2010

Intrepidus Group, Inc.

By David Schuetz
Senior Consultant

NAILS IN THE CRYPT(3)

Abstract: Admit it, password cracking is fun. And the only thing more fun than breaking a single root password is breaking a whole file of passwords, most of which are for administrators. Unfortunately, even with modern systems, brute forcing an entire large file remains terribly slow. So some years back, folks came up with the idea of rainbow tables, and suddenly fast attacks on an entire user base became possible. But for some reason, nobody's ever extended rainbow tables to support old-school UNIX crypt(3) passwords.

Maybe it was just processor speed. Or maybe it was disk space. Or maybe it was statements like "You'd need a full set of rainbow tables for each salt." But nobody's done it, at least not openly. This paper outlines a simple method to do just that, extending rainbow tables backwards into crypt(3) territory.

INTREPIDUS GROUP
PROACTIVE SECURITY

Introduction

Password cracking can be a powerful tool, both for auditing and penetration testing. However, password cracking can also be slow. Trying to crack an entire database of hundreds or thousands (or hundreds of thousands) of hashes can be very, very slow. So the obvious solution is to simply store the results of your password crack attacks for future use – eventually building up a huge database of password hashes. Once you do that, all future cracking sessions will be NEARLY INSTANTANEOUS! Win!

There's just one drawback.

An eight-character password, taken only from letters and numbers (A-Z, a-z, 0-9), comes from a key space of (26 + 26 + 10)^8, or 62^8 possible passwords. If you have a database that stores each password, and the resulting (for example), Unix crypt(3) hash, you'd need:

62^8 keys * (8 bytes / key) * (11 bytes / hash)

or approximately

4.1 PETABYTES.

That's quite a bit of storage. This doesn't even include special characters like +, $, or !. Adding special characters in could push the number to well over 100 Petabytes.

Then, some time ago, some clever folks realized that you don't have to store every hash you compute. If you have a predictable procedure to walk through the key space, you can simply record key values in that process and re-compute the missing values on the fly. It's called a Time-Memory Tradeoff. The idea dates to a paper published in 1980 by Martin Hellman. In 2003, Philippe Oechslin expanded on Hellman's ideas, and Rainbow Tables were born.

Rainbow Tables allow for a very sparse storage of very large password hash databases. Today, there are many different programs for computing and using Rainbow Tables. There are distributed projects for generating large tables. There are sites dedicated to distributing pre-computed tables. Rainbow Tables exist for just about every popular password hash algorithm: LAN Manager (LM), NT LAN Manager (NTLM), MD5, SHA, even for keys used in WiFi WPA security. Only one major omission remains: there are still no Rainbow Tables for the grandfather of all the password hashes, the old-school Unix crypt(3) hash.

Why Not?

So why haven't Rainbow Table geeks built tables for crypt(3)? The most-cited reason is the salt.

As a protection against pre-computed attacks (like we're discussing here), the creators of the crypt(3) algorithm added a "salt." The salt is a pair of characters (taken from the character set A-Z, a-z, 0-9, ., and /), which alters the encryption algorithm used to create the password hash. This guarantees that a single password will produce vastly different hashes when the salt is changed. In order for the system to verify a password provided by the user, it needs to know the salt, so it's prepended to the password hash (turning the 11-character hash into a 13-character salt+hash), then stored in the password file.

So having the salt makes Rainbow Tables impossible, or so conventional wisdom would have us believe. Even as I was writing this, I saw a tweet that said "I'm telling you that crypt(3) passwords (the Unix DES kind) have salts. You can't rainbow-table them."

How bad does the salt make it? That depends on who you ask. A quick review of discussion in various Rainbow Table forums yields quotes like "Thousands of times more work at hundreds of times slower speeds," "Increases key space by billions, to billions of billions," and the most common "You need a rainbow table for every salt."

Making matters worse, the hash algorithm itself is much slower than today's algorithms. So for these reasons, nobody (that I've been able to find) has developed crypt(3) Rainbow Tables.

End of Story?

Not so fast. It's not that bad. At all. Yes, the salt effectively increases the size of the key space, by a factor of 4096 (which is 64^2, the number of characters in the salt character set, raised to the length of the salt). Yes, generating and managing 4096 different rainbow tables would be difficult. Difficult, but certainly not insurmountable.

