TR-A-0008

Ｕｎｉｘのセキュリティに関する考察

城　和貴

Kazuki　Joe

１９８７．６．２２

ＡＴＲ視聴覚機構研究所

目次

序章　Ｕｎｉｘのセキュリティ

Ｕｎｉｘにログインするためにはアカウントとパスワードが必要なことは言うまでもないが、このパスワードはどれくらい安全なのであろうか？
辞書順に、しらみ潰しにパスワードを入力して、任意のアカウントのパスワードを手にいれる確率はどれくらいあるのだろうか？
或は、パスワード・ファイルのデータをもとに、システムが破られる危険性はどれくらいあるのだろうか？

このような、システム管理者なら誰でも一度は考える疑問に答えるべく、Ｕｎｉｘのパスワード生成ルーチンの解析と、パスワード解読ルーチンの可能性について調べてみた。

第１章　　Ｕｎｉｘの暗号生成ルーチン

Ｕｎｉｘにはｃｒｙｐｔという暗号生成のためのシステム・ルーチンがあり、ｐａｓｓｗｄコマンド等でパスワードの作成、変更を行うときには、このルーチンが使われている。
このルーチンは２つの引数を持ち、（１つがパスワード、他方はｓａｌｔと呼ばれる２バイトの文字列で、ｐａｓｓｗｄコマンドによりランダムに決定される）印字可能な１３文字の暗号を出力する。

このルーチンを利用して、パスワードをしらみ潰しに探すと、任意のアカウントのパスワードを探り当てるのにどれくらいＣＰＵを消費するだろうか？
概算するためにはｃｒｙｐｔの１回あたりのＣＰＵタイムが必要であるので、次のようなプログラムでＣＰＵタイムを測定してみた。

```
/*****************************************************************/
/*      crypt_bench.c                                            */
/*      crypt benchmark test program                             */
/*                      By K.Joe                                 */
/*      % cc -p crypt_bench.c                                    */
/*      % prof                                                   */
/*****************************************************************/

#include <stdio.h>
main()
{
        int     count;
        for (count=0;count<100;count++)
                crypt("abcdefgh","XY");
}
```

結果は次のとうりである。（ａｔｒ－ｈｒにて実行）

```
 %time  cumsecs  #call  ms/call  name
  97.4    30.02   2500    12.01  _encrypt
   2.6    30.82    100     8.01  _crypt
   0.1    30.84                  _monstartup
   0.0    30.84      1     0.00  _main
   0.0    30.84      1     0.00  _profil
```

つまり、１回のｃｒｙｐｔに約０．３秒かかっている。
今、仮に探したいパスワードが英小文字８文字からなるとすると、これを辞書順でしらみ潰しに探すのに要する時間は、

$$26^8 * 0.3 \text{（秒）} = 60424 \text{（年）}$$

どうやら現在より数億倍速いコンピュータができるまでは安全のようだ。
（もっとも、そのころまでＵｎｉｘは使われていないだろうが....）
もちろん、プログラムの高速化もできるかもしれないが、パスワードには数字、英大文字も使えることを考えると、現在の段階ではスーパー・コンピュータを数十台使わなければ不可能と考えるべきだろう。

