OpenZeppelin

Security Review ink! & cargo-contract

OPENZEPPELIN SECURITY | APRIL 6, 2023

Security Audits

March 31, 2023

This security assessment was prepared by **OpenZeppelin**.

Table of Contents

Review Summary

The security review of ink! and cargo-contract has been deemed successful, and no critical issues were detected. However, two high-severity issues were discovered, which the Parity team is already addressing. Furthermore, OpenZeppelin has suggested some modifications and improvements to follow best practices, minimize the potential attack surface, and enhance both the language and the tool.

Description

The Parity team asked us to conduct a security review of version 4 of their Rust eDSL for writing smart contracts, *ink!*. This engagement was funded by the Polkadot treasury. To ensure a comprehensive and structured evaluation, we have divided the engagement into three distinct phases. The first phase was to gain an understanding of ink! and its ecosystem, the second to carry out the security review, and the third to work with the Parity team to address the issues found in the previous phase. We utilized the knowledge gained from ink! and the cargo contract to perform a security analysis. Our objective was to verify the efficacy of ink! in mitigating known security risks. The primary goal of this security overview report is to share our identified security vulnerabilities, provide recommendations to fix them, and share our experience and insights gained from maintaining OpenZeppelin contracts and building developer tools.

OpenZeppelin sees this engagement as an opportunity to help the Parity team improve the overall experience of using *ink!* for developers and auditors in the future. The upcoming deliverables will include reviewing all items mentioned in the Scope section below.

Scope

OpenZeppelin performed a security analysis of the *ink!* repository, the `cargo-contract` CLI, and a features comparison against Solidity. The primary focus was to identify potential vulnerabilities and provide recommendations for improvement. Specifically, the analysis encompasses the following areas:

- A review of the *ink!* repository, with particular attention given to the env, storage, and ink crates. This action will involve analyzing all macros to ensure they are secure and evaluating

implementations of the features, misconfigurations, and other issues that may impact the tool's security.
- A features comparison between Solidity and ink!, adding recommendations to improve the latter. This will include identifying any key features or functionality that ink! lacks compared to Solidity and suggesting ways to enhance the ink! ecosystem.
- A review of the documentation, tutorials, and tools, adding recommendations to improve the overall user experience when using ink! and its periphery.

The commits and repositories in scope for the review are:

- paritytech/ink, at commit c8aa3ee41112b327d4f3cb3959f188945c8ccace
- paritytech/cargo-contract at commit 60bc0f0402b0c74873f849fef1bc45e326d67191
- paritytech/ink-docs at commit 7a62015b4ea9c020a175404017bb5492beb24328

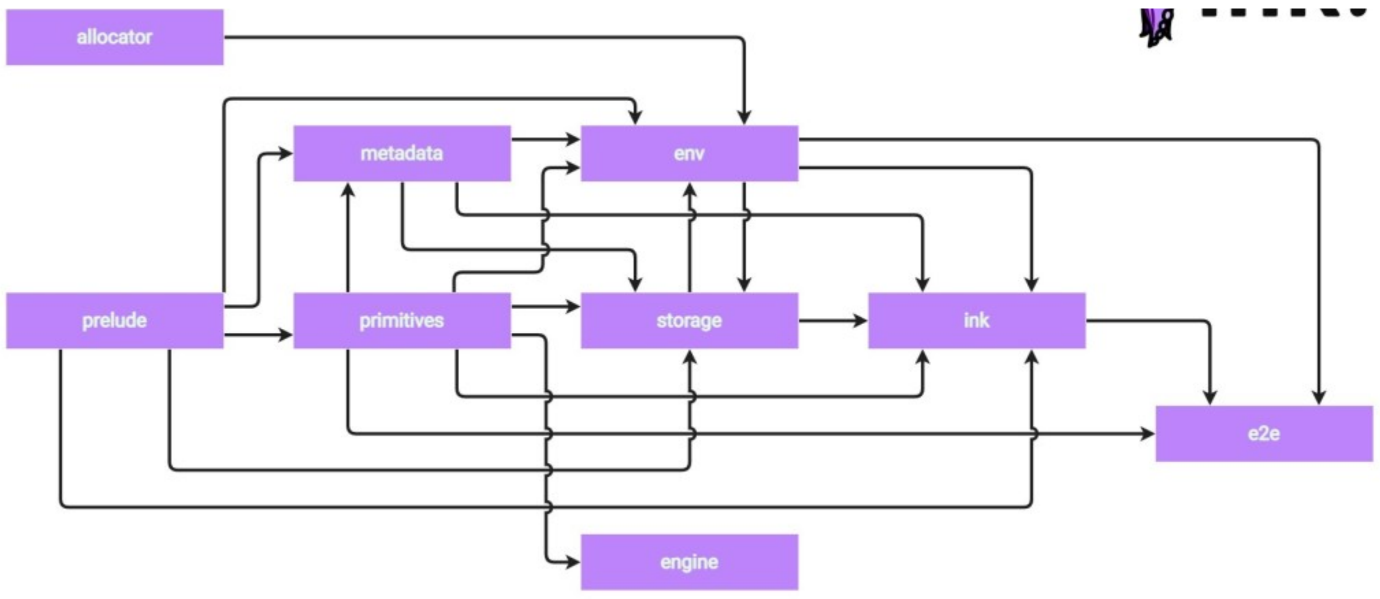
Disclaimer: In light of the limited duration of this service and the substantial size of the codebases, it is essential to note that this review was not exhaustive. While OpenZeppelin made its best efforts to share any situations that may constitute issues or areas of improvement, we cannot guarantee that all such issues have been detected.

