// HALBORN TOKEN CONTRACT

Smart Contracts CTF audit report

Prepared by: Michael Maximov
Date of Engagement:
2022, May 21st - June 5th

CONTACTS

Michael Maximov

maksymovmm@gmail.com

OVERVIEW

1.1 INTRO

I have decided to participate in your challenge on May 21st after an awesome talk with Thomas Richard.

I have been interested in crypto space since 2017. However, I never dug deep into it, so I decided to go ahead and give it a chance.

Thank you guys for this awesome opportunity, I enjoyed participating in your CTF a lot.

1.2 AUDIT SUMMARY

I have dedicated a week to familiarize myself with all the basics i need to dive into audit:

1. I have read the “Mastering Ethereum” book by Andreas Antonoupoulos and dr.Gavin Wood
2. Familiarized myself with the basics of solidity using cryptozombies.io
3. Made a “learning speed run” using “Ethereum, solidity, brownie course” by Patrick Collins on Youtube
4. Started “playing” with coding and deploying my own contracts. At the same time I have explored some of the basic smart contracts security topics. Mostly following

along with the “console cowboys course”

Then, during next week, I have started exploring Halborn CTF contracts:

1. For 6 days exploring these contracts, I identified and managed to exploit 6 vulnerabilities, outlined in the next sections
2. Had a lot of fun ;)

1.3 APPROACH

I have followed a fully manual approach while auditing smart contracts:

1. Firstly, I took a quick look at the architecture and the layout.
2. Then I googled literally everything that seemed unfamiliar to me. (At the end of this stage, I already had a full understanding what this contract is for)
3. When I identified purposes which all of the functions intended for, I have created a list of common pitfalls which may occur in implementing these functionalities.
4. In the next step I have created a list of possible attack vectors regarding given functionalities. (At the end of this phase, I already had few leads to investigate further)
5. Then I started a line-by-line code review. I have written what every particular function is supposed to do in plain english.
6. I have prepared a VScode brownie environment for testing.
7. Also I have used Remix for transactions debugging frequently.

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur.

This framework works for communicating the characteristics and impacts of technology vulnerabilities.

The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores.

For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.

4 - High probability of an incident occurring.

3 - Potential of a security incident in the long term.

2 - Low probability of an incident occurring.

1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.

4 - May cause a significant level of impact or loss.

3 - May cause a partial impact or loss to many.

2 - May cause temporary impact or loss.

1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating

a value of 10 to 1 with 10 being the highest level of security risk.

10 - **CRITICAL**

9 - 8 - HIGH

7 - 6 - MEDIUM

5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The Engagement was scoped to the following contract:

HalbornToken.sol

2. FINDINGS AND TECH DETAILS

2.1 SETSIGNER FUNCTION ALLOWS ANYONE OBTAIN THE ‘SIGNER’ ROLE

1. setSigner function has a faulty require check:

```
function setSigner(address _newSigner) public {
    // this is weird. are they checking for msg.sender IS whoever but not the current deployer?
    require(msg.sender != signer, "You are not the current signer");
    signer = _newSigner;
}
```

