随机数之殇—新型随机数攻击手法细节分析

by yudan@慢雾安全团队，Jerry@EOS Live 钱包

本次攻击及分析基于 EOS 1.8 以前版本，在新版本中未做测试

0x01 事件回顾

根据慢雾区情报，EOS DApp EOSPlay 中的 DICE 游戏于9月14日晚遭受新型随机数攻击，损失金额高达数万 EOS。经过慢雾安全团队的分析，发现攻击者（账号：muma******mm）在此次攻击过程中利用 EOS 中的经济模型的缺陷，使用了一种新型的随机数攻击手法对项目方进行攻击。

在开始详细的细节剖析之前，需要先了解一些技术背景作为铺垫。

0x02 技术背景

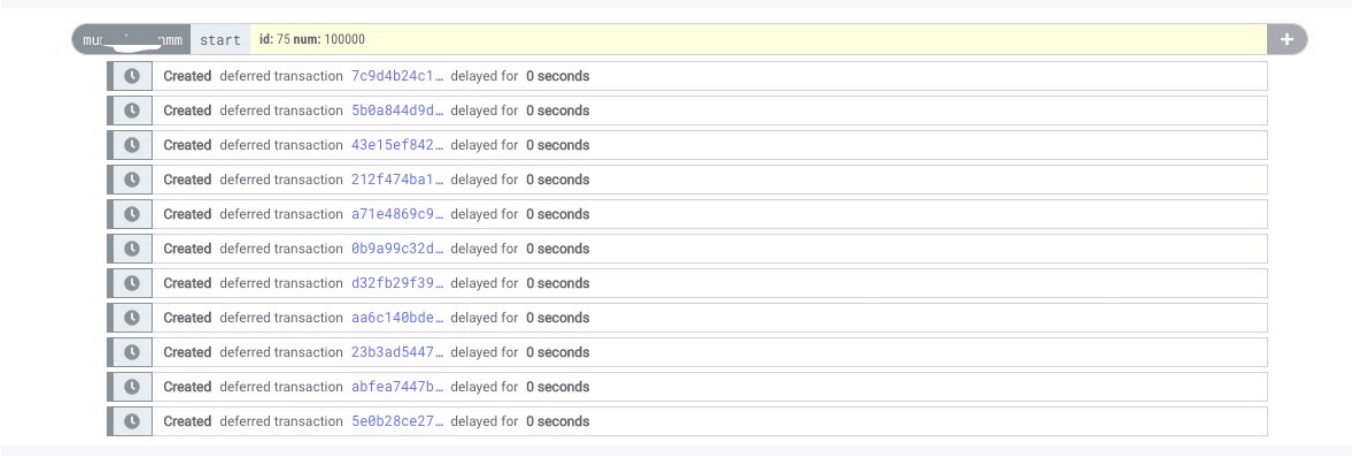
我们知道，在 EOS 系统中，资源是保证整个 EOS 系统安全运行的关键，在 EOS 上发出的每一笔交易都要消耗对应的资源，如 CPU、NET 或 RAM，而这些资源都是需要一定量的 EOS 作为抵押换取的，如果抵押的 EOS 的数量不够，则无法换取足够的资源发起交易。而本次的攻击行为则是利用了 EOS 资源里面的 CPU 资源模型进行攻击。我们知道，在 EOS 上 CPU 是一个十分重要，也是消耗最多的资源。在 EOS DApp 活跃的时候，会经常出现 CPU 不足的情况，为此，还催生出许多做 CPU 抵押的第三方 DApp，缓解资源使用紧张的问题。那么 CPU 资源的份额具体是怎么计算的呢?
参考 MEETONE 的文章(https://github.com/meet-one/documentation/blob/master/docs/EOSIO-CPU.md) 我们可以看到 CPU 的计算方式为

```
max_block_cpu_usage * (account_cpu_usage_average_window_ms / block_interval_ms) *
your_staked_cpu_count / total_cpu_count
```

其中，max_block_cpu_usage * (account_cpu_usage_average_window_ms / block_interval_ms) 是一个固定的变量，也就是常量。那么真正影响帐号的 CPU 使用情况的是后面的公式 your_staked_cpu_count / total_cpu_count，这里的意思就是，你抵押的 CPU 总量除以总的 CPU 抵押总量，那么，如果总量变多了，如果你的抵押数量没跟上去，能用的 CPU 就会变得越来越少。而 CPU 不足，则无法正常地发起交易。在了解完 CPU 的相关知识后，就可以开始相关的分析了。

