RESEARCH BULLETIN MRL-0002

Counterfeiting via Merkle Tree Exploits within Virtual Currencies Employing the CryptoNote Protocol

12 September 2014

Jan Macheta[1], Sarang Noether[2*], Surae Noether[2] and Javier Smooth[2]

*Correspondence: lab@monero.cc
[2] Monero Research Lab
(Full list of author information on last page)

Abstract
On 4 September 2014, an unusual and novel attack was executed against the Monero cryptocurrency network. This attack partitioned the network into two distinct subsets which refused to accept the legitimacy of the other subset. This had myriad effects, not all of which are yet known. The attacker had a short window of time during which a sort of counterfeiting could occur, for example. This research bulletin describes deficiencies in the CryptoNote reference code allowing for this attack, describes the solution initially put forth by Rafal Freeman from Tigusoft.pl and subsequently by the CryptoNote team, describes the current fix in the Monero code base, and elaborates upon exactly what the offending blcok did to the network. This research bulletin has not undergone peer review, and reflects only the results of internal investigation.

1 Introduction

On 4 September 2014, a novel attack was executed against the Monero cryptocurrency network, leading to a never-before-seen phenomenon in the network. The attacker must have had extensive knowledge of the Monero code, had good knowledge of Merkle Trees, and basic knowledge of cryptographic hash functions.

The Monero code was forked from the Bytecoin reference code in April 2014 before the CryptoNote reference code was released. The Bytecoin code appeared to be somewhat obfuscated and poorly commented, so any given segment of code must necessarily be considered with skepticism. Hence, any attacker with an extensive knowledge of the Monero code also, presumably, has an extensive knowledge of the Bytecoin and CryptoNote reference codes.

A Merkle Tree is a data structure in which every non-leaf node in the tree gains a label, and the label is the hash of the child nodes[3]. That is to say, to build a Merkle Tree, we take some blocks of data (transactions), and we hash them. Then we hash those, two at a time. Then we hash those, two at a time. And we repeat this process until we are done. You can think of a Merkle Tree as a football bracket of cryptographic hashes. A natural question arises: what if we do not have a nice, even, power-of-two, divisible number of transactions? This is where the exploit came in, as we will see later.

CryptoNote currencies, and indeed, most cryptocurrencies, use Merkle Trees to construct the hash of a block of transactions (which is subsequently packed into the header of the block). The attacker noticed that a mis-implementation of a commonly used power-of-two rounding algorithm could be exploited while computing the hash of a block in order to generate two distinct blocks with the same hash. This should be all but impossible, of course, as collisions of cryptographic hash functions are rare. When discussing the rarity of these collisions, comparisons like "how many fundamental particles exist in the universe?" start to pop up, and usually those numbers are not big enough to describe collision rarity. In general, hash collisions typically imply mis-implementation rather than a failure of the hash functions, presuming we are using a good hash function [2].

To the authors' best knowledge, this attack has only occurred once. Although Monero was the target, any coin utilizing CryptoNote reference code from before 4 September 2014 suffered the mis-implementations allowing for such exploitation, with two exceptions. Fantomcoin and Moneta Verde apparently fixed the relevant exploit in secret several months ago. Again, only to the authors' best knowledge, all popular CryptoNote currencies have implemented the changes described by R. Freeman and the CryptoNote team, although we have not performed an extensive code review. The purpose of this document is to describe the section of code that allowed this attack and to elaborate on the attack's effects. For a slightly different take on the attack, see, for example, [1].

2 Exploitation of the CryptoNote Reference Code

Throughout this section, we abuse terminology slightly when we say *CryptoNote reference code*, as Monero was forked from ByteCoin before the CryptoNote reference code was released; we do not fear confusion in the reader. The exploitation of the CryptoNote reference code can be understood using the football bracket analogy described before: what happens when the data we wish to hash does not come in nice, even powers of two? We simply take any *overflow* above a power of two and play a smaller, first-round bracket, where all other, pre-overflow data gets a *bye*[1] to the next round. Of course, this just kicks the can down the road to the next smallest power of two, but eventually this process will stop at two teams and we can proceed with our bracket like usual. In order to perform this strategy, we wish to first figure out how many teams we really do have playing.

