





360 核心安全技术博客

 主页 Home

 归档 Archive

 分类 Category

 关于 About



# EOS Asset Multiplication Integer Overflow Vulnerability

08月08, 2018

Yuki Chen of Qihoo

360 Vulcan Team

## Description

The asset structure is defined in EOS's system header file, it can be used to

define the amount of some tokens (such as the official EOS token or some custom tokens defined by user). Recently we discovered a bug in asset's multiplication operator (operator `*=`) which makes the integer overflow check in the function to have no effect. If a developer uses asset multiplication in his EOS smart contract, he may need to face the risk of integer overflow.

### 文章目录

- [Description](#)
- [Vulnerability Detail](#)
- [The result of the vulnerability](#)
- [The fix](#)
- [Our Suggestion](#)
- [Timeline](#)

## Vulnerability Detail

The problematic code exists in `contracts/eosiolib/asset.hpp`:

```
01. asset& operator*=( int64_t a ) {
02.   eosio_assert( a == 0 || (amount * a) / a == amount, "multiplication overflow or underflow" ); <
    == (1)
03.   eosio_assert( -max_amount <= amount, "multiplication underflow" ); <= (2)
04.   eosio_assert( amount <= max_amount, "multiplication overflow" ); <= (3)
05.   amount *= a;
06.   return *this;
07. }
```

As you can see from the above code, there are 3 integer overflow checks in this function. However,



360 核心安全技术博客

 主页 Home

 归档 Archive

 分类 Category

 关于 About

unfortunately, none of the 3 checks can really take effect. First let's take a look at the check (2) and (3). They are used to check whether the result of the multiplication is in a valid range  $[-\text{max\_amount}, \text{max\_amount}]$ . The problem here is that the checks are wrongly put before the multiplication code "amount = a". *The correct sequence is to put the checks behind the code "amount = a" in order to check the result of the multiplication operation.* The correct code should look like this:

```
01. amount *= a;
02. eosio_assert( -max_amount <= amount, "multiplication underflow" ); <= (2)
03. eosio_assert( amount <= max_amount, "multiplication overflow" ); <= (3)
```

Now let's have a look at the check (1). It is a very import check to ensure that: 1. The result does not cause a signed integer overflow (e.g.  $a > 0, b > 0, a * b < 0$ ) 2. The result does not overflow the 64-bits integer range

```
01. eosio_assert( a == 0 || (amount * a) / a == amount, "multiplication overflow or underflow" ); <= (1)
```

The bug here is not straight-forward, and is hard to notice by just looking at the C++ source code. But you know, in EOS, every smart contract written in C++ will be finally compiled into web assembly binary to be executed in the VM. Let's take a look at the result bytecode for the overflow check (1):

```
01. (call $eosio_assert
02. (i32.const 1) // always true
03. (i32.const 224) // "multiplication overflow or underflow\00")
04. )
```

The above bytecode is for the C++ source code:

```
01. eosio_assert( a == 0 || (amount * a) / a == amount, "multiplication overflow or underflow" ); <= (1)
```




360 核心安全技术博客

 主页 Home

 归档 Archive

 分类 Category

 关于 About

What a surprise! Because if we translate the bytecode back to c++ source code, the result will be:

```
01. eosio_assert(1, "multiplication overflow or underflow" );
```

Which means this asset will be always true, and the overflow check code just disappeared. Based on our experience, this might be the result of compiler optimization. So we checked the official script for compiling a contract (eosiocpp):





```
01. ($PRINT_CMDS; /root/opt/wasm/bin/clang -emit-llvm -O3 --std=c++14 --target=wasm32 -nostdinc \
```

We can see it uses clang to compile the source code and use O3 as the optimization level by default. We disabled the compiler optimization (using -O0) and then compiled the same code again, this time the result bytecode was:

```
01. (block $label$0
02.   (block $label$1
03.     (br_if $label$1
04.       (i64.eqz
05.         (get_local $1)
06.       )
07.     )
08.     (set_local $3
09.       (i64.eq
10.         (i64.div_s
11.           (i64.mul
12.             (tee_local $1
13.               (i64.load
14.                 (get_local $0)
15.               )
16.             )
17.             (tee_local $2
18.               (i64.load
19.                 (get_local $4)
20.               )
21.             )
22.             )
23.             (get_local $2)
24.           )
25.           (get_local $1)
26.         )
27.       )
28.     (br $label$0)
29.   )
```



360 核心安全技术博客

-  主页 Home
-  归档 Archive
-  分类 Category
-  关于 About



```

30. (set_local $3
31. (i32.const 1)
32. )
33. )
34. (call $eosio_assert
35. (get_local $3) // condition based on "a == 0 ||
    (amount * a) / a == amount"
36. (i32.const 192) // "multiplication overflow or underflow\00")

```

You can see this time the overflow check remained in the result bytecode, so we can confirm that this problem is caused by compiler optimization. So why this happens? Because in the below overflow check code, the variable `|amount|` and `|a|` are both signed integer (`int64_t`):

```

01. eosio_assert( a == 0 || (amount * a) / a == amount, "multiplication overflow or underflow" );

```

In C/C++ standard, the overflow of signed integer is an “undefined behavior” (UB). When an UB occurs, the execution result of the code is not defined. Some compilers such as clang will not consider the situation of undefined behaviors when doing optimization. In one word, in this case, when doing optimization, clang will not consider the possibility of integer overflow in the following statement:

```

01. (amount * a)

```

And if there is no need to consider the situation of integer overflow here, the following statement will be always true:

```

01. a == 0 || (amount * a) / a == amount

```

Thus it will be just optimized out when you enable the compiler optimization.

## The result of the vulnerability

Since all the 3 overflow checks in the function are invalid, you will face all possible types of integer overflow when using asset multiplication, such as:



360 核心安全技术博客

 主页 Home

 归档 Archive

 分类 Category

 关于 About

1.  $a > 0, b > 0, a * b < 0$
2.  $a > 0, b > 0, a * b < a$
3.  $a * b > \text{max\_amount}$
4.  $a * b < -\text{max\_amount}$

## The fix

In August 7th, EOS released the official fix for this issue:

<https://github.com/EOSIO/eos/commit/b7b34e5b794e323cdc306ca2764973e1ee0d168f>

## Our Suggestion

For EOS smart contract developers, if you have used the asset multiplication in your smart contract, we suggest you to rebuild your contract after applying the official fix for this bug.

## Timeline

2018-7-26: 360 Vulcan discovered the overflow issue in our code auditing. 2018-7-27: The issue was submitted to the vendor. 2018-8-07: EOS released the fix for the issue.

本文链接: <http://blogs.360.cn/post/eos-asset-multiplication-integer-overflow-vulnerability.html>

-- EOF --

作者 [admin001](#) 发表于 2018-08-08 06:51:08, 添加在分类 [Blockchain](#) [Threat Intelligence](#) [Vulnerability Analysis](#) 下, 最后修改于 2018-08-24 08:35:55

分享到: [新浪微博](#)[微信](#)[Twitter](#)[印象笔记](#)[QQ好友](#)[有道云笔记](#)

---


« 2018年7月勒索病毒疫情分析  
EOS官方API中Asset结构体的乘法运算溢出漏洞描述 »


---



360 核心安全技术博客

## Comments

 主页 Home

 归档 Archive

 分类 Category

 关于 About

---

© 2020 - 360 核心安全技术博客 - [blogs.360.cn](https://blogs.360.cn)

Powered by [ThinkJS](#) & [FireKylin 1.2.8](#)