0x03 攻击细节剖析

本次的攻击帐号为 muma******mm,通过分析该帐号行为，发现该帐号通过 start 操作发送了大量的 defer 交易，这些 defer 操作里面又发起了另一个 start 操作，然后无限循环下去。



由于这些大量的交易充斥着区块，为分析工作带来了困难，所以首先要对这些交易进行筛选。查看攻击者帐号的接收开奖通知的交易，通过查询游戏的开奖区块，我们发现一个问题，就是在多个开奖区块内只有系统的 onblock 一笔交易，说明这个区块为空块。很明显，这是因为 CPU 价格的太高导致交易无法发出而出现的情况。而本次受攻击的 EOS Play 采用的是使用未来区块哈希进行开奖。

那么攻击者为什么要选择这样做呢？我们来分析下区块哈希的计算方法

```
digest_type block_header::digest()const
{
    return digest_type::hash(*this);
}

uint32_t block_header::num_from_id(const block_id_type& id)
{
    return fc::endian_reverse_u32(id._hash[0]);
}

block_id_type block_header::id()const
{
    // Do not include signed_block_header attributes in id, specifically exclude producer_signature.
    block_id_type result = digest(); //fc::sha256::hash(*static_cast<const block_header*>(this));
    result._hash[0] &= 0xffffffff00000000;      Daniel Larimer, a year ago • adding missing files
    result._hash[0] += fc::endian_reverse_u32(block_num()); // store the block num in the ID, 160 bits is plenty for the hash
    return result;
}
```

通过上图我们可以看到，区块哈希的计算最终调用了 digest() 方法，而 digest() 方法的实现为对 block_header 结构体本身进行哈希。我们继续看 block_header 的定义

```
struct block_header
{
   block_timestamp_type             timestamp;
   account_name                     producer;

   /**
    *  By signing this block this producer is confirming blocks [block_num() - confirmed, blocknum())
    *  as being the best blocks for that range and that he has not signed any other
    *  statements that would contradict.
    *
    *  No producer should sign a block with overlapping ranges or it is proof of byzantine
    *  behavior. When producing a block a producer is always confirming at least the block he
    *  is building off of.  A producer cannot confirm "this" block, only prior blocks.
    */
   uint16_t                         confirmed = 1;

   block_id_type                    previous;

   checksum256_type                 transaction_mroot; /// mroot of cycles_summary
   checksum256_type                 action_mroot; /// mroot of all delivered action receipts


   /** The producer schedule version that should validate this block, this is used to
    * indicate that the prior block which included new_producers->version has been marked      Daniel Larimer, a
    * irreversible and that it the new producer schedule takes effect this block.
    */
   uint32_t                          schedule_version = 0;
   optional<producer_schedule_type>  new_producers;
   extensions_type                   header_extensions;


   digest_type       digest()const;
   block_id_type     id() const;
   uint32_t          block_num() const { return num_from_id(previous) + 1; }
   static uint32_t   num_from_id(const block_id_type& id);
};
```

可以看到 block_header 结构体的变量分别有

1、confrimed
2、previous
3、transaction_mroot
4、action_mroot
5、schedule_version
6、new_producers
7、header_extensions

其代表的含义分别为

1、每一轮开始前需要确认的上一轮生产的区块数
2、之前区块的区块哈希
3、区块 transaction 的默克尔根哈希
4、区块 action 的默克尔根哈希
5、节点出块顺序表版本，这个在一段时间内是不会改变的
6、新的出块节点，可为空
7、拓展，1.8 之前这个变量为空

到了这里，我们可以清楚的看到，除了 action_mroot 和 transaction_mroot 外，所有的其他变量都是可以轻松获取的，然后我们继续深入看下 transaction_mroot 和 action_mroot 的计算方法。

```
void set_action_merkle() {
   vector<digest_type> action_digests;      Daniel Larimer, a year ago • progress
   action_digests.reserve( pending->_actions.size() );
   for( const auto& a : pending->_actions )
      action_digests.emplace_back( a.digest() );

   pending->_pending_block_state->header.action_mroot = merkle( move(action_digests) );
}

void set_trx_merkle() {
   vector<digest_type> trx_digests;
   const auto& trxs = pending->_pending_block_state->block->transactions;
   trx_digests.reserve( trxs.size() );
   for( const auto& a : trxs )
      trx_digests.emplace_back( a.digest() );

   pending->_pending_block_state->header.transaction_mroot = merkle( move(trx_digests) );
}
```