System Overview

ink! is a domain-specific language (DSL) based on Rust for writing smart contracts to be executed on blockchains built with the Substrate framework. The main objective of designing this language was to make it as similar as possible to writing regular Rust code while still being safe and efficient. The choice of Rust as the base language for *ink!* provides multiple benefits such as type safety, memory safety, and small binary size, making it an ideal choice for developers.

To deploy smart contracts written with **ink!**, the target blockchain must have the `pallet-contracts`, a module that supports the execution of contracts. *ink!* consists of 9 crates: **allocator**, **e2e**, **engine**, **env**, **ink**, **metadata**, **prelude**, **primitives**, and **storage**.

Architecture



- **ink**: Contains the procedural macros to generate the final code that the compiler will convert to Wasm. This code is the one that runs on the blockchain and performs the specific actions of the contract.
- **env**: Provides the connection to the pallet-contracts, allowing for interactions with the underlying execution engine of the smart contract. These interactions include reading and writing to a smart contract's storage and access to environmental functions such as information about the caller of a contract call and self-terminating the contract.
- **allocator**: Used for dynamic memory allocation in smart contracts during execution.
- **engine**: An off-chain testing engine that simulates a blockchain environment and allows mocking specified conditions.
- **e2e**: Package for end-to-end testing of the contract.
- **metadata**: Describes the contract in a platform-agnostic way, its interface, types, storage layout, etc.
- **prelude**: Provides an interface to standard library types and functionality since contracts are run in a no_std environment.
- **primitives**: Collection of utilities used internally by multiple ink! Modules.
- **storage**: Provides collections for developers to use in contract storage.

Cargo Contract

Command-line interface (CLI) for creating, compiling, testing, uploading, instantiating, and decoding *ink!* contracts, making the development process more efficient. The app consists of 4

- **Cargo-contract**: CLI implementation developed with Clap.
- **Metadata**: Defines types for the extended metadata of smart contracts targeting Substrate.
- **Transcode**: Contains utilities for encoding contract calls to SCALE.

Threat Model

Actors, Assets, External Dependencies and Entry Points

This section defines actors, assets, external Dependencies and entry points for this threat model.

Actors

Users or smart contracts that make use of ink! and cargo-contract. This allows us to define the access rights or privileges required at each entry point, and those required to interact with each asset. Considering which actors interact with the ink! is helpful to determine how the language and the smart contracts made with it can be compromised.

ID	Name	Description
1	Regular User	User that interacts with the smart contract through non-restricted messages and access events.
2	Smart Contract Developer	Developers that create and maintain ink! smart contracts and use cargo-contract CLI.
3	Smart Contract	An ink! smart contract.
4	Privileged User	Authenticated user that could interact with a smart contract through its restricted messages.
5	Core Developer	Developers that maintain the codebase repository for ink! and cargo-contract.

6	Contributor	Non-official team member who contributes to the development of ink! or cargo-contract.

Entry Points

Specify how malicious users can access and interact with ink!, its command-line interface tool, and the smart contracts built using it.

ID	Name	Description	Actors
1	Constructors	These entry points are called when the smart contract is instantiated. It initializes the contract's state variables and executes some logic if needed.	(2) Smart Contract Developer, (3) Smart Contract, (4) Privileged User
2	Non-restricted Messages	Public functions with the message attribute are the only way for non-priviledged users to interact with a smart contract. This method defines the behavior of the contract in response to specific parameters.	(1) Regular User, (2) Smart Contract Developer, (3) Smart Contract, (4) Privileged User

3	Restricted Messages	Functions that modify sensitive parts of the contract’s storage and can only be called by privileged `AccountID`s set in the contract storage.	(4) Privileged User
4	Contract Metadata	It provides information about the contract’s name, version, methods, storage, and general data.	(2) Smart Contract Developer
5	Cargo-contract CLI	The command-line interface (CLI) used to manage the ink! smart contract, including compilation, testing, deployment, and interaction with deployed smart contracts.	(2) Smart Contract Developer, (3) Smart Contract
6	UI of the Dapp	The user interface (UI) used to interact with ink! Dapps.	(1) Regular User, (2) Smart Contract Developer, (4) Privileged User

7	GitHub repository	The code repository used to store and manage the ink! smart contract source code. This entry point is a common entry point for malicious developers.	(2) Smart Contract Developer, (5) Core developers

Assets

Refer to the elements in ink! and the contracts developed with it that could represent value to an attacker, such as tokens, balances, credentials, and abstract assets. These assets require protection against potential attackers.

ID	Name	Description	Actors
1	ink Language	The language implementation and its specification that defines the behavior of ink! smart contracts. The correctness and security of the ink! language are essential to prevent potential attacks.	(5) Core Developer, (2) Smart Contract Developer, (6) Contributor
2	ink dependency	The crates that define the ink! language and its dependencies. The ownership and control of these crates are critical to the stability and security of ink!.	(5) Core Developer, (6) Contributor
3	Open Contract storage	The smart contract's storage comprises various user data such as balances and allowances. Users can grant privileges to other users or smart contracts over their assets.	(4) Privileged User, (3) Smart Contract, (1) Regular user

4	Restricted Contract storage	The smart contract's storage may include user permissions, owners, and other contract data.	(2) Smart Contract Developer, (3) Smart Contract, (4) Privileged User
5	User keys	The private keys of users which are used to sign transactions and interact with the contract.	(1) Regular user, (4) Privileged User
6	Modify Codebase repository	The integrity of the repositories where ink! and cargo-contract are hosted should only be only modified by their core devs. This includes version control and code reviews.	(5) Core Developer
7	Chain native token	The underlying token used for paying gas.	(1) Regular user, (3) Smart Contract
8	Project reputation	The reputation is key to the success of ink! as the go-to language for building smart contracts.	(1) Core Developers

