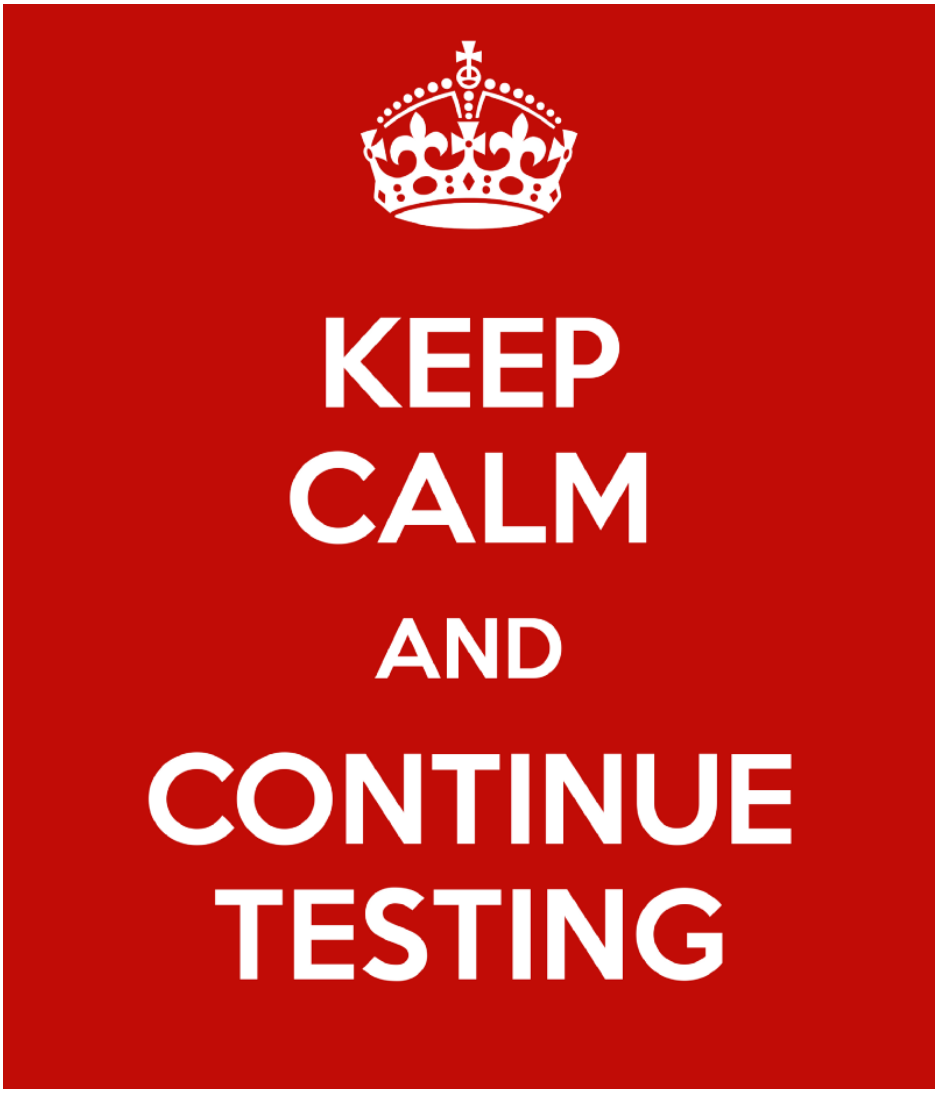
Merunas Grincalaitis Follow
I'm the Ethereum expert.
Sep 17, 2017 · 12 min read

The ultimate guide to audit a Smart Contract + Most dangerous attacks in Solidity



Have you ever wondered how to audit a Smart Contract to find security breaches?

You can learn it by yourself or you can use this handy step-by-step guide to know exactly what to do at what moment and audit those contract.

I've been researching several Smart Contract audits and I've learn the most common steps they took to extract all the essential information from any contract.

You'll learn the following:

- Steps to take in order to fully audit a Smart Contract to generate a pdf with all the findings.
- The most important types of attacks that you need to know as an Ethereum Smart Contract Auditor.
- What to look for in a contract and useful tips that you won't find anywhere else but here.

Let's get right to it and start auditing contracts:

. . .