可以看到 transaction_mroot 和 action_mroot 是分别对 pending_block 内的 action 和 transaction 的总的 digests 进行一个 merkle 运算最终得到一个 transaction_mroot 和 action_mroot。我们继续追踪各自的 digest 的生成方式

```
struct transaction_receipt : public transaction_receipt_header {

   transaction_receipt():transaction_receipt_header(){}
   explicit transaction_receipt( const transaction_id_type& tid ):transaction_receipt_header(executed),trx(tid){}
   explicit transaction_receipt( const packed_transaction& ptrx ):transaction_receipt_header(executed),trx(ptrx){}

   fc::static_variant<transaction_id_type, packed_transaction> trx;

   digest_type digest()const {      Daniel Larimer, a year ago • progress
      digest_type::encoder enc;
      fc::raw::pack( enc, status );
      fc::raw::pack( enc, cpu_usage_us );
      fc::raw::pack( enc, net_usage_words );
      if( trx.contains<transaction_id_type>() )
         fc::raw::pack( enc, trx.get<transaction_id_type>() );
      else
         fc::raw::pack( enc, trx.get<packed_transaction>().packed_digest() );
      return enc.result();
   }
};
```

```
   struct action_receipt {
      account_name                    receiver;
      digest_type                     act_digest;      arhag, a year ago • reorganize header files of chain library
      uint64_t                        global_sequence = 0; ///< total number of actions dispatched since genesis
      uint64_t                        recv_sequence   = 0; ///< total number of actions with this receiver since genesis
      flat_map<account_name,uint64_t> auth_sequence;
      fc::unsigned_int                code_sequence = 0; ///< total number of setcodes
      fc::unsigned_int                abi_sequence  = 0; ///< total number of setabis

      digest_type digest()const { return digest_type::hash(*this); }
   };

} }  /// namespace eosio::chain
```

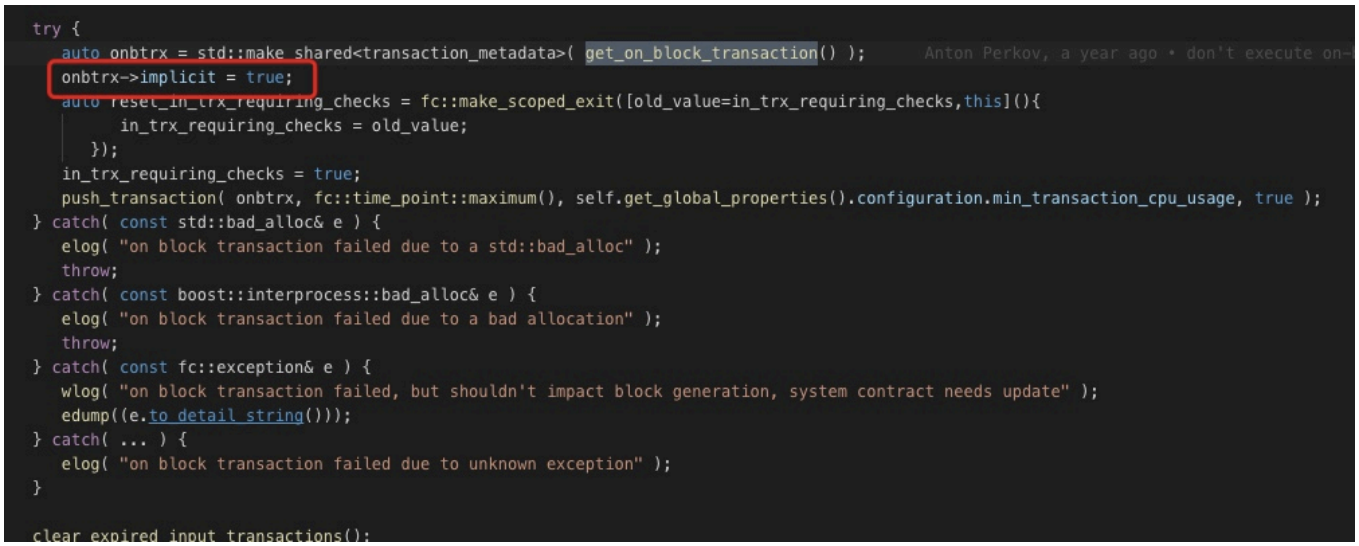
以上两条分别是 transaction digest 和 action digest 的生成方式，我们可以看到，transsaction digest 的计算使用到了 cpu 和 net 作为因子，这两个因子取决于交易执行时 bp 的节点状态，虽然透明但较难准确预测，而 action digest 没有使用到这两个变量，可以轻松的算出来。那么分析到这里，我们可以确定的是，要预测一个区块的哈希，需要引入不确定的 cpu 和 net 的使用量，那么，有没有办法不将这两个因子引入到计算当中呢?