There's another way to look at it, though. We're really just adding 2 characters to the password key space. Not even that, since the password itself can come from a much wider range of characters than the salt does. With the massive hard drives available today, people are already making and sharing Rainbow Tables for 9 or 10 character passwords. So the key space itself shouldn't be a reason not to try.

Also, computing power is getting cheap. Very cheap. The most powerful computers available to the casual password cracker today are incredibly fast, and take advantage of the massive horsepower developed for high-speed, high-resolution graphical applications. Those systems (graphical processing units, or GPUs) are even available for rent in "the cloud," so you don't even need to buy your cracking hardware any more.

How about some numbers? There are three key dates for this technology: Hellman's original paper (1980), Oechslin's practical application and the birth of Rainbow Tables (2003), and today (2010). So let's compare technology for those dates.

First, computing power. In 1980, the state-of-the-art was the Motorola 68000 CPU, used in workstations from HP, Silicon Graphics, and Sun (and of course, in the original Macintosh). In 2003, the Intel Pentium P4 was on the leading edge of performance, and today, that lead is arguably held by the Intel Core i7. The i7 is 15 times faster than the P4, and almost 150,000 times faster than the 68000. Storage has followed a similarly astronomical rise over the last 30 years, with today's top-end about 50 times the size of common drives from 2003, and vastly larger than 1980's technology.

	1980	2003	2010
Typical CPU	68000	P4	i7
Speed (MIPS)	1	9700	147600
Relative to 2003	0.0001	1	15 x

Increases in Processor Power

	1980	2003	2010
Typical Drive	5 MB	40 GB	2 TB
Relative to 2003	0.00012	1	50 x

Increases in Storage Capacity

What does this mean for crypt(3) tables? I recently measured the algorithm at about 9 times slower than LM or MD5 algorithms. But CPUs are 15 times faster than they were when people first started building Rainbow Tables for LM hashes. If that speed was good enough for LM, in 2003, then surely today's speeds (and storage capacities) are good enough for crypt(3)?

To better understand what this means in real-world usage, I estimated the size required to store a Rainbow Table set for 8-character passwords, consisting only of lowercase letters and numbers. The chain length (a measure of how sparse the resulting table is) was assumed to be 10,000 hashes. The calculations indicated the entire set would take about 17.3 Terabytes, or about 9 of today's 2 TB drives.

Hash	Bytes	Number of Top-end Drives		
		1980	2003	2010
NTLM	4.32 GB	885	<1	<< 1
Crypt(3)	17.2 TB	3.6 million	443	9

Approximate Storage Requirements for NTLM and Crypt(3) Rainbow Tables (8-characters, lowercase letters and numbers)

While that's still a tremendous amount of data, today you can purchase that amount of space for under $1000. That's not even considering the 3 TB drives which are already on the market. This cost would be trivial for a dedicated InfoSec corporation (or criminal enterprise). Of course, this is a small character space, but I'm just showing a starting point. Some rough calculations for other key spaces follow:

Key Space	Bytes	Number of 2 TB Drives
6-char a-z, A-Z, 0-9	346 GB	< 1
8-char A-Z	1.3 TB	< 1
6-char, all chars (96)	4.8 TB	3
8-char a-z, 0-9	17.2 TB	9

Size of Rainbow Table for Various Key Spaces

The formula I'm using here is "(Charset Size)$^{(\text{Password Length})}$ * (Index Size) * (Salt Multiplier) / (Chain Length)." The tables use an index of 8 bytes, but you store the beginning and end for each chain, so that's 16 bytes total. The chain length (in this example) is 10,000 hashes/chain, and the Salt Multiplier is 4096 (the number of different hashes that the salt forces you to compute). So, 6-characters, a-z A-Z 0-9 would be $(26+26+10)^6$ * 16 * 4096 / 10000, approximately 346 GB. Longer chains would therefore decrease the table size proportionately, at the cost of longer search times.

So, yes, the salt still makes it difficult, but it's certainly not impossible. Is it finally time to invite crypt(3) to the party?

This is great and all, but, well, "Who Cares?"

That can be answered with two words: "Gawker Media." On December 12, 2010, hackers announced that they had compromised the systems at Gawker and stolen over 1.3 million email addresses (and associated password hashes) in use at Gawker's multiple web sites. Over 750,000 of these hashes are stored with crypt(3). At this writing, well over 200,000 of those hashes have been cracked, but that's still only about

a quarter of the entire set. What if the entire set could have been cracked with a Rainbow Table, in just a day?