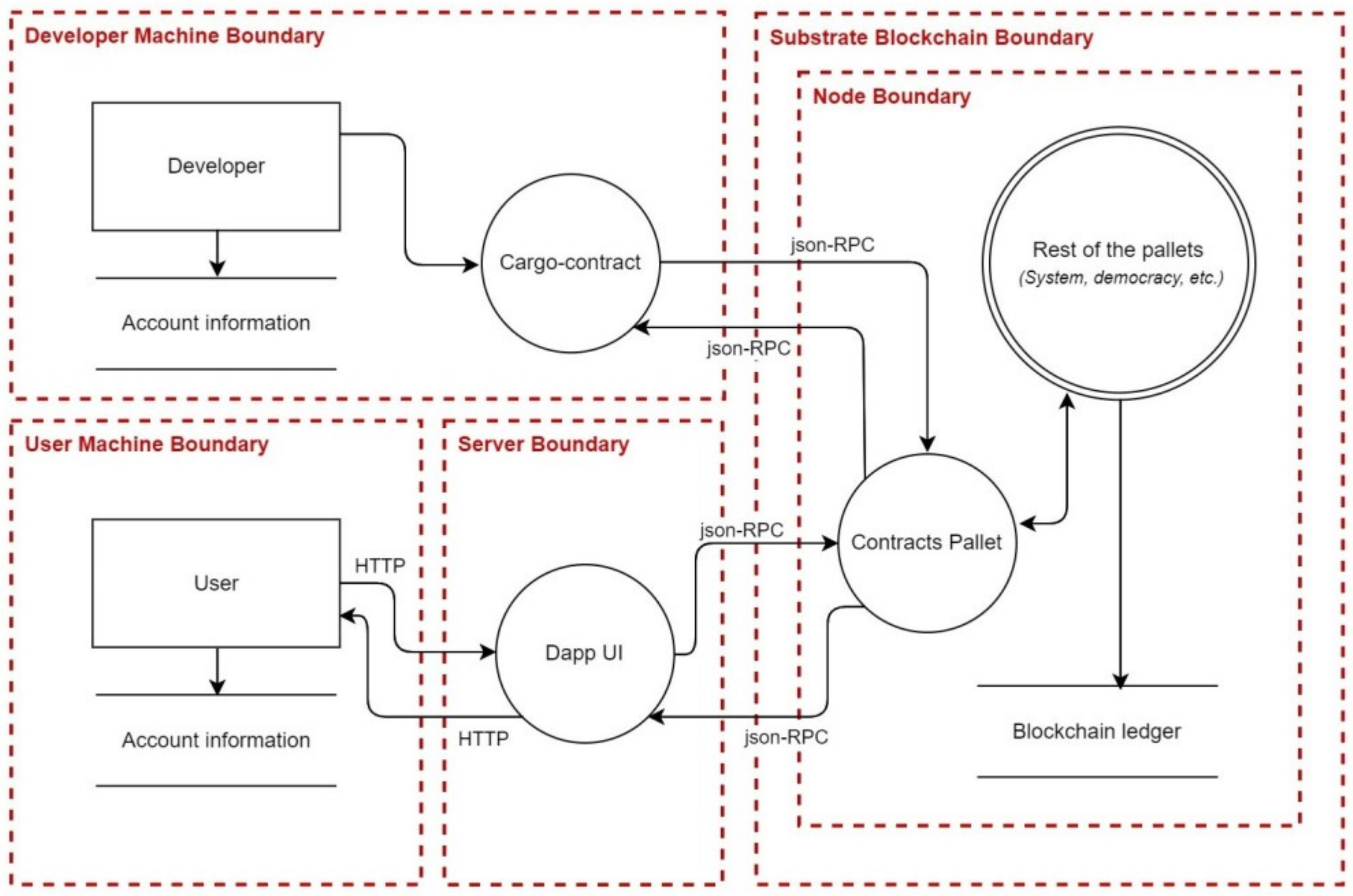
External Dependencies

These are the main external elements that may pose a threat to the language and the contracts made with it.

ID	Description
1	Rust language and its package manager (Cargo), which are used to write and manage dependencies of ink! and cargo-contract.
2	Substrate runtime modules, which provide blockchain-specific functionality and interfaces for smart contracts written in ink!. The most relevant in this case is the contract-pallet, runtime to deploy and execute WebAssembly smart contracts.
3	Substrate client libraries, which are used to interact with Substrate blockchain nodes.

4	Blake2 hashing implementation.
5	GitHub, where the official codebase for ink! and cargo-contract is hosted.
6	Rust WebAssembly target, which is used to compile ink! smart contracts into WebAssembly.
7	Third-party Rust crates, which provide additional functionality for ink! and cargo-contract.
8	Substrate blockchains, on which ink! smart contracts will be deployed and executed.

