EtherCamp's Hacker Gold (HKG) public code audit

OPENZEPPELIN SECURITY | OCTOBER 17, 2016

Security Audits

We've been asked by our friends at ether.camp to review the code for their soon-to-launch Hacker Gold (HKG) token, and to publish the results of our work.

The audited contract can be found in the hkg-tests branch of their GitHub repo, specifically, commit 2529ffe5efd5294b44f1bc89dc9a4721a7b16409. Main contract file is `HackerGold.sol`.

Here's our assessment and recommendations, in order of importance:

Severe

We have not found any severe security problems with the code.

Potential problems

Timestamp usage

We verified that the timestamp list included in the HackerGold constructor correspond to the times stated in comments, and they check.

That said, there's a problem with using timestamps and `now` (alias for `block.timestamp`) for contract logic, based on the fact that miners can perform some manipulation. In general, it's better

Given the nature of the contract, we think the risk of miner manipulation is really low. The potential damage is also limited: miners could only slightly manipulate price of HKG near the times where it changes (p1, p2, etc.). We recommend the EtherCamp team to consider the potential risk and switch to `block.number` if necessary.

For more info on this topic, see this stack exchange question

Use safe math

There are many unchecked math operations in the code. We couldn't find any related attack vectors on the HKG contract, but it's always better to be safe and perform checked operations. Consider using a safe math library, or performing pre-condition checks on any math operation.

The fact that HKG supply is limited to 4,000,000 ether (and thus at most 800,000,000 HKG assuming all tokens are created at the best possible price) helps prevent possible overflows.

Be careful with small transactions

Token calculation at line 90 of HackerGold.sol uses an integer division by 1,000,000,000,000,000, which truncates the result to an integer value. Any call to the function `createHKG` with a `msg.value` less than 0.001 ether will be lost, resulting in a 0 token balance. Note that the value of 0.0005 ether in theory corresponds to 0.1 HKG, which is a valid representation (less than 3 decimals), but in this case it is considered zero HKG.

Moreover, any amount of ether after the 3rd decimal place will be ignored. If 0.12345678 ether is sent to `createHKG`, only 24.6 HKG will be created and assigned to the `msg.sender`.

Consider changing that line to:

```
uint tokens = msg.value * getPrice() / 1000000000000000;
```

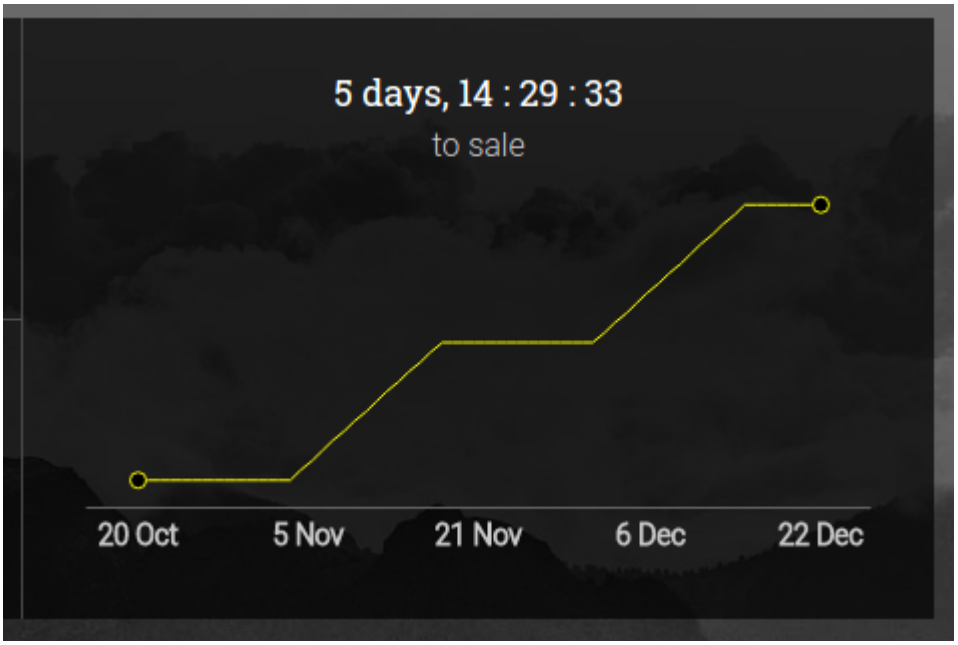
This yields 24.691 HKG for the previous example, which is better, but still ignores some small amounts. Consider warning users against sending amounts smaller than 0.001 ether.