Even beyond the Gawker incident, crypt(3) passwords are in use in many places. They're simple, well-understood, and well-supported. I've personally seen them in use supporting several different applications, including:

- In .htaccess files, providing basic access control for web pages
- OpenLDAP databases, for user authentication to desktops and servers
- TACACS password files, authenticating network administrators for access to routers, switches, and other enterprise hardware

When you get down to it, crypt(3) passwords, while fading from popularity, will continue to be found anywhere legacy systems or lazy administrators and programmers exist (which probably describes quite a few organizations).

Again, though, who cares? Why even bother? Well, I personally think that the proliferation of Rainbow Tables have helped to fuel better awareness of password storage systems, and their drawbacks. Also, I don't think that just because nobody's done this *publicly* before, that well-funded criminal organizations (or adversarial corporate or governmental organizations) haven't already developed this capability. So if this helps to put one more nail in the crypt(3) coffin, so we can move forward to better technology, then I'll be happy.

How does this work, anyway?

Well, first let's review how Rainbow Tables work. As stated above, they're just a sparsely populated table of hashes (though they're not really stored as hashes). You store only some very small fraction of key values, and then re-compute the missing values on the fly as you search. It's easiest to show with a pair of charts, presented on the next page (all computations are simulated).

The "Convert to plaintext" function is an arbitrary, but deterministic, conversion between a binary number and a string of readable letters. It won't be as neat as the (imaginary) example shown here – it just needs to be predictable and consistent. At the end of the process, you store the index for the beginning and end of the chain (here, "1337D00D 71011345") in the table, and continue on with another chain. The table itself is then just a list of thousands of such chains.

Start with a 4-byte index	1337D00D
Convert index to plaintext	passwd
Apply the hash function	76a2173b
Convert hash into a new index	DEADBEEF
Convert index to plaintext	S33krt
Plaintext to hash	197ee8e0
Hash to index	31415926
(etc.)	(etc.)
End of chain	71011345

Building a Rainbow Table

Once you have a table built, how do you search for a hash? Let's say you have the password hash "197ee8e0" – here's how you then recover the password that hash represents:

Target hash	197ee8e0
Convert to index. Is it in the table?	31415926
No. Convert to plaintext, hash it, index, compare, repeat, etc.	(etc.)
Yes! This index is in the table!	71011345
Return to the start of that chain and continue	1337D00D
Index to plaintext....	passwd
...plaintext to hash... Does it match the target?	76a2173b
No. Hash to index...	DEADBEEF
...index to plain....	S33krt
...plain to hash...Match? YES! Our password is "S33krt".	197ee8e0

Searching Rainbow Table for Hash

Why do we call these "Rainbow" Tables? The reduction function (that converts the index to a plaintext value) has to be predictable, as I said earlier. However, there's a strong

benefit to changing that function, subtly, for each step of the chain. This gradual morphing of the function is like a rainbow of colors across the whole spectrum. The advantage of this slight change is that a single index will resolve to a different plaintext, depending on where in the chain it falls. This helps to eliminate chain loops, merging, crossovers, and other things that complicate Rainbow Table generation and use.

That's neat. I mean, really neat. So...crypt(3)?

There are a few different approaches that could be taken to add support for crypt(3). The easy, and obvious way, is to do what the conventional wisdom suggests: Build a different table for each salt. Of course, that makes things quite complicated. Your front end has to take a list of hashes, break them up according to the salts in use, and pass each hash off to a different cracking process using a different set of tables. Furthermore, it's possible that, because of various Rainbow Table optimizations, a single (very large) table set might actually be smaller than 4096 different smaller sets.

I think I've found a better approach. Since the salt, essentially, makes an 8-character password into a 10-character password.... Instead of making 4096 tables, one for each salt, just make a single table, and include the salt as part of the password.

There's a little sleight-of-hand here, so I'll rephrase it. You build the table as usual, but change the target password length from 1-8 characters, to 3-10 characters. Then, after each index-to-plain step, you strip off the first two characters, and use those as the salt for the next plain-to-hash step. Eventually, you'll get every password and every salt. (Well, in a "Perfect Table," at least, but that's a deep, and deeply arcane, digression).