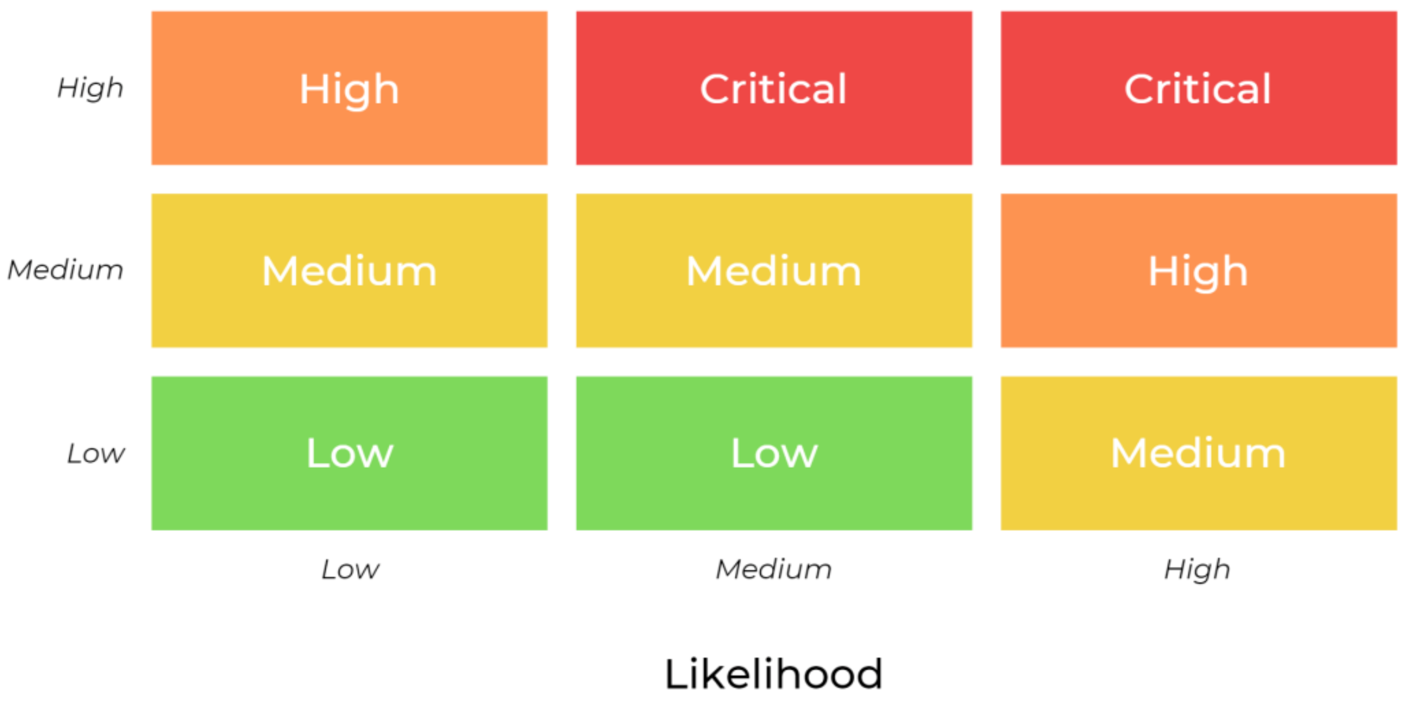
Simplified Data Flow



Risk Assessment Methodology

As a means of standardization, OpenZeppelin has implemented a matrix for classifying issues that adheres to the OWASP risk rating methodology. The method, widely used in the cybersecurity technologies. We have deemed it necessary to place greater emphasis on the impact axis in our matrix. The presence of financial assets within the ecosystem and its decentralized nature amplify the potential for damage. Furthermore, the possibility of mitigating is usually low as the damage caused may be irreversible due to the immutable nature of these technologies.

Severity Classification



High	High	Critical	Critical
Medium	Medium	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

Terminology:

- *Likelihood*: How likely it is that an attacker will find and exploit the vulnerability.
- *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 Levels Descriptions

have a catastrophic impact on the client's reputation or serious financial implications for the client and users. It often involves, but is not limited to, some form of loss or locking of funds or other core system functionality failures.

High

The issue puts a large number of users' sensitive information at risk, and/or is reasonably likely to have a high impact on the client's reputation or notable financial implications for the client and users. It often involves, but is not limited to, some form of temporary loss or locking of funds or other core system functionality failures for which reasonable mitigations may be available.

Medium

The issue puts a subset of users' sensitive information at risk, which would be detrimental to the client's reputation if exploited, or is reasonably likely to have a moderate financial impact on the client and users.

Low

The issue is relatively small and could not be exploited regularly or is a risk that the client has classified as low impact given their business model.

Notes and Additional Information

The issue is not security-relevant but still worth noting to increase the quality of the code base.

Threat Categorization

To thoroughly assess the potential vulnerabilities in *ink!*, we used the DASP framework, a widely recognized open project that identifies the top 10 categories of smart contract vulnerabilities, and the SWC registry, a classification system outlined in `EIP-1470`. This approach allowed us to identify the components of *ink!* that are most at risk based on common Solidity vulnerabilities. We can identify the following threats and policies using applicable parts of Dasp 10 and OpenZeppelin's internal knowledge:

- Access Control: Threat actors may bypass access control rules by invoking restricted functions.
- Arithmetic Issues: ink! must be able to handle overflow and underflow conditions in arithmetic operations.
- *Bad Randomness*: If the ink! contract relies on random numbers, the underlying random number generation must be secure and free from predictability, as predictable or manipulated randomness can be exploited by attackers.
- *Collisions of Storage Layouts*: To prevent unintended data modifications or loss in upgradable contracts, ink! must minimize the likelihood of storage layout collisions that can be exploited by attackers.
- *Proxy Selector Clashing*: If different contracts within a system use the same function signature as an already deployed function in a proxy contract, it can result in proxy selector clashing, causing transactions to fail and leading to potential loss of funds. This can also allow attackers to manipulate the contract behavior and bypass security controls
- *Denial-of-service*: Attacks can be launched by an attacker targeting a smart contract to cause a service disruption by inducing unexpected reverts, consuming excessive memory or gas, or filling multiple blocks in the blockchain, which can prevent other transactions from being included in any of the blocks.
- *External Data Source Dependency*: ink! contracts may depend on external data sources, such as APIs or oracles, which can introduce vulnerabilities if the data sources are not secure or can be manipulated by attackers. This can result in malicious actors exploiting these vulnerabilities to gain control over the contract, steal funds, or manipulate contract behavior.
- *Dependency Hijacking*: Malicious actors may inject malicious code into one of the dependencies of the ink! or cargo-contract repositories. As a result, the attacker can gain control of the entire system and bypass the security controls. Therefore, it is critical to ensure that all dependencies used by ink! and cargo-contract are verified and come from trusted sources.
- *Phishing*: Attackers may use phishing tactics to trick users into revealing sensitive information, such as private keys or passwords, which can then be used to compromise the security of ink! contracts and steal funds.
- *Infrastructure Compromise*: The infrastructure used to run ink! contracts, such as the blockchain network, nodes, or hosting servers, can be compromised by attackers. This issue

transactions, loss of funds, and disruption of contract execution.

