Optimizing computation of Hash-Algorithms as an attacker

Version 1.0 - 2013.05.12

About me

- Name: Jens Steube
- Nick: atom

- Coding Projects:
  - hashcat / oclHashcat
- Work Status:
  - Employed as Coder, not crypto- or security-relevant

- In the context of password guessing the computation of hash-algorithms can be optimized so that an attacker needs less work to do then a defender.

- The reason is that hash-algorithms typically are not designed to protect passwords - but developers use them to do it since decades.

- This situation leads to optimizations that are possible for an attacker that the autors of the hash-algorithms never tried to avoid.

General Techniques

Overview

General Techniques

Technique
Zero-based optimizations
Early-exit optimizations
Initial-step optimizations
Precomputing
Reversing aka Meet-in-the-middle

NOTE

- Some of the techniques in these slide are known, some are not
- All of them are used in oclHashcat-lite

Zero-based optimizations

General Techniques

Zero-based optimizations

- From computers view, passwords are just numbers
- The password 'password' takes two 32-bit integers

```
0000000: 70617373 776f7264                      password
```

- It is copied to the algorithms input data block
- Typically fixed 512-bit, padded with zeros

```
0000000: 70617373 776f7264 00000000 00000000  password........
0000010: 00000000 00000000 00000000 00000000  ................
0000020: 00000000 00000000 00000000 00000000  ................
0000030: 00000000 00000000 00000000 00000000  ................
```

- The above buffer shows the w[] array build out of the 16 elements

Zero-based optimizations

- The password 'password' only took w[0] and w[1] to be populated, the rest are zero

```
0000000: 70617373 776f7264 00000000 00000000  password........
0000010: 00000000 00000000 00000000 00000000  ................
0000020: 00000000 00000000 00000000 00000000  ................
0000030: 00000000 00000000 00000000 00000000  ................
```

- This is how a single step function from the MD4 F() is calculated:

```
#define FF(a,b,c,d,x,s) a += F (b, c, d) + x; ...
```

- The function is called 16 times, each call using a different element of the input vector

Zero-based optimizations

```
#define FF(a,b,c,d,x,s) a += F (b, c, d) + x; ...
```

- If x is 0 then there is no change in that last addition
- If the password only has the length 8, we know that w[2] - w[15] are zero
- For all function calls that use w[2] - w[15] we can remove the addition completely
- Note that most algorithms fill w[14] or w[15] in final()

Zero-based optimizations

- Conclusion
  - In MD4, this saves (on passwords < length 12) a total of 36 / 128 ADD instructions
  - In MD5, this saves (on passwords < length 12) a total of x / 128 ADD instructions
  - TBD, add more hash-algorithms …
- NOTE
  - AFAIK, this technique works on all raw hashing algorithms
  - HMAC defeats this

Initial-step optimizations

General Techniques

Initial-step optimizations

- Here is a snippet from MD5

```
a = 0x67452301; b = 0xefcdab89;
c = 0x98badcfe; d = 0x10325476;

FF (a, b, c, d, w[0],  7, 0xd76aa478);
FF (d, a, b, c, w[1], 12, 0xe8c7b756);
FF (c, d, a, b, w[2], 17, 0x242070db);
FF (b, c, d, a, w[3], 22, 0xc1bdceee);
...
```

- The MD5 step macro looks like MD4 step macro in the beginning

```
#define FF(a, b, c, d, x, s, ac) a += F (b, c, d) + x + ac; …
```

Initial-step optimizations

```
a = 0x67452301; b = 0xefcdab89;
c = 0x98badcfe; d = 0x10325476;

FF (a, b, c, d, w[0], 7, 0xd76aa478);
```

- The only unknown part in the first step is w[0]
- All the other data is known
- We can replace the original MD5 step macro:

```
#define FF(a, b, c, d, x, s, ac) a += F (b, c, d) + x + ac; ...
```

- With this one:

```
#define FF(a, b, c, d, x, s, ac) a  = 0xd76aa477 + x; ...
```

Initial-step optimizations

- Conclusion
  - Per step 1
    - In MD4, this saves 6 instructions (x AND, x NOT, x OR, x ADD)
    - In MD5, this saves 6 instructions (x AND, x NOT, x OR, x ADD)
    - TBD, add more hash-algorithms …
  - The next 3 steps can also be optimized like this
- Note that HMAC does not defeat this

Early-exit optimizations

General Techniques

Early-exit optimizations

- Here is a snippet from the last 4 steps of MD5

```
II (a, b, c, d, w[ 4], S41, 0xf7537e82); /* 61 */
II (d, a, b, c, w[11], S42, 0xbd3af235); /* 62 */
II (c, d, a, b, w[ 2], S43, 0x2ad7d2bb); /* 63 */
II (b, c, d, a, w[ 9], S44, 0xeb86d391); /* 64 */

state[0] += a;
state[1] += b;
state[2] += c;
state[3] += d;
```

- Again, in password guessing, the password is to short to force a second transform
- Thus, we know the value of state[0], which is 0x67452301 in MD4/MD5/SHA1

Early-exit optimizations

- We know the value of state[0], which is 0x67452301 in MD4/MD5/SHA1
- We simply subtract that constant from the hash we want to guess
- Example, if our hash is 8743b52063cd84097a65d1633f5c74f5, then we can do:

  hash[0] = 0x8743b520 - 0x67452301 = 0x1ffe921f

- Note that we do not need to be afraid to underflow the integer. Actually, this is exactly what hashes do, too

Early-exit optimizations

```
hash[0] = 0x8743b520 - 0x67452301 = 0x1ffe921f
```

- Since ‘a’ does not change anymore after step 61 in MD5 we can do the comparison that makes this optimization an early-exit

```
II (a, b, c, d, w[ 4], S41, 0xf7537e82); /* 61 */

if (a != hash[0]) return - 1;

II (d, a, b, c, w[11], S42, 0xbd3af235); /* 62 */
II (c, d, a, b, w[ 2], S43, 0x2ad7d2bb); /* 63 */
II (b, c, d, a, w[ 9], S44, 0xeb86d391); /* 64 */
```

Early-exit optimizations

- Conclusion
  - We can skip 3 entire steps
    - In MD5, this means 30 instructions
    - In MD4, this means 20 instructions
    - TBD, add more hash-algorithms …
- Note that Keccak is safe from this

Precomputing

General Techniques

Precomputing

- This technique heavily depends on how much memory we have and how fast it is to lookup a value from a table
- Keep our eyes open to see chances for precomputation
  - Typically they give high rates of optimization
  - Good examples for this is the Whirlpool and Oracle (old) hash, we'll focus them later
- Example following …

Precomputing

- Assume we want to compute a SHA256 and we have a candidate generator that only changes the last characters of the password
- In other words, do not change the first 4 characters in w[0]
- If w[0] does not change, in step 1 we see:

```
t1 = H + S1(E) + Ch(E,F,G) + K[0] + w[0];
t2 = S0(A) + Maj(A,B,C);
H = G; G = F; F = E; E = D + t1; D = C; C = B; B = A; A = t1 + t2;
```

- Since A - H and K[0] are all constants and w[0] does not change, we can precompute the new values for A and D completely
- In the inner loop, we can start from step2

Partial reversing / Meet-in-the-middle

General Techniques

Partial reversing / Meet-in-the-middle

- This technique can not reverse back to it’s original password, but it can be used to speed up the computation
- There is that statement that MD5 is a one-way function. It’s true and it’s not true;
    - In case we have the original data used to generate the MD5 digest we -can- reverse the digest back to the original MD5 initial values
    - This works for MD4 and SHA1, too, maybe more…
- But in password guessing, how does that make any sense since what we don't have is exactly the original data?

Partial reversing / Meet-in-the-middle

- When we generate password candidates in our password guesser, we can generated them in a specific way

- Assume we have a generator that only changes the first 4 characters of a password and hold the rest stay constant

```
AAAApass
AAABpass
ZZZZpass
```

- As a result, we change only w[0] and w[1] – w[15] stay constant

Partial reversing / Meet-in-the-middle

- In the outer loop
  - The goal is to reverse back the algorithm steps as long w[0] is not required to compute that specific step
  - Store the intermediate digest of that specific step
- In the inner loop
  - Start the normal "forward" computation of the hash
  - Once we hit the step 1.1 on which we calculated the intermediate digest we can stop the computation
  - Compare the digest with the intermediate digest from step 1.2
  - If it does not match there is no need to continue to calculate the rest of the steps. This is what gives the speed boost

Targeted Algorithms

Overview

Targeted Algorithms

Algorithm
MD4
MD5
SHA1
NetNTLMv1
Whirlpool
Oracle (DES)
HMAC

MD4

Targeted Algorithms

MD4

Used in

- MD4 (raw)
- NTLM
- Domain Cached Credentials
- Domain Cached Credentials2
- NetNTLMv1
- NetNTLMv2