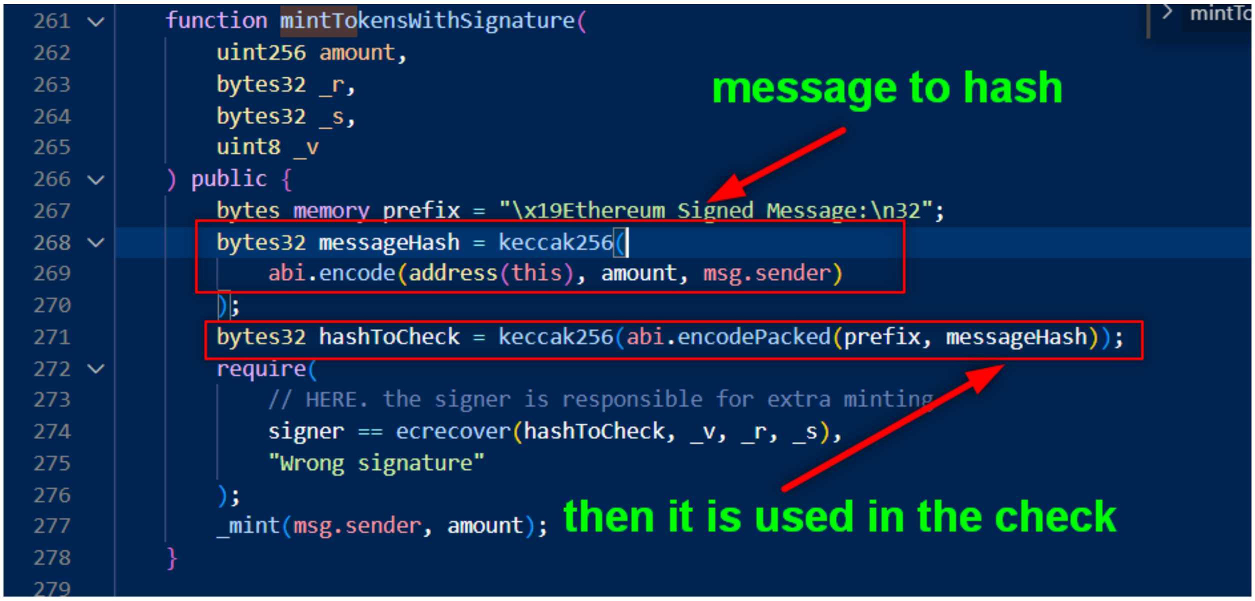
This line of code says: execute the code **IF** msg.sender **IS NOT** the signer.

As long as the external caller **IS NOT** the current signer, They can call this function and assign the 'signer' role to an arbitrary address of their choice.

2. **mintTokensWithSignature** function takes **amount** and ECDSA **r,s,v** values as arguments

(*which are used to prove that this signature comes from the private key who has the 'signer' role*)

An attacker just need to hash the message and sign it off-chain, then supply respective ECDSA values to that function and they would be able to mint tokens as a signer.



Provided, an attacker assigned the 'signer' role to themself during the previous step, they can mint unlimited tokens.

Proof of Concept:

- call a setSigner function from the attacker's account. Assign a 'signer' role to attacker
- in the next steps produce a valid messageHash, which will successfully pass the require clause
- firstly, abi_encode **address of the contract**, **amount** you'll want to mint, and **attacker's address**.
- after that, pass this abi_encoded message to hash function
- after that, sign this hashedMessage
- extract **r, s, v** ECDSA values and pass these values as arguments to the **mintWithSignature** function.

I have prepared the brownie Proof Of Concept script for you to verify this issue.

Take following steps:

1. Open halborn.7z archive
2. You'll find **token/scripts/deploy_halborn_token.py**
3. Run this script. It will deploy a **HalbornToken.sol** contract
4. Find **token/scripts/poc_signer.py**
5. Run this script. It will prepare the required state for you and execute the attack scenario. ***(###I have included multiple comment lines to help you understand the flow)***

Risk level:

Likelihood 5

Impact 5

Remediation Recommendation:

Invert require clause logic like that:

```
require(msg.sender == signer, "You are not the current signer")
```

- note that comparison operator here changed to '**==**'