答案是必然的

我们把目光聚集到项目方开奖区块的唯一一个交易 onblock 中，首先我们先看看 onblock 的生成定义

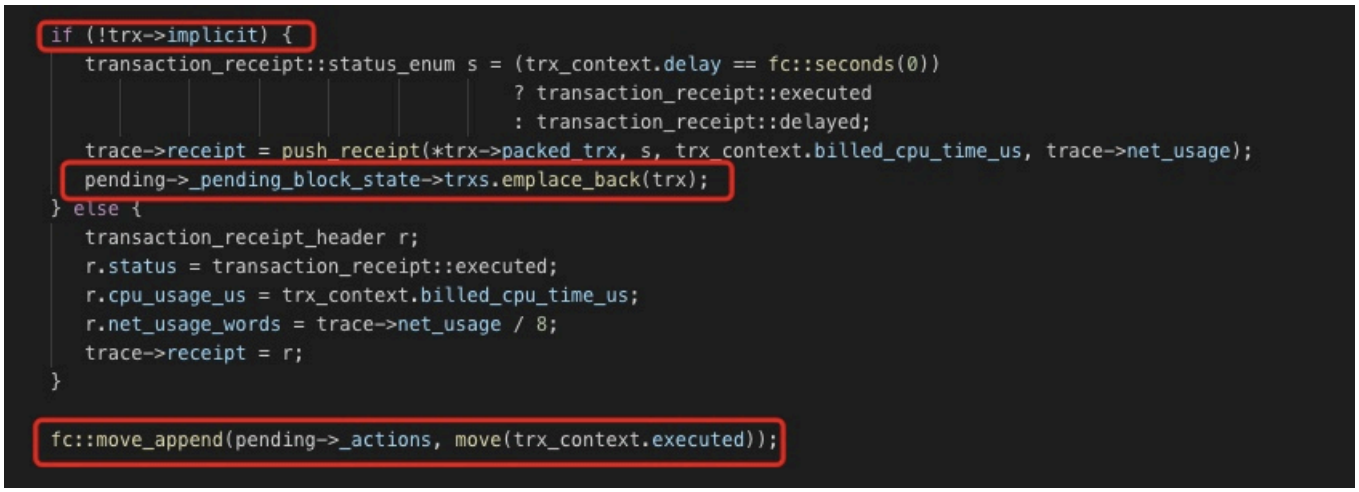
```
signed_transaction get_on_block_transaction()      Daniel Larimer, a year ago • progress
{
   action on_block_act;
   on_block_act.account = config::system_account_name;
   on_block_act.name = N(onblock);
   on_block_act.authorization = vector<permission_level>{{config::system_account_name, config::active_name}};
   on_block_act.data = fc::raw::pack(self.head_block_header());

   signed_transaction trx;
   trx.actions.emplace_back(std::move(on_block_act));
   trx.set_reference_block(self.head_block_id());
   trx.expiration = self.pending_block_time() + fc::microseconds(999'999); // Round up to nearest second to avoid appearing expired
   return trx;
}

}; /// controller_impl
```

以上是构造 onblock 操作的一些定义，可以看到，onblock 操作的 data 字段为 header_block_header，就是当前的区块头信息，即相对于当前 pending 区块上一个块的区块信息，接下来我们追踪该函数的调用过程。



```
try {
   auto onbtrx = std::make_shared<transaction_metadata>( get_on_block_transaction() );      Anton Perkov, a year ago • don't execute on-
   onbtrx->implicit = true;
   auto reset_in_trx_requiring_checks = fc::make_scoped_exit([old_value=in_trx_requiring_checks,this](){
         in_trx_requiring_checks = old_value;
      });
   in_trx_requiring_checks = true;
   push_transaction( onbtrx, fc::time_point::maximum(), self.get_global_properties().configuration.min_transaction_cpu_usage, true );
} catch( const std::bad_alloc& e ) {
   elog( "on block transaction failed due to a std::bad_alloc" );
   throw;
} catch( const boost::interprocess::bad_alloc& e ) {
   elog( "on block transaction failed due to a bad allocation" );
   throw;
} catch( const fc::exception& e ) {
   wlog( "on block transaction failed, but shouldn't impact block generation, system contract needs update" );
   edump((e.to_detail_string()));
} catch( ... ) {
   elog( "on block transaction failed due to unknown exception" );
}

clear_expired_input_transactions();
```