Techniques

- Zero-based optimizations
- Initial-step optimizations
- Early-exit optimizations
- Precomputing
- Partial reversing / Meet-in-the-middle

MD4

- It's a good example how to do all the techniques, especially the partial reversing
- As mentioned in the technique description we need to “reverse” the hash to a specific step
- This step is where w[0] is set the last time in the algorithm
- In case of MD4, it is step 33

```
HH (a, b, c, d, w[0], S31);
```

MD4

- Here's the original HH step macro of MD4:

```
#define HH(a, b, c, d, x, s)
  a += 0x6ed9eba1;
  a += x;
  a += H (b, c, d);
  a  = ROTATE_LEFT (a, s);
```

- And the reversal is just an inverted function:

```
#define HH_REV(a, b, c, d, x, s)
  a  = ROTATE_RIGHT (a, s);
  a -= H (b, c, d);
  a -= x;
  a -= 0x6ed9eba1;
```

MD4

- We call the inverted function from the last step back to step 34

```
HH_REV (d, a, b, c, w[ 8], S32); /* 34 */
```

- We can not reverse more at this point since w[0] is unknown in the outer loop

```
HH_REV (a, b, c, d, w[ 0], S31); /* 33 */
```

- What we now have is the last intermediate value of D. This means we can do the comparison of D when it was set the last time which is:

```
GG (d, a, b, c, w[ 7], S22); /* 30 */
```

- Note that our meet-in-the-middle comparison is to do after step 30

MD4

- In our inner loop, we just compute MD4 forward:

```
FF (a, b, c, d, x[ 0], S11); /* 1 */
FF (d, a, b, c, x[ 1], S12); /* 2 */
FF (c, d, a, b, x[ 2], S13); /* 3 */
…
```

- Now we can do the compare at step 30

```
…
GG (a, b, c, d, x[ 3], S21); /* 29 */
GG (d, a, b, c, w[ 7], S22);

if (d != d_rev) return -1;
```

- If „d“ does not match, there is no need to continue the computation

MD4

- NOTE
  - Actually it’s possible to reverse this even more but this takes more explanation
  - oclHashcat-lite does it’s meet-in-the-middle comparison after step 26

MD4

- Conclusion
  - We only need to calculate 30 of 48 steps due to the reversing resulting in 33% speed increase
  - All other optimization techniques will also apply, but in combination with reversing they do not make sense, except the Initial-step optimization

MD5

Targeted Algorithms

MD5

Used in

- MD5 (raw)
- Everywhere

Techniques

- Zero-based optimizations
- Initial-step optimizations
- Early-exit optimizations
- Precomputing
- Partial reversing / Meet-in-the-middle

MD5

- The partial reversing works like in MD4
- The inverted function looks a bit different

```
#define II_REV(a, b, c, d, x, s, ac)
   a -= b;
   a  = ROTATE_RIGHT (a, s);
   a -= I (b, c, d);
   a -= x;
   a -= ac;
```

MD5

- Conclusion
  - We only need to calculate 46 of 64 steps
- NOTE
  - It’s possible to reverse this even more but this takes more explanation
  - oclHashcat-lite does it’s meet-in-the-middle comparison after step 43

SHA1

Targeted Algorithms

SHA1

Used in

- SHA1 (raw)
- Everywhere

Techniques

- Zero-based optimizations
- Initial-step optimizations
- Early-exit optimizations
- Precomputing

SHA1

- I've already published a very specific optimization technique for SHA1 that is a form of precomputing at Passwords^12
- It works by reducing the number of XOR's that are required in the key-stretching phase and gains a total of 22% in performance increase
- It’s too much to explain the details in this presentation. For details, please take a look at this PDF:
  - http://hashcat.net/p12/js-sha1exp_169.pdf
- There is another optimization for SHA1 based on early-exits
- You can use it as a standalone optimization in case you can not use the XOR based optimization and you can use it in combination with it. Both variants work

SHA1

- SHA1's step macro is a bit different to MD4 / MD5
- It sets both B and E (not just A) for each step

```
#define F4(f,a,b,c,d,e,x)
    e += 0xca62c1d6;
    e += x;
    e += b ^ c ^ d;
    e += ROTATE_LEFT (a,  5);
    b  = ROTATE_LEFT (b, 30);
```

- Typical implementation of the last steps looks like:

```
F4 (d, e, a, b, c, w[77]);
F4 (c, d, e, a, b, w[78]);
F4 (b, c, d, e, a, w[79]);
```