EDIT: EtherCamp fixed this problem in this commit

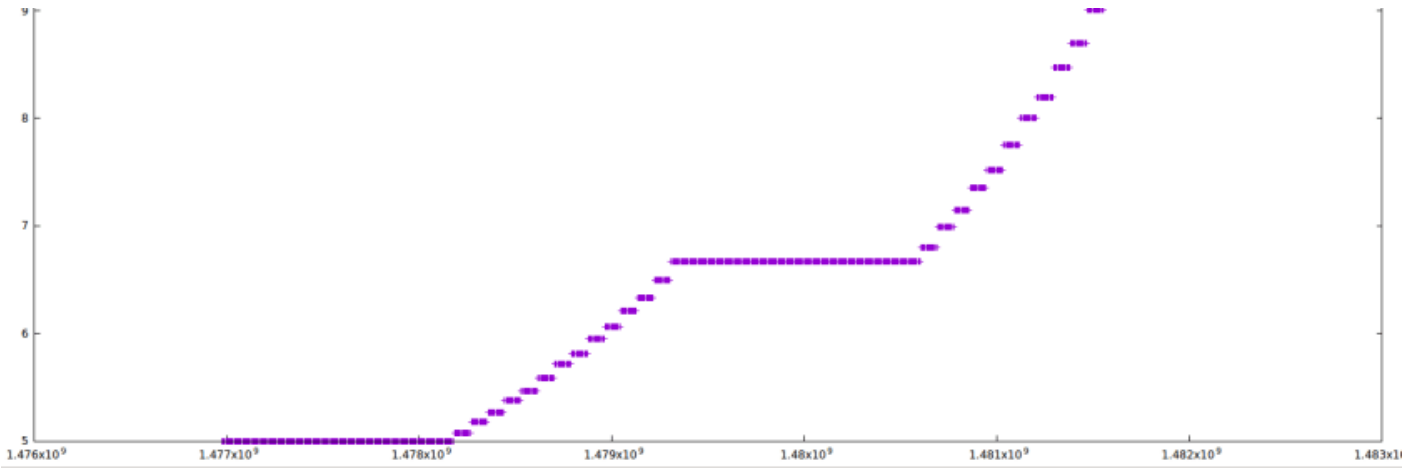
Price function check

We wrote a small test script to calculate price evolution in time, and check the intended result. We realized that the milestone (p1, p2, etc.) transitions are correct, but the price curve may not be exactly as intended, again, due to integer division.

Here's the intended price curve, from the HackerGold crowdsale page:



The price curve I reproduced is roughly the same, but instead of the continuous straight lines, we can see a stepped increase in price:



Consider reordering arithmetic operations in `getPrice` to achieve a smoother price curve.

You can find the tester code here

Warnings

Usage of magic constants

There are several magic constants in the contract code. Some examples are:

1. https://github.com/ether-camp/virtual-accelerator/blob/2529ffe5efd5294b44f1bc89dc9a4721a7b16409/contracts/HackerGold.sol#L88-L90
2. https://github.com/ether-camp/virtual-accelerator/blob/2529ffe5efd5294b44f1bc89dc9a4721a7b16409/contracts/HackerGold.sol#L116-L133

Use of magic constants reduces code readability and makes it harder to understand code intention. We recommend extracting magic constants into contract constants.

EDIT: EtherCamp fixed this problem in this commit

Remove unnecessary code

The `uint totalValue` variable may be unnecessary. Consider using `wallet.balance` in its place unless funds in the multisig address will be moved (and thus balance changed) before p6.

Additional Information and notes

- There are several typos in the comment for TokenInterface.sol the allowance function. See: mouch, permited (twice). EDIT: EtherCamp fixed this problem in this commit
- Use of `send` is always risky and should be analyzed in detail. Only one occurrence found in line 96 of HackerGold.sol
- Always check send return value: OK.
- Always call send at the end of the function: OK.
- Favor pull payments over push payments: Warning. No problems with push payment used, because `wallet` will be controlled by EtherCamp. Bear in mind that if that send fails for any reason, the whole `createHKG` call will fail.

Conclusions

No severe security issues were found. Some changes were recommended to follow best practices and reduce potential attack surface.

Note: EtherCamp followed our recommendations and fixed the code, as you can see in their GitHub repo

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to HKG token contract. We have not reviewed the related Virtual Accelerator project. The above should not be construed as investment advice or an offering of HKG. For general information about smart contract security, check out our thoughtshere

Related Posts

Beefy

BRUSHFAM
OpenBrush Contracts

Linea

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits