Scout

A Security Analysis Tool for Substrate

Vulnerabilities Report

Executive Summary

The purpose of this report is to list relevant security issues introduced in smart contracts developed with ink!. It includes a summary of findings and how the results were procured, a detailed description of each vulnerability/best practice and links to our project's repository providing code examples for each vulnerability.

The team committed to the elaboration of this document was integrated by software security experts, computer science researchers and blockchain developers with vast experience in their fields.

This report is a deliverable for the grant project ScoutCoinfabrik, conducted as a Level 2 grant for the Web3 Grants Program.

Summary of Findings

The following table summarizes our current findings of vulnerabilities for ink! smart contracts. Explanation for the categories can be found in our list of Analysis Categories.

ID	Category	Description	Severity
integer-overflow-or-underflow	Arithmetic	An arithmetic operation overflows or underflows the available memory allocated to the variable.	High
set-contract-storage	Authorization	Insufficient access control on set_contract_storage() function.	High
reentrancy	Reentrancy	Consistency of contract state under recursive calls.	High
panic-error	Validations and error handling	Code panics on error instead of using descriptive enum	Informational
unused-return-enum	Validations and error handling	Return enum from a function is not completely used	Low
dos-unbounded-operation	Denial of Service	Return enum from a function is not completely used	High
dos-unexpected-revert	Denial of Service	DoS due to improper storage.	High

How the results were procured

A problem we discovered in ink! and more generally Substrate-based networks security is the lack of public vulnerabilities disclosed, e.g., as part of security audits of deployed smart contracts. Ideally, we would compile a nicely-sized set of smart contracts with documented vulnerabilities, grow a database from there and use this database as a source to extract snippets, classify vulnerabilities, and develop and tune our detection tools on these snippets. With this missing, we could not come up with a reasonable-sized list of vulnerabilities in real-life smart contracts.

The second best option we came up with was to recreate these vulnerable smart contracts from other sources. As we analyzed different audited and deployed ink! smart contracts, we came upon various pieces of code that were *almost* vulnerable. That is, maybe the smart contracts were not vulnerable in their form, but could become vulnerable after some changes, e.g., removing checks that were in place. These modified smart contracts were consistent with some vulnerable contracts that we found while auditing smart contracts on other blockchains.

This is how we created this list of vulnerabilities.

Description of Vulnerabilities

As a result of our research, we produced seven types of vulnerabilities each falling under the six following different vulnerability categories (so two types fall in one category): Arithmetic, Authorization, Denial of Service, Reentrancy, and Validations and error handling. There follows a description of each vulnerability in the context of ink! smart contracts. In each case, we produced a smart contract exposing a vulnerability of these types. Check the vulnerabilities folder for details on these smart contracts and the vulnerabilities.

1 - Integer Overflow and Integer Underflow

This type of vulnerability occurs when an arithmetic operation attempts to create a numeric value that is outside the valid range in substrate, e.g, a u8 unsigned integer can be at most M:=2**8-1=255, hence the sum *M+1* produces an overflow.

An overflow/underflow is typically caught and generates an error. When it is not caught, the operation will result in an inexact result which could lead to serious problems. We classified this type of vulnerability under the Arithmetic Category type and assigned it a High Severity.

In the context of Substrate, we found that this vulnerability could only be realized if overflow and underflow checks are disabled during compilation. Notwithstanding, there are contexts where developers do turn off checks for valid reasons and hence the reason for including this vulnerability in the list. Check the following code snippet and documentation.

2 - Unauthorized Set Contract Storage

Smart contracts can store important information in memory which changes through the contract's lifecycle. Changes happen via user interaction with the smart contract. An *unauthorized set contract storage* vulnerability happens when a smart contract call allows a user to set or modify contract memory when he was not supposed to be authorized.

Common practice is to have functions with the ability to change security-relevant values in memory to be only accessible to specific roles, e.g, only an admin can call the function reset() which resets auction values. When this does not happen, arbitrary users may alter memory which may impose great damage to the smart contract users. We classified this vulnerability under the Authorization Category and assigned it a High Severity.