SHA1

- From looking at the code one would say we can early-exit is in step 77, checking for E
- That’s correct, but we can do more. Take a close look again at the step:

```
b  = ROTATE_LEFT (b, 30);
```

- All it does is a simple rotate. Therefore we have no “loss” of information at all
- We know E at the end (from our hash) so we can pre-reverse it

```
e  = ROTATE_RIGHT (e, 30);
```

- Now, instead of doing an early-exit at step 77, we can do the early-exit at step 75

SHA1

- Conclusion

  - 22% increase due to XOR optimization
  - Another 4 steps saved due to extended early-exit optimization

NetNTLMv1

Targeted Algorithms

NetNTLMv1

Used in

- Windows Networks
- File Shares
- SAMBA

Technique

- Early-exit

NetNTLMv1

- Even if it‘s „just“ an early-exit, it‘s an early-exit with big impact
- To explain it, it‘s important to explain how to algorithm works
- NOTE
    - Examples takes from hashcat.net forum

NetNTLMv1

- The algorithm work by generating an NTLM hash out of the password

```
hashcat -> b4b9b02e6f09a9bd760f388b67351e2b
```

- This hash is then broken into 3 x 56 bit parts and padded with zeros

```
b4b9b02e6f09a9  bd760f388b6735  1e2b0000000000
```

- Each 56 bit part is odd parity adjusted to result in 3 x 64 bit parts

```
b4b9b02e6f09a9 -> b55d6d04e6792652
bd760f388b6735 -> bcba83e6895b9d6b
1e2b0000000000 -> 1f15c10101010101
```

- NOTE: MD4 outputs only 128 bit, the third part uses only 16 bits

NetNTLMv1

- Each of these part is then used as a key to DES encrypt the 8 byte challenge resulting in 3 ciphertext blocks:

```
b55d6d04e6792652 KEY-> DES(1122334455667788) -> 51a539e6ee061f64
bcba83e6895b9d6b KEY-> DES(1122334455667788) -> 7cd5d48ce6c68665
1f15c10101010101 KEY-> DES(1122334455667788) -> 3737c5e1de26ac4c
```

- The key of the DES buffer that results in the 3737c5e1de26ac4c is limited to a keyspace of only 64k possible variants
- This is such a tiny keyspace so that we can brute-force it in the startup phase on CPU typically in less than a second
- Once we found the input key, we know the last 16 bit of the NTLM that was used to generate the NetNTLMv1 hash is 0x1e2b

NetNTLMv1

- Instead of doing all these steps again and again for each candidate, all we need to do is to generate the NTLM hash
- Then we do:

```
if ((d & 0xffff) != d_pre) break;
```

- Once it passes this comparison (with a chance of 2^16) we continue to calculate the hash the original way

NetNTLMv1

- Conclusion
  - By doing this optimization we can crack NetNTLMv1 by nearly the same speed as an NTLM plus we still can use Zero-based optimizations and Initial-step optimizations to speed up the NTLM computation
  - Especially on GPU’s this optimization helps a lot since this save the expensive s-box lookups from DES
  - The performance gain is massive and it‘s hard to calculate, but it‘s at least 5-10 times faster than without the optimization, maybe more

whirlpool

Targeted Algorithms

whirlpool

Used in

- TrueCrypt

Technique

- Precomputing

whirlpool

- In whirlpool we can precompute all the rounds of K[] which takes 50% of the entire whirlpool computation otherwise
- K[] is based on the initial hash value, which is always 0 at start. For each round it’s doing some s-box lookups and XOR’s them with the old value
- I've written a C program for precomputing, you can grab it here:
    - http://hashcat.net/p13/whirlpool_pc_K.c

whirlpool

- But we can do even more! Once we have K[] precomputed we can reverse our target hash as we did in the early-exits in MD5/SHA1:

```
state[0] = digest_buf[0] ^ pc_K[10][0];
state[1] = digest_buf[1] ^ pc_K[10][1];
state[2] = digest_buf[2] ^ pc_K[10][2];
...
```

- Still there is more room for optimization. Since whirlpool does not use any arithmetic instructions but only bitwise there is no data loss that we can not reproduce
- That means if we use our special password candidates generator again that only changes w[0] in the inner loops, we can precompute all values of the first round of Whirlpool
- In the first iteration we have to "correct" the data from the precomputation. That is because w[0] changed slightly.

