

Case study

Smart Contract Security Audit: Penetration Testing and Static Analysis

Our client is a leading software manufacturer and service provider in the area of business process digitization. They're working on an innovative blockchain solution that enables the streamlined and efficient integration of trade contracts and invoices into blocks. With this solution, their end users will be able to safely store all their data and documents and directly contact all their business partners openly and transparently.

To raise funds for this project, our client will introduce smart contracts on the Ethereum network. These contracts are designed to mint and sell a new cryptocurrency, so our client wanted to avoid any possible security issues in their source code.

Apriorit experts were assigned to perform a security audit of the smart contracts and provide recommendations on code optimization and improvements.








When we received these contracts, they were at the development stage. It's good practice to ask for an independent audit of smart contracts at this stage, as their code becomes immutable once they're deployed to a blockchain network.

The scope of work

Our client provided us with four smart contracts. All of these contracts were written in Solidity and based on the OpenZeppelin library.

Only two of the smart contracts were to be deployed to the Ethereum network. The first was for minting coins and the second was for the crowdsale. Functionality for the crowdsale contract was implemented within two auxiliary contracts. Moreover, the crowdsale contract was designed to allow a token owner to issue tokens manually after accepting payment in any cryptocurrency or as a bonus.

After initial research, we agreed to perform the following tests and analyses as part of our well-rounded audit:

-  Smart contract behavioral consistency analysis
-  Test coverage analysis
-  Penetration testing
-  Static analysis
-  Manual code review and evaluation of code quality
-  Analysis of GAS usage
-  Contract analysis with regards to the host network

We conducted the audit using the Truffle testing framework and Solidity compiler 0.4.25.

Behavioral consistency and test coverage analyses

To perform the security audit, we asked our client to provide us with a white or yellow paper for their smart contracts. Our client was quite surprised by this request, as nobody had offered to analyze their contract logic and expected behavior before. However, we always do this to make sure there's no unexpected behavior during the ICO or any hidden functions.

A white paper can help auditors to understand what initial expectations were imposed on smart contracts.

Unfortunately, our client couldn't provide us with a white paper with smart contract specifications. However, our experts managed to reconstruct the initial requirements and compare them with the actual behavior of the contracts. To do this, we had to rely on the source code, unit tests, and comments that were available within the source repository.

We analyzed the behavior of each contract's functions to see that they complied with initial expectations. The results showed no signs of unexpected or potentially malicious behavior in the contracts, and there were no inconsistencies in their behavior.

The coverage of the smart contracts by the provided unit tests was adequate, yet not complete. We revealed no issues with unit tests as all important functions were covered. The uncovered parts of code were mostly unused functions or negative cases in some conditions. We explained to the client why the unit tests couldn't cover these parts of the code and how to deal with them.

Penetration testing

Penetration testing (or pen testing) is one of our strong points. It allows us to provide clients with reliable and comprehensive security reports. Our pen testing for this project included checking the contracts against our database of vulnerabilities and performing manual attacks on the contracts.

As a result of pen testing, our team discovered no critical issues with the smart contracts, but we found two vulnerabilities that needed to be mitigated:

Vulnerabilities discovered after penetration testing



Lost ownership during deployment



Vulnerability to short address attacks

- **Lost ownership during deployment**

After deploying the coin contract to the network, its ownership would be transferred to the crowdsale contract for minting tokens. When the crowdsale was finished, the original owner of the coin contract would need to access a certain function to stop minting new coins, but this would be impossible because of loss of ownership. Minting of tokens would therefore be allowed forever. To avoid this situation, we recommended that our client add a function that allows returning ownership of the coin smart contract.

- **Vulnerability to short address attacks**

The Ethereum Virtual Machine automatically adds zero bytes if transaction parameters contain fewer bytes than necessary. Hackers can easily exploit this vulnerability by skipping zeros at the end of a transaction parameter and making a user send more tokens. In order to protect both coin and crowdsale smart contracts, we advised that our client handle parameter checking on the frontend when referencing vulnerable functions.

Static analysis

We conducted static analysis of the contracts using our analysis tools. We performed automated static analysis on the full scope of the smart contracts, including base contracts from the OpenZeppelin library. This ensured that no vulnerabilities would be introduced by inherited functions.