As the DASP and SWC registry may not be up-to-date, we also leveraged OpenZeppelin's internal knowledge to enhance our spectrum of the potential vulnerabilities that could apply within the *ink!* context.

Our focus was on potential attackers who may take advantage of their knowledge of common vulnerabilities within EVM blockchains and attempt to exploit them in the context of *ink!*.

To ensure a comprehensive and structured review, we followed the SCSVS checklist (Smart Contract Security Verification Standard). This approach could help prevent potential vulnerabilities from being overlooked during the assessment of *ink!*.

Countermeasures and Mitigation

This section lists the protective measures that could prevent a threat from being realized. If a threat has no countermeasure, it is vulnerability.

- **Reentrancy**:
  - ink! requires that developers explicitly use `set_allow_reentry` flag to allow the callee to reenter into the current contract.
  - Enforce the use of the `Checks-Effects-Interactions` pattern to ensure that all state changes are made before any external calls are made.
  - Use Reentrancy Guards like OpenBrush implementation to prevent reentrant attacks.
- **Access Control**:
  - Use the Principle of Least Privilege to ensure that only authorized users have access to restricted functions.
  - Implement access control mechanisms such as role-based access control (RBAC) or attribute-based access control (ABAC).
  - ink! devs can leverage third-party access control contracts to define and enforce access control rules.
- **Arithmetic Issues**:
  - Rust can detect at compiled time underflows and overflows and reverts the build process.
  - Developers can implement input validations to prevent invalid or malicious inputs.

- Devs can use gas-efficient coding techniques to reduce the cost of executing transactions.
- Implement rate limiting to prevent excessive usage and protect the contract from abuse.

- **External Data Source Dependency**:
    - Use trusted sources for external data and oracles, and avoid relying on a single source for critical decisions.
    - Implement input validations to prevent invalid or malicious inputs from external data sources.
    - Use a fallback mechanism in case an external data source fails or is compromised.
- **Dependency Hijacking**:
    - Verify all dependencies and ensure that they come from trusted sources.
    - Use a dependency management tool to track and manage all dependencies in the project.
    - Cargo has a `yank` feature that allows package maintainers to remove a pushed crate from the index. This can be useful if a version is found to have a security vulnerability or other issue.
    - Regularly review the dependencies in the project and check for any known vulnerabilities or suspicious activity. RustSec can help in this process. Update all dependencies to the latest versions that address known vulnerabilities. Parity uses Dependabot as one of the CI workflows.

Suggestions

The following are general suggestions of potential features that the Parity team could implement to improve the overall user and developer experience using ink! and `cargo-contract`, while facilitating a smooth onboarding for Solidity developers.

Create new initiatives for smart contract monitoring

As smart contracts become more prevalent, it is becoming increasingly important to have better tools and processes for monitoring and responding to incidents. Creating new initiatives for smart contract monitoring that are more aligned with incident response, such as Forta or OpenZeppelin

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

> *Pending talks with the Sirato Substrate Explorer.*

Detect `set_code_hash` use

One way to implement contract upgradability is to use the set_code_hash function, which allows for the contract code to be updated without modifying the contract's address. To detect if a contract has this capability, block explorers or third-party applications can inspect the contract's Wat and look for the set_code_hash import. If present, this indicates that the contract can upgrade its code.

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

> *Pending talks with the Sirato Substrate Explorer.*

Implement Solidity immutable-like variables

Immutable variables in Solidity allow developers to create read-only variables that cannot be modified at runtime, which can be useful for security and efficiency purposes. Implementing similar functionality in ink! would give developers more flexibility in designing and building secure and efficient smart contracts.

Update: *Recommendation acknowledged in issue 1714 of the "ink" repository.*

Allow default implementation in trait methods.

Adding the ability to have default implementations in trait methods, similar to abstract contracts in the Ethereum ecosystem, would make it easier for developers to write reusable and composable contracts.

Update: *Recommendation acknowledged in issue 1689 of the "ink" repository.*

Add support for Solidity-like libraries.

Adding support for Solidity-like libraries in ink! would allow developers to create and reuse common functions across multiple contracts, improving code reusability and reducing duplication.

Update: *Recommendation acknowledged in issue 1684 of the “ink” repository.*

`cargo-contract` Feature Suggestions

The following are some ideas that the OpenZeppelin team has come up with over the engagement after exhaustively using the `cargo-contract` tool. The viability of developing some of these suggestions depends on how flexible the rust libraries used to build `cargo-contract` are.

Add a command to generate interfaces for Solang-compiled contracts

```
cargo contract generate-interface $language
```

Creating a command on `cargo-contract` that generates interfaces for a target programming language would make it easier for developers to integrate their smart contracts with other contracts written in a different programming language. This command would facilitate cross-contract communication and help create more complex and sophisticated contract applications.

Initially, the feature should support ink! and Solidity, as those are the most common options for the `pallet-contracts`. Its design should be flexible enough to add other languages that may become popular.

