Solidproof was used to audit D2T token and presale contract to measure security and vulnerability to future attacks and misusing. The engagement was technical in nature and focused on identifying the security flaws in the design and implementation of the contracts. They provided Solidproof.io with access to their code repository and whitepaper. Like checking A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken or A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.

First they have checked the functionality of contracts. For example they checked if the contract is really pausable or not. Or if the functions like totalSupply, balanceOf ,… work fine or not.

At the next step they tried different attacks on contracts and checked how secure the contract are.

- Unencrypted Private Data On-Chain: it is a common misconception that private type variables cannot be read. Even if your contract is not published, attackers can look at contract transactions to determine values stored in the state of the contract. For this reason, it's important that unencrypted private data is not stored in the contract code or state.

Code With No Effects :Reduce the attack surface for malicious contracts trying to hijack control flow after an external call. Motivation: The Ethereum Virtual Machine does not allow for concurrency. When calling an external address, for example when transferring ether to another account, the calling contract is also transferring the control flow to the external entity. This external entity is now in charge of the control flow and can execute any inherent code, in case it is another contract. Most of the times, this will not cause any problems, but in case the called contract is acting in bad faith, it could alter the control flow and return it in an unexpected state to the initial contract. A possible attack vector is a re-entrancy attack, in which the malicious contract is reentering the initial contract, before the first instance of the function containing the call is finished. This attack can be used to repeatedly invoke functions that should only be executed once and was part of the most prominent hack in Ethereum history: the DAO exploit. The described vulnerability is not present in other software environments, making it hard to avoid for developers not familiar with the quirks of smart contract development. The pattern presented in this section, together with the Secure Ether Transfer pattern, aims to provide a safe solution, in order to make functions unassailable against re-entrancy attacks of any form.

Message call with hardcoded gas amount :The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction

Hash Collisions With Multiple Variable Length Arguments : Hash Collisions With Multiple Variable Length Arguments .Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision. Since abi.encodePacked() packs all elements in order regardless of whether they're part of an array, you can move elements between arrays and, so long as all elements are in the same order, it will return the same encoding. In a signature verification situation, an attacker could exploit this by modifying the position of elements in a previous function call to effectively bypass authorization.

Unexpected Ether balance: Unexpected Ether balance, Contracts can behave erroneously when they strictly assume a specific Ether balance. It is always possible to forcibly send ether to a contract (without triggering its fallback function), using selfdestruct, or by mining to the account. In the worst case scenario this could lead to DOS conditions that might render the contract unusable.

Presence of unused variables: Presence of unused variables. Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can: cause an increase in computations (and unnecessary gas consumption)indicate bugs or malformed data structures and they are generally a sign of poor code qualitycause code noise and decrease readability of the codeand other attacks like Right-To-Left- Override control character (U+202E), Typographical Error, DoS With Block Gas Limit, Arbitrary Jump with Function Type Variable, Incorrect Inheritance Order, Write to Arbitrary, StorageLocation, Requirement Violation, Lack of Proper Signature Verification, Missing Protection against Signature Replay Attacks, Weak Sources of Randomness from Chain Attributes, Shadowing State Variables, Incorrect Constructor Name, Signature Malleability, Timestamp Dependence, Authorization through tx.origin, Transaction Order Dependence, DoS with Failed Call, Delegatecall to Untrusted Callee, Use of Deprecated Solidity Functions, Assert Violation, Uninitialized Storage Pointer, State Variable Default Visibility, Reentrancy, …