Our team confirmed that the OpenZeppelin library is well suited for this application, as no critical issues were found. However, we still provided recommendations to our client on how to deal with minor errors.

Code review

Our experts conducted a code review of the smart contracts. After analyzing the results, we provided our client with the following recommendations:

1. Avoid using deprecated constructor definitions

The coin smart contract defined its constructor as a function with the same name as the contract. This approach is deprecated and potentially dangerous as it allows anyone to execute the constructor, which usually contains critical code such as for setting the contract owner.

2. Use a consistent code style

The code style was somewhat inconsistent throughout the smart contracts. This made the code more difficult to understand and support. A clean and consistent code style also make contracts

look more professional.

3. Avoid leaving commented code

Leaving comments in code makes smart contracts more difficult to read and understand. In our case, we recommended that our client remove unnecessary parts of the smart contract from OpenZeppelin, which is the basis for these contracts.

4. Avoid unused code and unnecessary inheritance

The inheritance hierarchy of the crowdsale contract reflected the hierarchy of the original OpenZeppelin implementation. However, in the case of custom implementation for a specific crowdsale, there's rarely a need for multiple levels of inheritance. In our client's case, the code of the base contracts was used only once, so there was no real need for two auxiliary smart contracts. The functionality of the crowdsale could easily fit within a single smart contract.

5. Use a fixed version of the Solidity compiler

It's considered best practice to use a fixed compiler version. If left unfixed, a compiler version that's different from the one used during development may be used during deployment. This can lead to unexpected errors and possible vulnerabilities because of changes in a newer version of the compiler. This also makes a contract more difficult to reuse, since different developers may have different versions of the compiler.

6. Update the compiler version

The Solidity compiler is constantly being improved. It's important to use the latest stable version of the compiler while the contract is under active development. Newer versions of the compiler have important security updates and help to avoid obsolete practices.

7. Add the possibility to retrieve Ether directly from the contract's account

Sufficient measures were made to ensure that the contracts can't have any Ether stuck through normal use. But payable functions aren't the only way to deliver Ether to a contract account. There's no security vulnerability related to this, but all the same such Ether will be stuck forever. It's not uncommon to add a general function that allows for retrieval of any stuck Ether in a trusted account.

Analysis of GAS usage

Our Apriorit experts advised on how to reduce GAS consumption during coin distribution. Particularly, we found that GAS savings could be achieved in the following ways:

- **By limiting the number of events**

This would significantly reduce the GAS usage of the buyTokens function of the crowdsale contract. In the client's implementation, three events were triggered: the TokenPurchase event of the crowdsale contract and the Transfer and Mint events of the coin contract. A single event would be enough to notify clients about a successful purchase.

- **By sending coins in batches**

This would save additional GAS on the base transaction costs for each investor. However, this approach requires limiting the batch size to avoid denial of service if too many customers are accidentally added in a single batch.

Analysis of smart contracts in regards to the host network

We also analyzed the smart contracts in regards to the Ethereum network. It's well known that Ethereum is now vulnerable to denial of service attacks, so some smart contracts may suffer from this. Fortunately, our specialists revealed that our client's smart contracts were free of time-constrained and other volatile functionality that depends on the state of the network.

Final results and recommendations

The results of our audit revealed no critical issues with our client's smart contracts. However, we recommended that our client make some optimizations and improvements to the contract code.

Our client's team analyzed and implemented all of our recommendations and are now fully ready to deploy their smart contracts to the Ethereum network.

Though our client was quite surprised that we conducted contract behavioral analysis, we consider this an essential part of security testing. The overall success of a security audit depends on the following:



Timely execution of security testing, as no code changes can be implemented after contracts are deployed



The availability of a contract white paper so auditors can verify that there's no unexpected behavior in a smart contract



Only conducting all mentioned tests and analyses can ensure the completeness of testing results

Hiring independent penetration testers is common practice when developing a smart contract. Apriorit pen testers have a good understanding of the inner processes of smart contracts as well as the whole blockchain network, so they're capable of finding even the most non-obvious defects in smart contract code that are subtle at the development stage. In addition, our auditors provide useful advice on how to improve code quality and efficiency.