Update: *Recommendation acknowledged in issue 807 of the “cargo-contract” repository.*

Improve the `build` command for multi-contract projects

The current build process for multi-contract projects requires building each contract individually, which can be a time-consuming and tedious process. To improve this process and make it more convenient for developers, it is suggested to include the ability to build multiple contracts within a single project directly in the cargo contract build command.

This feature would allow developers to build all contracts within a project with a single command, reducing the time and effort required for the build process. It would also make it easier for developers to manage and maintain their projects, as they would not have to worry about managing multiple build commands for each contract.

```
cargo contract encode [options] --message <MESSAGE> --args <ARGS..
```

There is currently no encoding command in the cargo contract. An `encode` function would allow developers to encode the arguments of a function call, making it easier to test different scenarios and use cases and debug the code.

***Update:** Addressed in pull request #998 of the "cargo-contract" repository.*

Allow raw RPC calls

```
cargo contract rpc [options] METHOD [PARAMS...]
```

Users cannot make raw RPC calls to the node with the currently available commands.

Adding this capability would improve the debugging process by giving developers direct access to the node's RPC interface, making it easier to diagnose, fix, and experiment with different scenarios.

***Update:** Recommendation acknowledged in issue 987 of the "cargo-contract" repository.*

Implement fork capabilities in `cargo-contract`

To facilitate testing and development, it would be convenient to have the ability to implement fork capabilities in the `cargo-contract`. This feature would allow developers to test their smart contracts using the current state of a target blockchain, which can help identify and fix potential issues before deploying the contract to a live network.

***Update:** Recommendation acknowledged in issue 988 of the "cargo-contract" repository.*

Add `cargo-contract accounts` command

```
cargo contract accounts
```

__Update:__ Acknowledged, not resolved yet.

Add `cargo contract set-default-account $account` command

```
cargo contract set-default-account $account
```

Example:

```
$ cargo contract set-default-account //Alice
```

This action should set the $account sent as a parameter as the default account to be used in all the subsequent commands. In most cases, when interacting with one or more contracts through a CLI, it is common to always use the same account, instead of multiple accounts. Adding the ability to set a default account allows developers to avoid using the `--suri` flag every time they instantiate or interact with a contract. If the `--suri` flag is defined when using a command, the default account for that particular use should be overwritten by the one specified.

__Update:__ Acknowledged, not resolved yet.

Add `cargo contract deployed-contracts` command

```
$ cargo contract deployed-contracts
```

This command should let the developer see a list of all the contracts that have been deployed so far. It should show the name of the contract (which could be taken from the metadata), the address (or addresses) where it is deployed, and a list of all the accessible methods and their parameters (i.e., their interface). Most of these capabilities are already available in the contracts UI tool, but, based on our experience, developers see more value in using CLI tools when developing smart contracts, and being able to see all contracts deployed and their interfaces.

__Update:__ Acknowledged, not resolved yet.

receive a contract `$contract` as a parameter and show the address (or addresses) where it is deployed, and the list of all accessible methods and their parameters.

***Update:** Recommendation addressed in issue 783 of the "cargo-contract" repository.*

Split the `cargo contract call` command into `cargo contract call` and `cargo contract send`

```
$ cargo contract call --contract $contract --message $message --su
```

In Ethereum and other ecosystems, one of the differences between sending a transaction (i.e., calling a public or external non-view/non-pure function) and calling a view/pure function is very clear: the former consumes gas and the latter does not. In ink!, even though every call consumes gas and generates a transaction id (since there is no such thing as new and pure functions), it is possible to "call" a function in a contract without spending gas, by performing a dry run call.

Consider renaming the `call` contract to `send`, removing (or hiding) the `--dry-run` flag, and consider implementing a `$ cargo contract call` command that implements the `--dry-run` flag functionality under the hood.

***Update:** Addressed in pull request #999 at commit 852e5b4*

Implement an `upgrade` command

```
$ cargo contract upgrade $proxy-address $implementation-address --
```

Upgradeability is a well-known feature in the blockchain ecosystem: it allows a contract implementation to be modified, fixed, and improved without the need of deploying the new contract in a new address. As the Parity team is aware, using a proxy pattern to implement upgradeability has its pros and cons (some are mentioned below).

This command should allow developers to upgrade a given proxy `$proxy-address` to a new implementation `$implementation-address`. To mitigate problems, this contract should check that:

layout collisions can be found here.

- The functions defined in the proxy contract and the ones defined in the implementation contract do not share any function selector, to avoid function selector collisions.
- There are no shared keys between the proxy and the implementation: in other words, all the variables defined in the proxy contract should not collide with the variables defined in the implementation contract.

Update: *Recommendation acknowledged in issue 981 of the “cargo-contract” repository.*

Implement interactive commands

Developers could forget to send one or more arguments to a command, and the default behavior of the command is to fail its execution. Instead, prompting for the missing parameters could be a way to improve the overall experience using the commands.

For example, when the user writes `$ cargo contract` in the console without specifying parameters, the tool could help in the following way:

```
 $ cargo contract call

$ what contract do you want to call?
  [x] Contract A
  [ ] Contract B
  [ ] Contract C
  ....

$ what message would you like to call?
[ ] foo(u32)
[x] bar(u32, u32)
[ ] buz()

$ specify parameters, separated by a comma:
$ 42,50
```

In this example, if the user specifies manually (using flags) what contract they want to call, then only the message name, parameters, and metadata will be asked, and so on. This example can be applied to any of the other commands in the `cargo-contract` tool that receive parameters.