Also, for even more security concerns, please refer to 2.4 finding

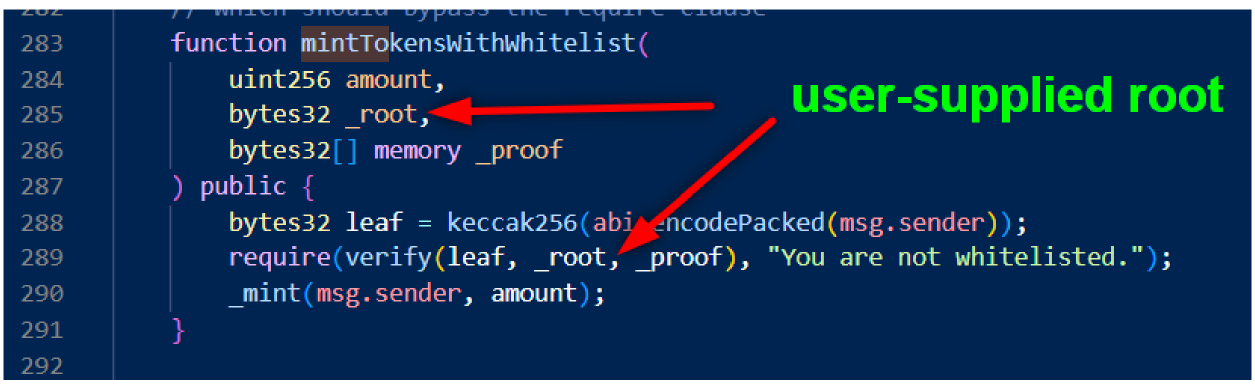
2.2 MINTWITHWHITELIST FUNCTION ACCEPTS USER_SUPPLIED ROOT HASH. ANYONE CAN MINT TOKENS

This whitelist function is used to mint new tokens.

It is based on the 'merkle tree proof'

OpenZeppelin has a contract which handles 'merkle tree proofs' efficiently.

The fact, that developers didn't use a secure and tested solution and decided to come up with their own, made me suspicious:



```
function mintTokensWithWhitelist(
    uint256 amount,
    bytes32 _root,
    bytes32[] memory _proof
) public {
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender));
    require(verify(leaf, _root, _proof), "You are not whitelisted.");
    _mint(msg.sender, amount);
}
```


1. As you can see, this function accepts an arbitrary root hash to compare with.

Instead of comparing user-supplied `proof` and `leaf` with the `root` specified during the constructor phase.

2. It means that nothing stops an attacker from feeding

a valid proof and root for the given leaf(attacker's address) to the verify function, hence bypassing the check.

Proof of Concept:

- create a valid merkle tree, which will bypass the check:
- hash your own address (it should be passed as a ‘leaf’ argument to verify function eventually)
- hash the Merkle Tree neighbor’s address (it may be random)
- calculate root hash for leaf and neighbor nodes
- store the neighbor’s hash in the bytes32 array (it is the required type of the function argument)
- then, finally feed those arguments and execute the **mintWithWhitelist** function

I have prepared the brownie Proof Of Concept script for you to quickly verify this issue.

Take following steps:

1. Open halborn.7z archive
2. You’ll find **token/scripts/deploy_halborn_token.py**
3. Run this script. It will deploy a **HalbornToken.sol** contract
4. Find **token/scripts/poc_whitelist.py**
5. Run this script. It will prepare the required state for you and execute the attack scenario.

(###I have included multiple comment lines to help you understand the flow)

Risk level:

Likelihood 5

Impact 5

Remediation Recommendation:

Consider using MerkleTreeProof.sol by OpenZeppelin instead of your custom code.

If it is not possible, then do not compare the leaf and the proof with the user-supplied root hash. Compare them with the ‘root hash’ stored in a root_ storage variable.

2.3 INTEGER OVERFLOW IN CALCMAXTRANSFERRABLE FUNCTION

When I tested every 'weird use case' i could come up with

and ran out of any ideas, I decided to use a smart contract as intended and found this issue:

1. There is a **calcMaxTransferable** function which is called **EVERY TIME** any user attempts to transfer tokens.

This is due to the **_beforeTokenTransfer** hook, whose purpose is executing special logic before every token transfer.

2. This return statement might produce an integer overflow:

```
        if (timelockedTokens[who] < maxTokens) {
            return balanceOf(who);
        }

        return balanceOf(who) - timelockedTokens[who] + maxTokens;
    }
```

I have noticed an EVM revert on the overflow transaction to calcMaxTransferable, which won't let a user to transfer their tokens (even those released ones)

After spending the whole day googling and playing with it, I still wasn't able to explain this behavior (why tokens are blocked).

