RACF password masking

Old style dating way back to beginning of RACF

Why it’s needs to be eliminated,

And how to manage it.

Masking algorithm

http://www.rshconsulting.com/racftips/RSH_Consulting__RACF_Tips__October_2015.pdf

Above link gives full explanation of the significance of old style masking and upgrading to z/OS v2.2 In summary RACF databases can easily contain old masked passwords if the password was stored when the original masking algorithm was active, and the password has never been changed since.

Any such masked passwords present in the RACF database will stop working after upgrading to z/OS v2.2

So just how bad is the old proprietary masking algorithm?

http://2000clicks.com/links/Computers/IBMMainframeHistory/ichdex01.htm

Above link by Thierry FALISSARD gives an insight, i.e. it doesn't take too much to realise it's pretty weak! (partial extract below shown)

```
Profile  Password
OMVS     C16337002C4C4C4C
OMVSKERN C1633700A5B13F72
```

OMVS and OMVSKERN have very close passwords : the first 4 bytes are the same. Not very hard to guess what are the passwords in clear...

More background on masking

RACF Update

Share Orlando

Session 17550 – August 11 2015

Eric Rosenfeld (IBM Poughkeepsie)

Some good info including:

- Originally, passwords were stored in a “masked” format
- Reversible!

Masking algorithm

Password Cracking and Self-Encrypting Drives
Presented by: Chad Rikansrud at SHARE in San Antonio 2016
Published the logic of the original masking algorithm as follows:

OMVS

```
pass  D6D4E5E240404040

p1    11010110 11010100 11100101 11100010 01000000 01000000 01000000 01000000
ls1   10101101 10101001 11001011 11000100 10000000 10000000 10000000 10000000
xor   01111011 01111101 00101110 00100110 11000000 11000000 11000000 11000000
rs4   00000111 10110111 11010010 11100010 01101100 00001100 00001100 00001100
xor   11010001 01100011 00110111 00000000 00101100 01001100 01001100 01001100
xor   0001

hash  C16337002C4C4C4C
```

So this appears to be a one-way function

However lets generate some masks and see what we can deduce

```
AAAAAAAA : C1C1C1C1C1C1C1C1 : D5E5E5E5E5E5E5E5
BBBBBBBB : C2C2C2C2C2C2C2C2 : D6B6B6B6B6B6B6B6
CCCCCCCC : C3C3C3C3C3C3C3C3 : D787878787878787
DDDDDDDD : C4C4C4C4C4C4C4C4 : D0101010101010101
EEEEEEEE : C5C5C5C5C5C5C5C5 : D121212121212121
FFFFFFFF : C6C6C6C6C6C6C6C6 : D272727272727272
GGGGGGGG : C7C7C7C7C7C7C7C7 : D343434343434343
HHHHHHHH : C8C8C8C8C8C8C8C8 : DD5D5D5D5D5D5D5D
IIIIIIII : C9C9C9C9C9C9C9C9 : DC6C6C6C6C6C6C6C
JJJJJJJJ : D1D1D1D1D1D1D1D1 : C6F6F6F6F6F6F6F6
```

Initial deductions

- First character is easy to predict
- The mask produced appears predictable,
  i.e. strong patterns are obvious which don't appear very randomised
- It's not quite as simple as a straight lookup,
  i.e. 'A' doesn't always equate to D5, only if it's the first character
- The difference with 'A' becoming D5 instead of E5 in the first position is not as a result of the final XOR'ing of 0001 in the algorithm

Closer examination of the algorithm (ignoring xor's)

left shift 1

right shift 4

EBCIDIC codes for A-Z and 0-9 all have the left most bit set, so a left shift 1 is going to be the same for all (I'll examine other characters in due course)

Right shift 4 means each character is going to exert an influence on the character to it's right, and it's going to be a predictable repeatable influence.

Closer examination of the algorithm continued…

Because each character is coded using eight bits, right shift 4 means that it's only the low order nibble of the first byte which influences the high order nibble of it's adjacent neighbour

This is very significant as it means the influence remains very localised between adjacent bytes and doesn't get cascaded further along!

In other words it must be possible to simulate this doing a lookup, which gives the decoded value of the current character, along with a mask to be applied (i.e. xor'd) with the next character. This was the *eureka* moment!

Closer examination of the algorithm continued…

So, we've almost cracked it, but why the last XOR on just first 4 bits?

This puzzled me, until I started generating the lookup table and offsets for simulating the algorithm. The characters are A-Z plus 0-9 plus the 3 National characters (@,#,$) and not forgetting the 'space' character!

The 'space' character is EBCDIC 40 which gets encoded to 4C and by XOR'ing with 0001 changes that to 5C making it invalid as a first character which makes sense if you want to prevent what would be a null password as spaces are assumed for any non-typed characters.

Those pesky National characters!

As mentioned earlier, all A-Z and 0-9 have the high order bit set. This effectively makes the left shift redundant, except with these National characters as they don't!

Hence, in order to cope with the possibilities of this bit being either 0 or 1 then additional lookup table entries are required for these characters.

These weren't included in racfunmask v1.0 but are included in racfunmask v1.1 (i.e. v1.0 only worked 50% where National characters were involved in the password)

Tools available

The following signed executable Windows programs are available:

racfmask – allows creation of proprietary hashed passwords for testing

racfunmask – finds and decrypts ALL masked passwords very quickly

Available from www.racfsnow.co.uk

Here is my lookup table complete with masks

(character ebcdic : encoded : mask)

space 40 : 4C : 0	G C7 : D3 : 9	Q D8 : CE : 8	0 F0 : E1 : 0
$ 5B : 45 : D	H C8 : DD : 8	R D9 : CF : B	1 F1 : E0 : 3
# 7B : 63 : D	I C9 : DC : B	S E2 : F0 : 6	2 F2 : E3 : 6
@ 7C : 64 : 4	J D1 : C6 : 3	T E3 : F1 : 5	3 F3 : E2 : 5
A C1 : D5 : 3	K D2 : C5 : 6	U E4 : F6 : C	4 F4 : E5 : C
B C2 : D6 : 6	L D3 : C4 : 5	V E5 : F7 : F	5 F5 : E4 : F
C C3 : D7 : 5	M D4 : C3 : C	W E6 : F4 : A	6 F6 : E7 : A
D C4 : D0 : C	N D5 : C2 : F	X E7 : F5 : 9	7 F7 : E6 : 9
E C5 : D1 : F	O D6 : C1 : A	Y E8 : FB : 8	8 F8 : E8 : 8
F C6 : D2 : A	P D7 : C0 : 9	Z E9 : FA : B	9 F9 : E9 : B

$ 5B : 55 : D
# 7B : 73 : D
@ 7C : 74 : 4

Worked example of unmasking!

```
OMVS
hash    C16337002C4C4C4C

hash    11000001 01100011 11100101 11100010 01000000 01000000 01000000 01000000
mask    0000     1010     1100     1111     0110     0000     0000     0000
xor     11000001 11000011 11110111 11110000 01001100 01001100 01001100 01001100
lookup  11010110 11010100 11100101 11100010 01000000 01000000 01000000 01000000

pass    D6D4E5E240404040
```

Conclusion

Hopefully this will have achieved the following:

- Convinced folks the need to eliminate masked passwords
- Shown why it’s never a good idea to have proprietary encryption
- Given background on the tools available to move forward safely