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

> *From all your suggestions we have assigned this one the lowest priority. We generally agree that it could be useful, but it triggered a discussion on the bigger vision for "cargo-contract". Some of us argued that "cargo-contract" should be a slim tool and more user-friendly features should be built on top, in tools like swanky-cli, which uses "cargo-contract" under the hood.*

Education recommendations

The Parity Team could aim to increase adoption of ink! as the primary smart contract language for parachains by targeting two different audiences: auditors and developers. Both audiences may have distinct reasons for learning a new language. Here are some suggestions that we have seen in other blockchain ecosystems that have been validated as valuable.

For auditors:

- Start a bug bounty program in Immunefi, for both cargo-contracts and ink!. This not only contributes to improving the overall security of the technology, but also calls the attention of auditors that might have never heard about it, to learn it and try to hack it to get a bounty.
- Build and/or promote CTFs (Capture the flag), similar to Ethernaut and DamnVulnerableDefi. The first one focuses on potential security issues in the language and the tools to code smart contracts, whereas the latter focuses more on a specific topic, such as DeFi or governance.
- Start a competitive audit program, similar to Code4rena, to get both the ink! and cargo-contract projects audited before a release. This can also be promoted by Parity for other ink! projects, such as OpenBrush.

For developers:

for users to transition from Ethereum.

- Initiate developer grants to fund the development of specific useful features for ink! and cargo-contract. Partner with other companies that use ink! as their primary smart contract language to develop new features. For example, develop new functionality for OpenBrush to implement smart contracts that have not yet been created.

Findings

Type
    Smart contract language and framework
Timeline
    From 2023-01-16
    To 2023-02-10
Languages
    ink! – Rust

Total Issues
    11 (3 resolved)
Critical Severity Issues
    0 (0 resolved)
High Severity Issues
    2 (0 resolved)
Medium Severity Issues
    2 (0 resolved)
Low Severity Issues
    5 (3 resolved)
Notes & Additional Information
    2 (0 resolved)

High Severity

Custom Selectors could facilitate proxy selector clashing attack

ink! has a feature that allows developers to hardcode the selector for a given function. This capability enables function name-changing while maintaining the same selector and also facilitates the creation of language-agnostic contract standards.

create malicious backdoors that are difficult to detect. In contrast to ink!, Solidity requires finding function signatures with matching selectors before taking advantage of this vulnerability, which is not trivial. If such function signatures are found and added, they are likely to raise red flags because the name usually does not make sense to the codebase.

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 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 the potential dangers outlined, it is worth rethinking this feature and looking for an alternative to handle language-agnostic contract standards. Alternatively, requiring the metadata of the implementation contract to build the proxy and preventing the code from being compiled if selector clashing occurs with the implementation may be a viable solution. If the benefits of using custom selectors are not greater than the potential risks, consider removing them.

Update: *Acknowledged, will resolve. The progress can be tracked on issue 1643 of the "ink" repository.*

Potential contract storage layout overlap in upgradable contracts

By default, ink! tries to store all storage struct fields under a single storage cell. This behavior causes an issue for upgradable contracts because both the proxy and the implementation write their Packed fields to the same storage key (`0x00000000`) unless the developer explicitly sets manual keys for the variables inside the implementation contract. As a result, overwrites of the storage could happen.

If the first variable in the implementation is modified, it will change the first variable in the proxy storage layout or some of its bytes, depending on the variable size. The same happens the other way around.

Without sufficient information, developers may fail to properly modify the storage of the implementation contract, which can result in unexpected behavior and potential malfunctioning of the contract. Additionally, this may lead to an unpredictable storage layout, further complicating the

Also, there are no validations between upgrades to check whether the storage layout changed. The documentation specified that developers should not change the order in which the contract state variables are declared, nor their type.

Even if the restriction is violated, the compilation process will still succeed, but it may cause confusion in values or failure in the correct reading of storage. These issues can result in severe errors in the application that is using the contract.

Some mitigations to these problems could be:

- To avoid clashes, define a set of standard slots to store variables present on the proxy code. The EIP-1967 defined in Ethereum can serve as an inspiration.
- Consider adding documentation and examples to illustrate that one of either the implementation or proxy variables needs to use the Lazy collection. The Lazy collection sets the storage keys used for each variable, ensuring that they do not overlap with other variables in the contract.
- Implement the `cargo contract upgrade` command mentioned in the suggestions section. This will retain the storage layout of the implementation and checks that it was not corrupted between upgrades, and check that the variables defined in the proxy were defined using manual keys instead of automatic keys, or that the first position of each variable defined in the proxy does not collide with any of the variables defined in the implementation.

***Update:** Acknowledged, will resolve. The progress can be tracked on issues 1679 and 1680 of the "ink" repository.*

Medium Severity

Nonce reset increases the risk of a successful replay attack

Replay attacks pose a significant security threat to blockchain technologies. To maintain the integrity of signatures in a blockchain network, it is essential to use a nonce, a value that tracks the number of transactions made by a given account.

possibility of a replay attack.

Instead of relying solely on the deadline, consider adding an alternative protection mechanism, like enforcing robust domain separators when hashing messages or advising developers to store the signatures used for a given address in the respective contract. Another solution is to keep the nonce even if the account's balance falls below the minimum required.

Update: *Acknowledged, more documentation will be added to make users aware of this behavior. The progress can be followed in issue 178 of the ink-docs repository. The Parity team stated:*

> *This behavior is normal in the Substrate world and the only thing we can do here is highlight it better to newcomers. We will add documentation about this behavior.*

Unbounded arrays are not possible in `ink!` Smart Contracts

By default, `ink!` tries to store all vector elements under a single storage cell. As a result, querying one item returns all the elements within the vector, but the buffer has a limited capacity (around 16KB in the default configuration). As a consequence, any contract attempting to decode beyond this limit will throw an error, making it impossible to implement certain smart contracts such as `ERC20votes` extension and `EnumerableSet`. If the limit is not exceeded, the operations would consume a large amount of gas in the execution, causing interactions with these contracts to be less appealing due to their cost.