Until I have found the same issue in the Halborn's public reports repository.

So, I feel like I've cheated here a bit :)

Anyway, here is the

Proof of Concept:

- execute newTimeLock from account A for 100000000000000000000 tokens, vestTime = chain.time() + 1, cliffTime = chain.time() + 15770000, disbursementPeriod = 31540000

- “Time Travel” 6 months to the end of the cliff period: chain.sleep(15770000). Mine a new block with this timestamp.

- Observe that 50000000000000000000 of the account A tokens are released.

- Transfer 20000000000000000000 tokens from account A to account B.

- Try to make one more 20000000000000000000 transfer from account A to account B.

- Observe the transaction reverts with an Integer overflow error.

Therefore, tokens are totally locked now till the end of the disbursement Period.

I have prepared the brownie Proof Of Concept script for you to quickly verify this issue.

Take following steps:

1.Open halborn.7z archive

2. You'll find **token/scripts/deploy_halborn_token.py**

3. Run this script. It will deploy a **HalbornToken.sol** contract and also transfer tokens to account A

4. Find **token/scripts/poc_calcMax.py**

5. Run this script. It will prepare the required state for you and execute the attack scenario.

(###I have included multiple comment lines to help you understand the flow)

Risk level:

Likelihood 5

Impact 5

Remediation Recommendation:

Consider rewriting the **calcMaxTransferrable** function so that it would not produce integer overflow.

2.4 LACK OF 0x00 ADDRESS CHECK IN SETSIGNER FUNCTION MIGHT LEAD TO LEAVING THE CONTRACT WITHOUT THE POSSIBILITY TO MINT NEW TOKENS WITH SIGNATURE

I'm not sure if this one is worth reporting as a CRITICAL issue. However, I think this vulnerability might lead to devastating consequences, so I decided to include it in my report.

Note, that **setSigner** function does not perform the check on the **_newSigner** parameter to make sure it **IS NOT** a '**0x00**' address:

```
/// @dev Sets a new signer account. Only the current signer can call this function
function setSigner(address _newSigner) public {
    require(msg.sender != signer, "You are not the current signer");
    signer = _newSigner;
}
```

no 0x00 address check

Checking if the '**to**' address is not a '**0x00**' address is a best practice.

In this very case, the '**0x00**' address might be assigned to a 'signer' role by anyone. This leaves the contract without the possibility to mint new tokens with signature.

BUT. As long as this function allows **anyone** to assign an **arbitrary** address to this role, it has a not-so-critical impact.

However, if you implement a check required to remediate the **2.1** issue with setSigner function:

```
function setSigner(address _newSigner) public {
    require(msg.sender == signer, "You are not the current signer");
    signer = _newSigner;
}
```

And after that, if one accidentally (*or maliciously - on purpose*) passes the **‘0x00’** address as a **_newSigner** argument, they block the possibility to mint new tokens FOREVER.

Proof of Concept:

I have prepared the brownie Proof Of Concept script for you to quickly verify this issue.

Take following steps:

1.Open halborn.7z archive

2. You’ll find **token/scripts/deploy_halborn_token.py**

3. Run this script. It will deploy a **HalbornToken.sol** contract

4. Find **token/scripts/poc_signer_to_0.py**

5. Run this script. It will prepare the required state for you and execute the attack.

(###I have included multiple comment lines to help you understand the flow)

Risk level:

Likelihood 4

Impact 5

Remediation Recommendation:

Consider include one more require statement to the setSigner function implementation

```
/// @dev Sets a new signer account. Only the current signer can call this function
function setSigner(address _newSigner) public {
    require(msg.sender == signer, "You are not the current signer");
    // This is the 0x00 address check
    require(_newSigner != address(0), "'To' address MUST NOT be zero");

    signer = _newSigner;
}
```