Of course, in order to do this, the *size, length, or measure* of the current data set needs to be rounded down to the greatest power of two strictly less than the current data size; all data *above* this amount (in terms of indices) needs to be folded into the data below this amount with cryptographic hash functions. This is where the exploit came in. The cutoff index splitting *above* versus *below* was being miscomputed. The attacker noticed that this would allow some data to be ignored when computing the hash of a block, and so two distinct blocks could obtain the same hash. This was not due to any sort of failure of the hashing functions used, but, as described above, a failure in a measurement of the size of the "real" data to be used in construction of the Merkle Tree.

The following code in `src/crypto/tree-hash.c` was exploited:

[1]Not byte.

```
src/crypto/tree-hash.c:

size_t cnt = count - 1;
for (i = 1; i < sizeof(size_t); i <<= 1) {
     cnt |= cnt >> i;
}
cnt &= ~(cnt >> 1);
```

The goal of this code, a version of a commonly used algorithm, is to round the number `count` to largest power of two that is *strictly* smaller than `count`, setting this value as `cnt`[4].

Input	Output
100	64
255	128
512	256
513	512
1205	1024

To understand how this segment of code works, we'll first consider a toy example and then compare it to how the algorithm actually works. Suppose we want to determine the largest power of two *strictly* less than the integer 1205. Define x to be the binary number formed by decrementing the integer 1205 and setting every bit to the right of the most significant bit to one:

$$1204_{10} = 10010110100_2 \Rightarrow x = 11111111111_2$$

We next set y to be the binary number formed by performing a single right-shift on x and taking its negative:

$$y = \overline{x \gg 1} = \overline{01111111111}_2 = 10000000000_2$$

The result is the number xy, using the binary *AND* operation:

$$xy = 11111111111_2 \cdot 10000000000_2 = 10000000000_2 = 1024_{10}$$

We now compare this toy example to the algorithm in the above-referenced block of code from `src/crypto/tree-hash.c`. Assume, as is true for most modern architectures, that `sizeof(size_t) = 8`; in this case, we have $i \in \{1, 2, 4\}$, so the loop executes three times. Consider the operation of the code from `src/crypto/tree-hash.c` using the sample data `count` $= 513_{10}$:

$$\begin{aligned}
\texttt{cnt} &= 513 - 1 = 512_{10} = 1000000000_2 \\
i = 1 : \texttt{cnt} &= 1000000000_2 + 0100000000_2 = 1100000000_2 \\
i = 2 : \texttt{cnt} &= 1100000000_2 + 0011000000_2 = 1111000000_2 \\
i = 4 : \texttt{cnt} &= 1111000000_2 + 0000111100_2 = 1111111100_2
\end{aligned}$$

Observe that because the loop executes only three times, we did not change the state of the two least significant bits. The algorithm proceeds:

$$\texttt{cnt} = 1111111100_2 \cdot 100000001_2 = 1000000000_2 = 512_{10}$$

We obtain the correct result. Suppose, however, that we instead use the sample data $\texttt{count} = 514_{10}$:

$$\begin{aligned}
\texttt{cnt} &= 514 - 1 = 513_{10} = 1000000001_2 \\
i = 1 : \texttt{cnt} &= 1000000001_2 + 0100000000_2 = 1100000001_2 \\
i = 2 : \texttt{cnt} &= 1100000001_2 + 0011000000_2 = 1111000001_2 \\
i = 4 : \texttt{cnt} &= 1111000001_2 + 0000111100_2 = 1111111101_2
\end{aligned}$$