Start with a 4-byte index	06021023
Convert to plaintext. Call the first two characters the salt.	jlpasswd
Apply the hash function	jlGL5WwVAD7fo
Convert hash to index	71311811
Convert index to plaintext	yzS33krt
Apply the hash function	yz5o1UvS5lDAo
Convert hash to index	58132134
(etc.)	(etc.)
End of chain	F00FC7C8

Building a Table With Salted Hashes

Target hash	yz5o1UvS5lDAo
Convert to index. Is it in the table?	58132134
No. Continue hashing, checking, etc.	(etc.)
Yes! This index is in the table!	F00FC7C8
Return to the start of that chain and continue	06021023
Index to plaintext....	jlpasswd
...plaintext to hash... Does it match the target?	jlGL5WwVAD7fo
No. Hash to index...	71311811
...index to plain....	yzS33krt
...and match? YES! Our password is "S33krt".	yz5o1UvS5lDAo

Searching Rainbow Table for Salted Hash

So how's it work, in practice?

The performance is about 9 times slower, as expected, and the tables end up being larger, as expected, but it's still not out of the realm of possibility. A set of crypt(3) tables (except for a very short passwords and/or small character sets) won't fit on a laptop, but it'll certainly fit in a server closet, or even under a desk.

A good advantage to this approach: the changes to the code are relatively minor. Best of all, the changes don't conflict with the rest of the code. The table format doesn't change, and the same binary can continue to be used for whatever other algorithms it already supports. Other standard tools (provided they receive the same modifications) should continue to work as well.

Show me the code!

Absolutely! That's the whole point. First, a few caveats:

- I'm not a full-time programmer. Having extensive experience with perl and python doesn't mean that I can write optimal C code. Something like this definitely needs optimization.
- The changes I made are based on the "linuxrainbowcrack" project at Google Code. This projects doesn't appear to have been touched since April of this year, and may not be actively maintained.
- Really, you should consider this proof-of-concept code. The important bit is the idea, the rest should be pretty simple to implement (and, in fact, this only took me a couple of days to code).

ChainWalkContext.cpp – CChainWalkContext::IndexToPlain()

Added at end of routine, this flattens salt characters (the password itself uses a larger character set than the salt).

```
if (m_sHashRoutineName == "crypt") {
    unsigned char salt[65] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
wxyz0123456789/.";

    m_Plain[0] = salt[m_Plain[0] & 0x3f];
    m_Plain[1] = salt[m_Plain[1] & 0x3f];
}
```

ChainWalkContext.cpp – CChainWalkContext::GetPlainBinary()

Changed original “GetPlain()” call to the following, retrieving only the password portion of the plaintext (as the 1st two characters are the salt):

```
if (m_sHashRoutineName == "crypt")
    sRet += GetPlain().substr(2,8);

else
    sRet += GetPlain();
```

ChainWalkContext.cpp – CChainWalkContext::GetHashStr()

Finally, I added the following function (also adding its header to ChainWalkContext.h). This was required because crypt(3) hashes are generally viewed in their traditional base64-ish form, not as a hexdump, as the rest of the algorithms supported by this code are displayed.

```
string CChainWalkContext::GetHashStr()
{
    string sRet;
    int i;
    for (i = 0; i < MAX_HASH_LEN && i < 13; i++)
    {
        sRet += m_Hash[i];
    }

    return sRet;
}
```

CrackEngine.cpp – CheckAlarm()

Added new plaintext print statement, just before the existing print:

```
if (cwc.GetHashRoutineName() == "crypt")
    printf("plaintext of %s is %s\n", cwc.GetHashStr().c_str(),
      cwc.GetPlain().substr(2,8).c_str());
```

HashRoutine.cpp – CHashRoutine::CHashRoutine()

Added a line to add the new hash to the program's list of algorithms:

```
AddHashRoutine("crypt", HashCrypt, 13);
```

RainbowCrack.cpp – NormalizeHash()

Added a check at the top, to read the human-readable hashes and convert to hex:

```
if (sHash.length() == 13) {
    sNormalizedHash = HexToStr((unsigned char *)sHash.c_str(), 13);
    sHash = sNormalizedHash;
    return true;
}
```

