure accurate documentation and avoid confusion for developers.

Update: *Resolved at commit 9b221c8.*

Typographical Errors

Throughout the codebase, multiple instances of typographical errors were identified:

- `inheritence` should be `inheritance`.
- `occured` should be `occurred`.

Consider fixing the aforementioned typographical errors in order to improve the readability of the codebase.

Update: *Resolved at commit 5052d30.*

External Macro Attribute Handling Inconsistency

The external macro currently accepts attributes that are not utilized in its implementation. This can lead to confusion, especially in comparison to the entrypoint macro which throws an error if it receives any attributes.

To ensure consistency and reduce potential confusion, consider adding validation to the external macro implementation to reject any attributes.

Update: *Resolved at commit a1267bf.*

Outdated Copyright Year

Outdated copyright years may not reflect recent modifications or ongoing development. Several files within the codebase have outdated copyright years, including the license file. Other

- `tx.rs`
- `lib.rs`
- `block.rs`

Consider updating all outdated copyrights to signal active maintenance and attention to detail.

Update: *Resolved at commit d57458d.*

Todo Comments in the Code

During development, having well-described TODO comments will make the process of tracking and solving them easier. Without this information, these comments might age and important information for the security of the system might be forgotten by the time it is released to production. These comments should be tracked in the project's issue backlog and resolved before the system is deployed.

Throughout the codebase, multiple instances of TODO comments were identified:

- Line 31 and Line 270 of `external.rs`
- Line 279 of `proc.rs`

Consider removing all instances of TODO comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO comment to the corresponding issues backlog entry.

Update: *Resolved at commit 73dbc1c.*

Conclusion

The Stylus SDK allows smart contract developers to build applications for the Arbitrum ecosystem using the Rust language. These Stylus programs are compiled to WebAssembly innovative approach aims to combine the efficiency of WASM execution with the robustness of programming in Rust, all the while maintaining compatibility with the Ethereum Virtual Machine.

During the security audit of the Stylus SDK, we discovered numerous security issues and also made extensive recommendations for the improvement of the overall design. The project is clearly still under development, having several features that are either non-functional or contain bugs. However, the development team showed a strong commitment to addressing these concerns and we encourage them to continue their efforts. Once all the identified issues are resolved, further improvements are made, and the project reaches a more mature stage, we strongly recommend the team to consider conducting a follow-up audit to ensure comprehensive security.

Despite the current challenges, we see great potential in the Stylus SDK. We look forward to seeing how the project evolves, particularly as the team addresses the identified issues and continues to refine the SDK. We believe that further development of the Stylus SDK could introduce exciting new possibilities to the Arbitrum ecosystem and the broader world of smart contract development.

Request Audit

OpenZeppelin

Related Posts

penZeppelin × ARBITR FOUNDATION
Next-Gen Development on Arbitrum Stylus

Next-Gen Development on Arbitrum Stylus

OpenZeppelin and Arbitrum are joining forces in a groundbreaking partnership to introduce a new era...

Announcements L2's Rust

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits Rust Ink! Ecosystem Library

The Parity Wallet Hack Reloaded

Today we witnessed a possible major attack on the Parity MultiSig contract. This follows a previous...

Security Audits Rust

Products

Services

Resources

Company