Similar to before, the last two bits remain unchanged. The algorithm proceeds:

$$\texttt{cnt} = 1111111101_2 \cdot 1000000001_2 = 1000000001_2 = 513_{10}$$

This is clearly incorrect, as we expect the algorithm to output the integer 512. This mis-implementation of the previously described rounding algorithm was the root of the exploitation of the CryptoNote reference code. Let us consider the ramifications of the mis-implementation of this rounding down by considering the affected code further down the line:

```
src/crypto/tree-hash.c:

47  char (*ints)[HASH_SIZE];
52  ints = alloca(cnt * HASH_SIZE);
53  memcpy(ints, hashes, (2 * cnt - count) * HASH_SIZE);
54  for (i = 2 * cnt - count, j = 2 * cnt - count; j < cnt; i += 2, ++
    j) {
55    cn_fast_hash(hashes[i], 64, ints[j]);
56  }
57  assert(i == count);
58  while (cnt > 2) {
59    cnt >>= 1;
60    for (i = 0, j = 0; j < cnt; i += 2, ++j) {
61      cn_fast_hash(ints[i], 64, ints[j]);
62    }
63  }
64  cn_fast_hash(ints[0], 64, root_hash);
```

In this code block, lines 47 through 56 perform the following tasks. We allocate an array of hashes to `ints`. We take *what we think* is supposed to be the data from the original `hashes` that has not yet overflowed our desired power-of-two structure, and we use `memcpy` to copy the data into `ints`. However, the quantity $2{\cdot}\texttt{cnt}{-}\texttt{count}$ is, as previously described, an overestimate of this amount of data. For example, the known exploit described above will round down the number 514 to 513, not 512. Hence, setting $\texttt{cnt}{=}\ 513$ and $\texttt{count}{=}\ 514$, we see that we copy $2{\cdot}\texttt{cnt}{-}\texttt{count}{=}\ 512$ elements, filling all of the array that is used in subsequent hashing rounds. However, we should have computed $\texttt{cnt}{=}\ 512$, copying $2{\cdot}\texttt{cnt}{-}\texttt{count}{=}\ 510$ pieces of data. Thus, the bug previously reported leads to two pieces of data being wholly ignored. That is to say, the first $2{\cdot}\texttt{cnt}{-}\texttt{count}$ hashes should receive a *bye* in the sports analogy, advancing to the next round without actually playing any games (or rather, being hashed with any other data). Since `cnt` is being overestimated, too much data is being passed forth without being properly handled, leaving the remaining data unused. Of course, we still hash the remaining data together, but these hashes are not used.

Note the stunning degree of technicality required to find and exploit this bug. Someone out there is very, very, very smart to not only find, but also to exploit, this deep bug. They know algorithms extremely well, and they appear to be malicious.

3 Fixes Applied

On 4 September 2014, the same date as the attack, the first solution to the exploit was publicly announced by Rafal Freeman from `Tigusoft.pl`[2]. The solution is not a unique one, as the CryptoNote reference code implements a different solution[3]. The CryptoNote solution establishes a "quick fix" in `src/crypto/tree-hash.c`, by using the ending condition 8·`sizeof(size_t)`, causing their loop to iterate through all bits in `cnt`. However, without adequate checks on the size of `cnt`, it may be possible to choose a suitably large value to conduct the same attack. Furthermore, using magic numbers is, simply, a generally frowned-upon practice in computer science and cryptographic protocols. Then again, so is code obfuscation, and it is considered good form to comment your code, both of which the CryptoNote team is ostensibly guilty.

The Monero codebase, by contrast, establishes checks on the size of `cnt` and uses the following rounding algorithm:

```
src/crypto/tree-hash.c:

assert( count >= 3); // cases for 0,1,2 are handled elsewhere
size_t tmp = count - 1;
size_t jj = 1;
for (jj=1 ; tmp != 0 ; ++jj) {
  tmp /= 2;
}
size_t cnt = 1 << (jj-2);
assert( cnt > 0 );  assert( cnt >= count/2 ); assert( cnt <= count
 );
assert( ispowerof2_size_t( cnt ));
return cnt;
```

This algorithm determines the number of powers of two less than `cnt` and performs an appropriate bitshift to recover the correct power of two. This algorithm does not suffer from the same size limitations as the CryptoNote fix.