HashAlgorithm.cpp

Added two includes, and the actual crypt(3) hash function (with appropriate header added to HashAlgorithm.h).

```
#include <unistd.h>
#include <string.h>

[......]

void HashCrypt(unsigned char* pPlain, int nPlainLen,
unsigned char* pHash)
{
    unsigned char *realPlain;
    unsigned char realSalt[2];

    realSalt[0] = pPlain[0];
    realSalt[1] = pPlain[1];
    realPlain = pPlain;
    realPlain += 2;

    memcpy(pHash, DES_crypt((const char*)realPlain, (const
char*)realSalt), 13);

}
```

RainbowCrack.cpp - main()

Added code to print these hashes and found passwords plainly (just before the existing print, in the "result" section):

```
if (vHash[i].length() == 26) {
    int t;
    unsigned char pHash[14];
    ParseHash(vHash[i], pHash, t);
    pHash[13] = '\0';

    printf("%s  %s hex:%s\n", pHash, sPlain.substr(2,8).c_str(),
sBinary.c_str());
} else
```

RainbowTableDump.cpp - main()

Added crypt(3) specific print statements just before the existing print:

```
if (cwc.GetHashRoutineName() == "crypt") {
    printf("#%-4d  %s  %s  %s\n", nPos,
      uint64tohexstr(cwc.GetIndex()).c_str(),
      cwc.GetPlainBinary().c_str(),
      cwc.GetHashStr().c_str());
} else
```

That's it. There are only about 50-some lines of code, and much of that is about reading, writing, and displaying the hashes.

Outstanding! Problems?

It's still kind of slow. I'm not sure how much attention crypt(3) has gotten in recent years, but perhaps this might encourage more research there. The algorithm might also benefit from being ported to GPU-based hardware. Unfortunately, I've read some discussion that DES, the basis of crypt(3), doesn't lend itself well to GPU systems, because of the extensive lookup tables that are involved in the algorithm.

Finally, no, I don't have prebuilt tables for you to download. Sorry.

Conclusion

Years ago, any approach for crypt(3) Rainbow Tables simply wasn't going to be feasible. However, with the continued application of Moore's law, CPUs (and storage) have reached the level where I think it's finally becoming practical. Unfortunately, a full, perfect table, of all 8-character passwords, using a full 96-element character set, is still beyond the reach of all but the most determined Rainbow Table enthusiasts.

Even with that, I still believe that shorter lengths can be (relatively) easily created, stored, and distributed. If other efforts, like Matt Weir's dictionary-based tables, are also applied, then even longer and more complex passwords can be added to the tables.

I hope this gets some people excited, gets some tables built, and that this has at least been an interesting read. Thanks!

References

Wikipedia: Rainbow Tables http://en.wikipedia.org/wiki/Rainbow_tables and crypt(3): http://en.wikipedia.org/wiki/Crypt_(Unix)

Good background reading: http://www.codinghorror.com/blog/2007/09/rainbow-hash-cracking.html and http://kestas.kuliukas.com/RainbowTables/

Seminal academic works: “A Cryptanalytic Time - Memory Trade-Off” (Hellmann, 1980): http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.2463&rep=rep1&type=pdf and “Making a Faster Cryptanalytic Time-Memory Trade-Off” (Oechslin, 2003): http://lasecwww.epfl.ch/~oechslin/publications/crypto03.pdf

Google code repository for linuxrainbowcrack: http://code.google.com/p/linuxrainbowcrack/