In ink! the function set_contract_storage(key: &K, value: &V) can be used to modify the contract storage under a given key. When a smart contract uses this function, the contract needs to check if the caller should be able to alter this storage. If this does not happen, an arbitrary caller may modify balances and other relevant variables. Check the following code snippet and documentation.

3 - Reentrancy

An ink! smart contract can interact with other smart contracts. These operations imply (external) calls where control flow is passed to the called contract until the execution of the called code is over, then the control is delivered back to the caller. A *reentrancy* vulnerability may happen when a user calls a function, this function calls a malicious contract which again calls this same function, and this 'reentrancy' has unexpected repercussions to the contract. This kind of attack was used in Ethereum for the infamous DAO Hack.

This vulnerability may be prevented with the use of the Check-Effect-Interaction pattern that dictates that we first evaluate (check) if the necessary conditions are granted, next we record the effects of the interaction and finally we execute the interaction (e.g., check if the user has funds, subtract the funds from the records, then transfer the funds). There's also so-called *reentrancy guards* which prevent the marked piece of code to be called twice from the same contract call. When the vulnerability may be exercised, the successive calls to the contract may allow the malicious contract to execute a function partially many times, e.g., transferring funds many times but subtracting the funds only once. This vulnerability is of the Reentrancy Category and assigned it a High Severity.

In the context of ink! Substrate smart contracts there are controls preventing reentrancy which could be turned off (validly) using the flag set_allow_reentry(true). A code example of reentrancy can be found at our repository.

4 - Panic error

The use of the `panic!` macro to stop execution when a condition is not met is useful for testing and prototyping but should be avoided in production code. Using `Result` as the return type for functions that can fail is the idiomatic way to handle errors in Rust.

We classified this issue, a deviation for best practices which could have security implications, under the Validations and Error Handling Category with the severity of an Enhancement.

5 - Unused Return enum

Ink! messages can return a Result enum with a custom error type. This is useful for the caller to know what went wrong when the message fails. The definition of the Result type enum consists of two variants: Ok and Err. If any of the variants is not used, the code could be simplified or it could imply a bug.

We put this vulnerability under the Validations and Error Handling Category with a Low Severity.

In our example, we see how lack of revision on the usage of both types (Ok and Err) leads to code where its intended functionality is not realized.

6 - DoS Unbounded Operation

Each block in a Substrate Blockchain has an upper bound on the amount of gas that can be spent, and thus the amount of computation that can be done. This is the Block Gas Limit. If the gas spent by a function call on an ink! smart contract exceeds this limit, the transaction will fail. Sometimes it is the case that the contract logic allows a malicious user to modify conditions so that other users are forced to exhaust gas on standard function calls.

In order to prevent a single transaction from consuming all the gas in a block, unbounded operations must be avoided. This includes loops that do not have a bounded number of iterations, and recursive calls. This vulnerability falls under the Denial of Service Category and has a Medium Severity. A denial of service vulnerability allows the exploiter to hamper the availability of a service rendered by the smart contract. In the context of ink! smart contracts, it can be caused by the exhaustion of gas, storage space, or other failures in the contract's logic.

Needless to say, there are many different ways to cause a DOS vulnerability. This case is relevant and introduced repeatedly by the developer untrained in web3 environments. A code example explaining this particular type of DOS can be found in our repository.

7 - DoS Unexpected Revert With Vector

Another type of Denial of Service attack is called unexpected revert. It occurs by preventing transactions by other users from being successfully executed forcing the blockchain state to revert to its original state.

This vulnerability again falls under the Denial of Service Category and similarly has a Medium Severity.

In this particular example, a Denial of Service through unexpected revert is accomplished by exploiting a smart contract that does not manage storage size errors correctly. It can be prevented by using Mapping instead of Vec to avoid storage limit problems. A code example explaining this particular type of DOS can be found in our repository.