How to audit a Smart Contract

To teach you exactly how to do this, I'll audit one of my own contracts. This way you'll see a real world audit that you can apply by yourself.

Now you may ask: **what exactly is a Smart Contract audit?**

> *A Smart Contract audit is the process investigating carefully a piece of code, in this case a Solidity contract to find bugs, vulnerabilities and risks before the code is deployed and used in the main Ethereum's network where it won't be modifiable. It's just for discussion purposes.*

Note that an audit isn't a legal document that verifies that the code is secure. Nobody can 100% assure that the code won't have future bugs or vulnerabilities. It's a guarantee that your code has been revised by an expert and it's secure.

To discuss possible improvements and mostly to find bugs and vulnerabilities that could risk people's ether.

Once that's clear, let's take a look at the structure of a Smart Contrac Audit:

1. **Disclaimer:** Here you'll say that the audit is not a legally binding document and that it doesn't guarantee anything. That it's just a discussion document.
2. **Overview of the audit and nice features:** A quick view of the Smart Contract that will be audited and good practices found.
3. **Attacks made to the contract:** In this section you'll talk about the attacks done to the contract and the results. Just to verify that it is, in fact secure.
4. **Critical vulnerabilities found in the contract:** Critical issues that could damage heavily the integrity of the contract. Some bug that would allow attackers to steal ether is a critical issue.
5. **Medium vulnerabilities found in the contract:** Those vulnerabilities that could damage the contract but with some kind of limitation. Like a bug allowing people to modify a random variable.
6. **Low severity vulnerabilities found:** Those are the issues that don't really damage the contract and could exists in the deployed version of the contract.
7. **Line by line comments:** In this section you'll analize the most important lines where you see potential improvements.
8. **Summary of the audit:** Your opinion about the contract and final conclusions about the audit.

Save that structure somewhere safe because it's all you need to really audit a Smart Contract securely. It will really help you find those hard to find vulnerabilities.

I recommend you starting by the point 7 "Line by line comments" because when analizing the contract line by line, you'll find the most important issues and you'll see what's missing. What could be changed or improved.

I'll show you a disclaimer that you can use as it is for the first step of the audit. You can go to point 1 and go down from there until the audit is finished.

Next I'll show you my personal audit that I made for one of my contracts using that structure with those steps. You'll also see a description of the most important attacks that can be made to a Smart Contract on step 3.

Casino Ethereum Audit

You can see the code being audited in my github: https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol

This is the audit of my contract Casino.sol:

Introduction

In this Smart Contract audit we'll cover the following topics:

1. Disclaimer
2. Overview of the audit and nice features
3. Attack made to the contract
4. Critical vulnerabilites found in the contract
5. Medium vulnerabilites found in the contract

6. Low severity vulnerabilites found
7. Line by line comments
8. Summary of the audit

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

2. Overview

The project has only one file, the `Casino.sol` file which contains 142 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation for the functions which is good to understand quickly how everything is supposed to work.

The project implements the Oraclize API to generate truly random numbers on the blockchain using a centralized service.

Generating random numbers on the blockchain is quite a hard topic because one of the core values of Ethereum is predictability whose goal is to not have undefined values.

Therefore the use of the trusted number generation from Oraclize is considered good practice since they generate random numbers off-chain. It implement modifiers and a callback function that verifies that the information is comming from a trusted entity.

The purpose of this Smart Contract is participate in a random lotery where people bet for a number between 1 and 9. When 10 people place their bets, the prize is automatically distributed across the winners. There's also a minimum bet amount for each user.

Each player can only bet once during each game and the winner number is only generated when the limit of bets has been reached.

Nice Features

The contract provides a good suite of functionality that will be useful for the entire contract:

1. Secure random number generation with Oraclize and proof verification on the callback.
2. Modifiers to check for the end game, blocking the critical functions until the rewards are distributed.
3. Good amount of check for verifying that the bet function is used propertly.
4. Secure number winner generation only when the maximum of bets has been reached.

3. Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

Reentrancy attack

> *This attack consists on recursively calling the* `call.value()` *method in a ERC20 token to extract the ether stored on the contract if the user is not updating the* `balance` *of the sender before sending the ether.*

When you call a function to send ether to a contract, you can use the fallback function to execute again that function until the ether of the contract is extracted.