4 Effects of the Attack

Once two blocks with the same hash were published nearly-simultaneously, part of the network had one block and the rest of the network had the other block (ignoring nodes that had not yet received the block). Simply checking transaction hashes on the opposing half of the network would cause a fault. This partitioned the network into two distinct subsets which refused to accept the legitimacy of the other subset. For awhile, the network was split into two. A reader may be tempted to call this a fork in the blockchain. Be wary. Calling this a fork would be inappropriate. Indeed, a fork occurs when two competing chains of transactions in the block-tree, both of which are ostensibly legitimate, compete for network hash rate. Eventually, one of

[2]see Monero commit 2ef0aee81d20c002ed50d6dec4baceee1ac40b44
[3]see CryptoNote commit 6be8153a8bddf7be43aca1efb829ba719409787a

the two chains will win since the network uses the “longest chain” decision method proposed by Satoshi Nakamoto. Furthermore, although it’s extremely difficult, it’s theoretically possible that, after a time, the network switches from one chain to another if it grows long enough.

The two parts of the network refused to accept the legitimacy of the blockchain upon which the other part was operating. It was as if, suddenly, half the Monero network switched to a brand new CryptoNote coin’s blockchain. It still appeared to, say, currency exchanges or vendors, that they were both one network still working on Monero.

This had myriad effects, not all of which are yet known. For one thing, balances were essentially doubled. If you had 1.0 XMR before the attack, you now had 1.0 XMR on each version of the network during the doubling. One could fancifully interpret this as a counterfeiting scenario: the attacker published two blocks with the same hash, and now instead of one network, there are two and the attacker has double the balance. The attacker had a short window of time during which they could sell “counterfeit” Monero from the new network on an exchange such as Poloniex for Bitcoin and immediately withdraw. The attacker still has his original balance on the “old” network, of course. With the major exception of MintPal, most major exchanges dealing with Monero were able to freeze those transactions during the attack.

For currently unknown reasons, most major Monero mining pools ended up together on one side of the network; this could have simply been a result of demographic stochasticity as the network pulled itself apart, or it could be a deterministic result of the speed at which blocks propagate. Indeed, if two blocks with the same block hash hit the network simultaneously, and if one block is heard by a large pool before the other due to stochasticity, it is almost certain that block will be heard by more of the network than the other. On the other hand, if two blocks with the same block hash hit the network *nearly* simultaneously but randomness is excluded, the first block will be the dominant block.

No matter what, the much smaller side of the network, of course, had significantly lower hashing power, and began experiencing terrifying errors. The difficulty was inherited from before the split, as well. Due to the protocol within CryptoNote to ignore outliers in block duration while computing difficulty, it would take somewhere between three days and one week for the B network to adjust difficulty so as to compensate. Any node stuck on the B side of the network saw a dramatic drop in their profitability; this, as well as the error spam they have received, has caused most nodes to resync their blockchain, or to reboot their computer, or something along those lines. The B side has all but vanished at this point, but the effects of the Block 202612 attack still linger on the Monero network. The Monero development team released patch 0.8.8.3 on 6 September 2014 to ensure that this attack can never occur via this route again, to allow miners to identify the A side of the network (that is, the A-side version of block 202612), and to identify foreign nodes with the B-side version of block 202612.

There may be other, as-yet-unknown, longer-term effects of having block 202612 stored in the blockchain. There may be other, as-yet-unknown, longer-term effects of using the transactions validated in that block as obfuscating elements in new ring signatures, if the owners of those transactions (presumably the attackers) proceed to spend those transactions with zero mix-ins.

Author details
[1] Tigusoft.pl. [2] Monero Research Lab.

References

1. Werner Albert. *Monero Network Exploit Post-Mortem*, 2014 (accessed Sept 15, 2014). `https://forum.cryptonote.org/viewtopic.php?f=7&t=270`.
2. Ross Anderson. The classification of hash functions. 1993.
3. Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO'87*, pages 369–378. Springer, 1988.
4. John Viega and Matt Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. O'Reilly Media, Inc., 2003.