The impact of this flaw might be significant because it limits the capabilities for contract developers. Unbounded arrays are essential for many use cases, and the inability to implement them using `ink!` significantly reduces the range of possibilities for Dapp development.

Consider, if possible, creating another storage collection to store array elements in different slots.

Update: *Acknowledged, will resolve. The progress can be tracked on issue 1682 of the "ink" repository.*

Low Severity

Confusing examples

written in ink! can be upgraded by updating the implementation logic through the `ink::env::set_code_hash` function.
- The second one, `forward-calls`, shows how to perform upgrades by using proxies. Similarly to well-known implementations of the Proxy pattern in solidity, the idea in this approach is to forward calls from the Proxy contract to an implementation contract, using the context of the former but the logic of the latter.

The issue lies in the fact that, in the latter, the `forward` function does not use the implementation of `delegatecall` but performs a regular call operation instead. As a result, the context and storage used in this example will not be the one of the Proxy, but the implementation, breaking the upgradeability pattern.

It is suggested to improve the documentation of how `delegatecall` works in ink! by including examples of both upgradeable and non-upgradeable proxies. Additionally, consider updating the mentioned example using delegate instead of regular calls.

Update: *Resolved in pull request #1697 and pull request #1704.*

Lack of input validation in the `decode` command

The Cargo contract has a decode command to parse the encoded input or output data and extract the underlying values. The feature has two flags. One to indicate the type of data to decode and the other for data itself that has to be a hex value. However, the current implementation of the function is accepting more bytes than the target type expects, which could cause misinterpretation of the data.

Consider updating the implementation to correctly accept only the expected number of bytes for the target function. This measure will reduce the risk of confusion and unexpected results.

Update: *Resolved in pull request #982 at commit 769c112.*

Potential clash between proxy and implementation function selectors

ink! allows developers to set custom selectors for the functions defined on a contract as mentioned in **Custom Selectors could facilitate proxy selector clashing attack**. When this feature is not bytes (similar to how it is done in Solidity).

The issue lies in the fact that the function selector is calculated using only the function's name, without taking into account any other value. This may cause a function selector collision since it is likely to use the same function name in both the proxy and implementation.

Here are some potential mitigations strategies for this issue:

- Develop an `upgrade` command, as mentioned in the suggestions section, to check that there are no repeated function selectors between the proxy and the implementation to which the system is being upgraded.
- Develop a new macro attribute named `proxy`, that could overwrite the implementation of the `compute` function so that it not only uses the name of the function, but also appends to it the name of the proxy contract, a hash of the name of the proxy contract, or any other item that will make the selector different, and properly document it. Additionally, the `proxy` macro will improve the readability of the contract itself, since developers and auditors will know the contract will behave like a proxy.

***Update:** Acknowledged, will resolve. The progress can be tracked on issue 981 of the "cargo-contract" repository.*

Misleading behavior of `ManualKey` functionality

ink! smart contracts include the `ManualKey` feature, which allows developers to specify the value of the key for a Mapping or a Lazy collection. However, a potential issue with this feature is that the key can be set to zero in the code while appearing as a different value in the metadata, leading to confusion and possible errors.

To avoid this issue, it may be worth disallowing users from setting the key of a variable to 0 when using ManualKey. Since developers may rely on the value specified in the code, this change could prevent confusion and improve the reliability of the code.

***Update:** Resolved in pull request #1670 at commit 63c846d.*

Non-determinism in ink! contract builds

is non-deterministic across different operating systems and architectures.

The non-determinism of the build process creates difficulties in contract verification, which makes it challenging to establish trust in the contract and its reliability, both of which are essential for users.

To address this issue, consider standardizing the build process and providing clear guidelines and notifications to developers at the earliest stages, rather than after the contract's deployment. This approach will ensure that contract verification is straightforward, and users can trust the contract.

Update: *Acknowledged, will resolve. The progress can be seen on issue 99 of the "ink-docs" repository and issue 525 of the "cargo-contract" repository.*

Notes & Additional Information

Incomplete Spanish translations

The Spanish version of the documentation has many pages written in English, causing confusion and making it challenging for Spanish speakers to understand the information.

Consider completing the Spanish translations and disabling them until they are production-ready.

Update: *Acknowledged, not resolved yet.*

`README.md` references on internal crates

The internal crates of the ink! repository have a reference to the repository's main README.md instead of their own, leading to a lack of information about the module and broken image links.

This lack of information can hinder the usability of the internal crates, making it difficult for developers to understand the purpose and usage of each crate. Additionally, broken image links can create a negative impression for users.

Creating individual README.md files for each crate would go a long way in resolving this issue.

Update: *Acknowledged, will resolve. The progress can be tracked on issue 1690 of the "ink" repository.*

improving the ink! ecosystem. We are pleased to report that working with the Parity team throughout this process has been fantastic. They have been receptive and open to our recommendations, and the weekly meetings were highly productive.

We see great potential in ink! and its tool `cargo-contract`, which has demonstrated robust security measures and an unwavering commitment to ensuring the safety and security of its users.

Overall, we are confident that ink! and its associated tools will see significant adoption in the future with continued collaboration and ongoing efforts to enhance security. We look forward to seeing the continued evolution and growth of ink!.

For more information about ink!, you can visit the ink! documentation or follow the ink! Twitter. The ink! documentation also has a general explainer on how smart contracts work in Polkadot.

Related Posts

Beefy Zap Audit

OpenBrush Contracts Library Security Review

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

Linea Bridge Audit

Security Audits

Security Audits

Security Audits