Because this contract uses `transfer()` instead of `call.value()`, there's no risk of reentrancy attacks since the transfer function only allows to use 23.000 gas which you can only use for an event to log data and throws on failure.

That way you're unable to recursively call again the sender function thus avoiding the reentrancy attack.

The transfer function is called only when distributing the rewards to the winners which happens once per game, when the game ends. So there shouldn't be any problem with reentrancy attacks.

Note that the condition to call this function is that the number of bets is bigger or equal the limit of 10 bets but that condition isn't updated until the end of the `distributePrizes()` function which is risky because someone could theorically be able to call that function and execute all the logic before updating the state.

So my recommendation is to update the condition when the function starts and set the number of bets to 0 to avoid calling the `distributePrizes()` more times than expected.

Over and under flows

> *An overflow happens when the limit of the type varibale `uint256`, 2**256, is exceeded. What happens is that the value resets to zero instead of incrementing more.*

For instance, if I want to assign a value to a uint bigger than 2**256 it will simple go to 0—this is dangerous.

On the other hand, an underflow happens when you try to substract 0 minus a number bigger than 0.

For example, if you substract 0 -1 the result will be = 2**256 instead of -1.

This is quite dangerous when dealing with ether. Hovewer in this contract there's no substraction anywhere so there's no risk of underflows.

The only time an overflow could happen is when you `bet()` for a number and the `totalBet` variable's amount is increased:

```
totalBet += msg.value;
```

Someone could send a huge amount of ether that would exceed the limit of 2**256 and therefore making the total bet 0. This is improbable but the risk is there.

Therefore I recommend using a library like the OpenZeppelin's SafeMath.sol.

It'll help you make secure calculations without the risk of under or over flows.

The way you use it is by importing the library, activating it for uint256 and then using the function .mul(), .add(), sub() and .div(). For instance:

```
import './SafeMath.sol';

contract Casino {
    using SafeMath for uint256;

    function example(uint256 _value) {
        uint number = msg.value.add(_value);
    }
}
```

Replay attack

> *The replay attack consists on making a transaction on one blockchain like the original Ethereum's blockchain and then repeating it on another blockchain like the Ethereum's classic blockchain.*

The ether is transfered like a normal transaction from a blockchain to another.

Though its no longer a problem because since the version 1.5.3 of Geth and 1.4.4 of Parity both implement the attack protection EIP 155 by Vitalik Buterin: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md

So the people that will use the contract depend on their own ability to be updated with those programs to keep themselves secure.

Reordering attack

> *This attack consists in that a miner or other party tries to "race" with a smart contract participant by inserting their own information into a list or mapping so the attacker may be lucky in getting their own information stored on the contract.*

When a user places his `bet()` and the data is saved on the blockchain, anybody will be able to see what number has been bet by simply calling the public mapping `playerBetsNumber` .

That mapping shows what number has each person selected. Hence, in the transaction data you can esasily see the amount of ether that has been bet.

This could happen in the `distributePrizes()` function because it's called when the callback of the random number generation is invoked.

Because the condition of that function isn't updating until the end of it, there is a risk of a reordering attack.

Consequently, my recommendation is like I said before: to update the condition of the number of bets at the beggining of the `distributePrizes()` function to avoid this kind of undexpected behaviour.

Short address attack

This attack affects ERC20 tokens, was discovered by the Golem team and consists of the following:

- A user creates an ethereum wallet with a traling 0, which is not hard because it's only a digit. For instance: 0xiofa8d97756as7df5sd8f75g8675ds8gsdg0
- Then he buys tokens by removing the last zero:

Buy 1000 tokens from account 0xiofa8d97756as7df5sd8f75g8675ds8gsdg

- If the token contract has enought amount of tokens and the buy function doesn't check the length of the address of the sender, the Ethereum's virtual machine will just add zeroes to the transaction until the address is complete.
- The virtual machine will return 256000 for each 1000 tokens bought. This is a bug of the virtual machine that's yet not fixed so whenever you want to buy tokens make sure to check the length of the address.

The contract isn't vulnerable to this attack since it's not an ERC20 token.