Whirlpool

- Conclusion
  - Whirlpool can be optimized by more than 50% of the entire calculation
  - The optimization works even when protected with HMAC

Oracle (old)

Targeted Algorithms

Oracle (old)

Used in

- Oracle 7-10g

Technique

- Precomputing

Oracle (old)

- This algorithm uses a cipher not a hash to store the password
- The scheme is the following:
  - Encryption is DES-CBC
  - IV = 0x0000000000000000
  - KEY = 0x0123456789ABCDEF
  - DATA = password
- In ciphers, to make cracking attempts slow, they have designed the keysetup phase to calculate slow
- In this algorithm is that part that should be the slow part, the keysetup, useless since it only the IV and the KEY and both of them are known :)
- We can precompute the slow part by 100%, the 16x 48-bit subkeys

Oracle (old)

- Conclusion
  - Since the algorithm steps depend on the password length it’s hard to say how much time is saved
  - Again, especially on GPU’s this optimization helps alot since this save the expensive s-box lookups from DES

HMAC

Targeted Algorithms

HMAC

Used in

- TrueCrypt
- WPA/WPA2
- 1password
- LUKS
- LastPass
- DCC2
- PBKDF2 (always used)

Technique

- Precomputing

HMAC

- Most important to say here, HMAC is not an algorithm itself. HMAC is a construction of a MAC (keyed hash) using a hash function
- We can call HMAC very important since it's used in many important algorithms to protect passwords
- It also defeats zero-based optimizations
- HMAC definition is simple
    - HMAC (K,m) = H ((K ^ opad) || H ((K ^ ipad) || m))

HMAC

- Because of the HMAC paddings of the ipad and the opad, we have a special situation
- The HMAC fills this buffer by it’s definition with 0x363636363636... and 0x5c5c5c5c5c5... and then XOR's the password on it
- This construct completely fills the buffers so that we have all data ready to run the first call to the selected hash transform
- We do this as soon we know the final password and then cache the results of both ipad and opad for later usage

HMAC

- Conclusion
  - Reduces the number of calls to H() by 2 for each iteration
  - Number of calls to H() in WPA/WPA2 is 4, lead to a performance increase of 50% compared to defender

Failed custom schemes

Overview

Failed custom Schemes

Application	Failure type
IPB2	Salt integration
1Password	Block-chaining

IPB2

Failed custom schemes

IPB2

Used in

- Invision Power Board

Technique

- Precomputing

IPB2

- This is simple to explain
- The IPB2 scheme is defined as
  - md5 (md5 (salt) + md5 (pass))
- Since the salt is known, we can precompute it after loading the hash
- Attacker requires just 2 calls to H() but defender requires 3

IPB2

- Conclusion
  - Performance increased by > 30%

1Password

Failed custom schemes

1Password

Used in

- AgileBits 1Password

Techniques

- Early-exit
- Precomputing

1Password

- This is an example of how an algorithm was designed in a way where the designer was lacking the vision of an attacker
- 1Password uses PBKDF2-HMAC-SHA1 to derive a 256 bit key
- PBKDF2 can be configured on how much bits of output to produce
- In this case it’s HMAC-SHA1, but SHA1 outputs only 160 bits. So that means the defender has to run two rounds on SHA1 to produce 320 bits output and then let PBKDF2 truncates it to 256 bits
- 1Password then uses AES in CBC mode to decrypt 1040 byte of data
- To be exact, it takes the first 128 bit of the derived key to setup the AES key and takes another 128 bit as an IV for the CBC

1Password

- The goal is match the final padding block after decrypting 1040 byte of data. If you find the last four 32-bit integers at 0x10101010 the padding is correct and you know your key was correct
- But in combination with the CBC mode you use the IV only for the first decryption. You then replace it with the ciphertext of current block (which is used for the next block)
- Again, to validate the masterkey is correct, all we need is to match the padding value. To satisfy the CBC we just need the previous 16 byte of data from the 1040 byte block. We do have it already since it was provided by the keychain
- There is no need to calculate the IV at all
- Instead of calculating a 256 bit key in the PBKDF2, we just need to calculate 128 bit

1Password

- Conclusion
    - Using both optimizations, Early-exit (using HMAC optimization) plus only calculating 128 bits instead of 256 increased the performance of an attacker compared to the defender by 400%

Thank you for listening!

- Feel free to contact me!
  - via Twitter: @hashcat
  - via Hashcat forum: http://hashcat.net/forum/
  - via IRC: freenode #hashcat
  - via Email: atom at hashcat.net