Diffs

What follows is the full, raw diff file, comparing my revisions against the last release of linuxrainbowcrack (on Google Code), from April 12, 2010.

```
diff -r original/ChainWalkContext.cpp djs/ChainWalkContext.cpp
428a429,435
> /* DJS */
>         if (m_sHashRoutineName == "crypt") {
>         unsigned char salt[65] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789/.";
>
>         m_Plain[0] = salt[m_Plain[0] & 0x3f];
>         m_Plain[1] = salt[m_Plain[1] & 0x3f];
>     }
470c477,482
<     sRet += GetPlain();
---
> /* DJS */
>     if (m_sHashRoutineName == "crypt")
>         sRet += GetPlain().substr(2,8);
>
>     else
>         sRet += GetPlain();
483a496,508
> /* DJS */
> string CChainWalkContext::GetHashStr()
> {
>     string sRet;
>     int i;
>     for (i = 0; i < MAX_HASH_LEN && i < 13; i++)
>     {
>         sRet += m_Hash[i];
>     }
>
>     return sRet;
> }
>
diff -r original/ChainWalkContext.h djs/ChainWalkContext.h
72a73
>     string GetHashStr(); /* DJS */
diff -r original/CrackEngine.cpp djs/CrackEngine.cpp
92c92,98
<         printf("plaintext of %s is %s\n", cwc.GetHash().c_str(), cwc.GetPlain().c_str());
---
> /* DJS */
>         if (cwc.GetHashRoutineName() == "crypt")
>             printf("plaintext of %s is %s\n", cwc.GetHashStr().c_str(), cwc.GetPlain
().substr(2,8).c_str());
>
>         else
>             printf("plaintext of %s is %s\n", cwc.GetHash().c_str(), cwc.GetPlain().c_str
());
>
131,132c137
<         //printf("debug: using %s walk for %s\n", fNewlyGenerated ? "newly generated" :
"existing",
<         //                                      vHash[nHashIndex].c_str());
---
> //printf("debug: using %s walk for %s\n", fNewlyGenerated ? "newly generated" : "existing",
vHash[nHashIndex].c_str());
```

```
diff -r original/HashAlgorithm.cpp djs/HashAlgorithm.cpp
15a16,19
> /* DJS */
> #include <unistd.h>
> #include <string.h>
>
59a64,78
>
>
> /* DJS */
> void HashCrypt(unsigned char* pPlain, int nPlainLen, unsigned char* pHash)
> {
>     unsigned char *realPlain;
>     unsigned char realSalt[2];
>
>     realSalt[0] = pPlain[0];
>     realSalt[1] = pPlain[1];
>     realPlain = pPlain;
>     realPlain += 2;
>
>         memcpy(pHash, DES_crypt((const char*)realPlain, (const char*)realSalt), 13);
> }
diff -r original/HashAlgorithm.h djs/HashAlgorithm.h
13a14,16
> /* DJS */
> void HashCrypt(unsigned char* pPlain, int nPlainLen, unsigned char* pHash);
>
diff -r original/HashRoutine.cpp djs/HashRoutine.cpp
22a23,25
>
> /* DJS */
>     AddHashRoutine("crypt", HashCrypt, 13);
diff -r original/RainbowCrack.cpp djs/RainbowCrack.cpp
77a78,84
> /* DJS */
>     if (sHash.length() == 13) {
>         sNormalizedHash = HexToStr((unsigned char *)sHash.c_str(), 13);
>         sHash = sNormalizedHash;
> //printf("Normalized: %s\n", sHash.c_str());
>         return true;
>     }
347a355,364
> /* DJS */
>             if (vHash[i].length() == 26)
>             {
>                 int t;
>                 unsigned char pHash[14];
>                 ParseHash(vHash[i], pHash, t);
>                 pHash[13] = '\0';
>
>                 printf("%s  %s hex:%s\n", pHash, sPlain.substr(2,8).c_str(), sBinary.c_str
());
>             }
349c366,367
<             printf("%s  %s  hex:%s\n", vHash[i].c_str(), sPlain.c_str(), sBinary.c_str());
---
>             else
>                 printf("%s  %s  hex:%s\n", vHash[i].c_str(), sPlain.c_str(), sBinary.c_str
());
```

```
diff -r original/RainbowTableDump.cpp djs/RainbowTableDump.cpp
56,59c56,69
<             printf("#%-4d  %s  %s  %s\n", nPos,
<                                           uint64tohexstr(cwc.GetIndex()).c_str(),
<                                           cwc.GetPlainBinary().c_str(),
<                                           cwc.GetHash().c_str());
---
> /* DJS */
>             if (cwc.GetHashRoutineName() == "crypt")
>             {
>                 printf("#%-4d  %s  %s  %s\n", nPos,
>                   uint64tohexstr(cwc.GetIndex()).c_str(),
>                   cwc.GetPlainBinary().c_str(),
>                   cwc.GetHashStr().c_str());
>             } else
>             {
>                 printf("#%-4d  %s  %s  %s\n", nPos,
>                   uint64tohexstr(cwc.GetIndex()).c_str(),
>                   cwc.GetPlainBinary().c_str(),
>                   cwc.GetHash().c_str());
>             }
```