You can read more about the attack here: http://vessenes.com/the-erc20-short-address-attack-explained/

4. Critical vulnerabilites found in the contract

There aren't critical issues in the smart contract audited.

5. Medium vulnerabilites found in the contract

The function `checkPlayerExists()` isn't constant when it should.

Ergo this increases the costs of the gas every time the function is called which is a big problem when dealing with a lot of calls.

Make it constant and avoid expensive gas executions.

6. Low severity vulnerabilites found

1. You're using `assert()` instead of `require()` in all the cases and at the beggining of the functions `__callback()` and `pay()` .

Assert and require behave almost identically but the assert function is used to validate contract state after making changes, while require is normally used at the top of the functions to verify the input of the function.

2. You're defining the variable players at the beggining of the contract but you aren't using it anywhere. Remove it if you're not gonna use it.

7. Line by line comments

- **Line 1:** You're specifiying a pragma version with the caret symbol (^) up front which tells the compiler to use any version of solidity bigger than `0.4.11` .

This is not a good practice since there could be major changes between versions that would make your code unstable. That's why I recommend to set a fixed version without the caret like 0.4.11.

- **Line 14:** You're defining the `uint` variable `totalBet` in singular which isn't correct since it stores the sum of all the bets. My recommendation is to change it to plural, totalBets instead of totalBet.

- **Line 24:** You're defining the constant variable in caps which is a good practice to know that it's a fixed, unmodified variable.

- **Line 30:** Like I said before, you're defining an unused array `player` . Remove it if you're not gonna use it.

- **Line 60:** The function `checkPlayerExists()` should be constant but it isn't. Because it doesn't modify the state of the contract, make it constant and save some gas every time it executes.

Also it's good practice to specify the type of visibility the function has even if it's the default value of public to avoid confusion. To that end, add the public visibility parameter to the function explicitly.

- **Line 61:** You're not checking if the parameter player is sent and well formated. Make sure to use a `require(player != address(0));` at the top of that function to checking if an invalid address exists or not. Also check the length of the address to protect the code from short address attacks just in case.

- **Line 69:** Again, specify the visibility of the function `bet()` explicity to avoid confusion and know exacly how it's supposed to get called.

- **Line 72:** Use `require()` instead of `assert()` for checking that the input of the function is well formated.

Likewise at the beggining of the functions require() is more often used. Change all those assert() at the beggining to require().

- **Line 90:** You're using a simple sum in the `msg.value` variable. This could lead to overflows since the value could get quite big. That's why I recommend to check for overflows and underflows any time you're making a calculation.

- **Line 98:** The function `generateNumberWinner()` should be internal since you don't want anybody to execute it outside the contract.
- **Line 103:** You're saving the result of `oraclize_newRandomDSQuery()` in a bytes32 variable. This isn't required to execute the callback function. Also you're not using that variable anywhere. So I recommend to not assign that value and just call the function.
- **Line 110:** The `__callback()` function should be external because you only want it to be called from outside.
- **Line 117:** That assert should be require for the reasons I explained above.
- **Line 119:** You're using `shae()` which isn't a good practice since the algorythm used isn't exacly sha3 but keccak256. My recommendation is to change it to keccak256() instead for clarity purposes.
- **Line 125:** The function `distributePrizes()` should be internal because only the contract should be able to call it.
- **Line 129:** Although you're using a variable sized array for a loop it's not that bad because the amount of winners should be limited to less than 100.

8. Summary of the audit

Overall the code is well commented and clear on what it's supposed to do for each function.

The mechanism to bet and distribute rewards is quite simple so it shouldn't bring major issues.

My final recommendation would be to pay more attention to the visibility of the functions since it's quite important to define who's supposed to executed the functions and to follow best practices regarding the use of assert, require and keccak.

This is a secure contract that will store safely the funds while it's working.

. . .

Conclusion

That's all the audit that I did by myself using the structure explained at the beggining. Hope you learned something and now you're capable of making secure audits to other Smart Contracts.

Keep learning and improving your knowledge about contract security, best practices and new functionalities.

If you enjoyed this article make sure to:

- Follow me on twitter @merunas2
- Clap this article as long as your hart desires
- Share the article with your fellow Ethereum developers
- Comment your questions, improvements and typos found
- Bring me work as an Ethereum developer. I'm always willing to take more work!