发现一个点，就是这里会把 onblock 交易的 implict 设置为 true，这样设置有什么用效果呢? 我们接下来看下交易的打包过程



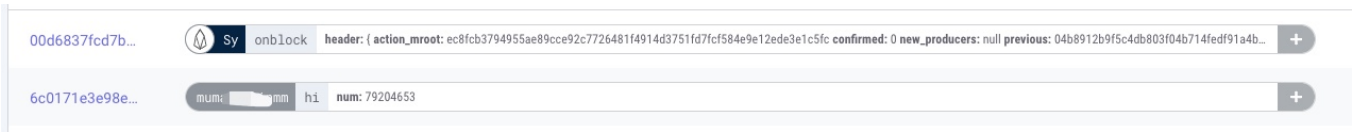
```
if (!trx->implicit) {
   transaction_receipt::status_enum s = (trx_context.delay == fc::seconds(0))
                                        ? transaction_receipt::executed
                                        : transaction_receipt::delayed;
   trace->receipt = push_receipt(*trx->packed_trx, s, trx_context.billed_cpu_time_us, trace->net_usage);
   pending->_pending_block_state->trxs.emplace_back(trx);
} else {
   transaction_receipt_header r;
   r.status = transaction_receipt::executed;
   r.cpu_usage_us = trx_context.billed_cpu_time_us;
   r.net_usage_words = trace->net_usage / 8;
   trace->receipt = r;
}

fc::move_append(pending->_actions, move(trx_context.executed));
```

这里看到，由于在区块开始的时候，implict 属性被设置为 true，导致交易在被打包时 onblock 交易不会进入 pending 区块的 transaction 序列中，但是却进入了 pending 区块的 action 序列。也就是说，项目方的开奖区块的 action 序列不为空，而 transaction 序列为空。所以说开奖区块的 transaction_mroot 必然为 0，这样，就避开了计算 transaction_mroot 时引入的 cpu 和 net 这些不确定因素，只计算 action_mroot 即可。而 action_mroot 的计算只需要对 onblock action 本身进行计算即可，且 onblock 的 data 是可以拿到的，即当前区块头信息，也就是相对于当前 pending 区块的上一个区块的信息。那么到了这里，攻击者已经完全可以拿到可计算开奖区块哈希 的所

有信息，计算出开奖区块哈希 就不是什么难事了。

0x04 攻击情况总结

在完成攻击细节剖析后，我们可以知道，攻击者在下注时，并不知道开奖区块的区块哈希，因为该哈希和开奖区块的前一个块相关，所以攻击者能做的是在开奖区块的前一个块对开奖区块的信息进行计算。那万一计算不对怎么办呢？关于这点，我们发现一个现象，当区块计算结果和预期有误差时，攻击者会在开奖区块发送一笔 hi 交易来企图改变结果，但是由于新引入的 hi 交易会引入 cpu 和 net 使用量这些变量，所以引入 hi 交易并不能确保结果可控，但是已经将赢的概率大大提升了，相当于重新开奖。



总结下来攻击者的攻击手法为：
1、通过 REX 购买大量的 CPU，导致正常交易无法发出
2、在假设开奖区块为空块的情况下对开奖区块的哈希进行计算
3、当发现计算结果不对时，向开奖区块内发送一笔 hi 交易，尝试改变结果。

0x05 预防方案

EOS Play 由于使用了未来区块哈希作为开奖因子，导致开奖结果被预测从而导致攻击发生。这次的例子再一次的告诉我们，所有使用区块信息进行开奖的开奖方案都是不成熟的，都存在被预测的可能。特别是 REX 出现以后，抬高 CPU 价格比以往变得更为容易。

慢雾安全团队在此提醒，DApp 项目方在使用随机数时，特别是涉及到关键操作的随机数，应尽量采用更为安全的线下随机数生成方案，降低随机数被攻击的风险。除此之外，同时还可以接入慢雾安全团队孵化的 EOS 智能合约防火墙 FireWall.X（https://firewallx.io），为合约安全保驾护航。