参考までに、ｃｒｙｐｔを使ったサンプル・プログラムをいくつか上げておく。

```
/*************************************************************************/
/*      simulate_passwd.c                                                */
/*************************************************************************/
/*      This program requires the encryption string in /etc/passwd  */
/*      and ask you for passwd until you type correctt one           */
/*                                                                       */
/*                      By K.Joe                                         */
/*************************************************************************/

#include <stdio.h>
main()
{       char user_passwd[20],encrypt_passwd[20],salt[3];
        printf("Input Encrypt :");
        gets(encrypt_passwd);
        salt[0] = encrypt_passwd[0];
        salt[1] = encrypt_passwd[1];
        salt[2] = 0;
        while (1)
        {       printf("¥nInput passwd :");
                gets(user_passwd);
                if (strcmp(encrypt_passwd,crypt(user_passwd,salt)) == 0)
                {       printf("¥nCongratulation !!!¥n");
                        printf("You Hacked this passwd¥n");
                        printf("Passwd : %s¥n",user_passwd);
                        break;
                }
                printf("Sorry...");
        }
}
```

```
/*********************************************************************/
/*       find_passwd.c                                               */
/*********************************************************************/
/*      This program search ALL posible passwd by dictionary order.*/
/*      Never run This program, because this program will run for  */
/*      longer years than human beings....                         */
/*                                                                 */
/*                              By K.Joe                           */
/*********************************************************************/
#include <stdio.h>
main()
{       char user_passwd[10],encrypt_passwd[20],
             input_passwd[10],input_encrypt[20],
             salt_member[64],tmp,salt[3];
        int  first,second;
        int  salt_mem_index=0;

        printf("Input Encrypted Passwd :");
        gets(input_encrypt);

        salt[0] = input_encrypt[0];
        salt[1] = input_encrypt[1];
        salt[2] = 0;

for(user_passwd[0] = 'a'; user_passwd[0] <= 'z';
        user_passwd[0]++ )
{  user_passwd[1] = 0;
   if (strcmp(input_encrypt,crypt(user_passwd,salt)) == 0)
   {    printf("¥nFound Passwd .... %s¥n",user_passwd);
        exit();
   }
   for(user_passwd[1] = 'a'; user_passwd[1] <= 'z';
                        user_passwd[1]++ )
   {  user_passwd[2] = 0;
      if (strcmp(input_encrypt,crypt(user_passwd,salt)) == 0)
      {    printf("¥nFound Passwd .... %s¥n",user_passwd);
           exit();
      }
      for(user_passwd[2] = 'a'; user_passwd[2] <= 'z';
                        user_passwd[2]++ )
        {  user_passwd[3] = 0;
           if (strcmp(input_encrypt,crypt(user_passwd,salt)) == 0)
           {    printf("¥nFound Passwd .... %s¥n",user_passwd);
                exit();
           }
```

```
for(user_passwd[3] = 'a'; user_passwd[3] <= 'z';
                    user_passwd[3]++ )
{    user_passwd[4] = 0;
     if (strcmp(input_encrypt,crypt(user_passwd,salt)) == 0)
     {    printf("¥nFound Passwd .... %s¥n",user_passwd);
          exit();
     }
     for(user_passwd[4] = 'a'; user_passwd[4] <= 'z';
                    user_passwd[4]++ )
     {    user_passwd[5] = 0;
          if (strcmp(input_encrypt,
                    crypt(user_passwd,salt)) == 0)
          {    printf("¥nFound Passwd .... %s¥n",
                         user_passwd);
               exit();
          }
          for(user_passwd[5] = 'a'; user_passwd[5] <= 'z';
                    user_passwd[5]++ )
          {    user_passwd[6] = 0;
               if (strcmp(input_encrypt,
                         crypt(user_passwd,salt)) == 0)
               {    printf("¥nFound Passwd .... %s¥n",
                              user_passwd);
                    exit();
               }
               for(user_passwd[6] = 'a';
                    user_passwd[6] <= 'z';
                         user_passwd[6]++ )
               {    user_passwd[7] = 0;
                    if (strcmp(input_encrypt,
                         crypt(user_passwd,salt)) == 0)
                    {    printf("¥nFound Passwd .... %s¥n",
                              user_passwd);
                         exit();
                    }
                    for(user_passwd[7] = 'a';
                         user_passwd[7] <= 'z';
                              user_passwd[7]++ )
                    {   user_passwd[8] = 0;
                        if (strcmp(input_encrypt,
                            crypt(user_passwd,salt)) == 0)
                        { printf("¥nFound Passwd .... %s¥n",
                              user_passwd);
                            exit();
```

}}}}}}}}}}

第２章　　ｃｒｙｐｔの解析

このように暗号解読に膨大な計算量を必要とするｃｒｙｐｔのアルゴリズムはどのようになっているのだろうか？

一般にＵｎｉｘのｃｒｙｐｔはＤＥＳを使っていることは知られているが、詳細な資料が見あたらないので解析してみることにした。

ａｄｂ（Ｕｎｉｘの機械語のためのデバッガ）を使って逆アセンブルした結果を次に上げる。

（ＶＡＸ／Ｕｌｔｒｉｘにて逆アセンブル）

```
_encrypt:       e00
76:             subl2   $4,sp
79:             clrf    r10
7b:             movl    4(ap),r0
7f:             cvtbl   mcount+34[r10],r1
87:             decl    r1
89:             movb    (r0)[r1],1118[r10]
92:             aoblss  $40,r10,7b
9a:             clrf    fffffffc(fp)
9d:             clrf    r10
9f:             movb    1138[r10],1158[r10]
ac:             aoblss  $20,r10,9f
b0:             clrf    r10
b2:             cvtbl   10e8[r10],r0
ba:             mull3   $30,fffffffc(fp),r1
bf:             addl2   $_errno+3c,r1
c6:             cvtbl   1137[r0],r0
ce:             cvtbl   (r1)[r10],r1
d2:             xorl2   r1,r0
d5:             cvtlb   r0,1198[r10]
dd:             aoblss  $30,r10,b2
e1:             clrf    r10
e3:             mull3   $6,r10,r11
e7:             cvtbl   119d[r11],r0
ef:             ashl    $6,r10,r1
f3:             addl2   $964,r1
fa:             cvtbl   1198[r11],r2
_encrypt+8e:    ashl    $5,r2,r2
_encrypt+92:    cvtbl   1199[r11],r3
_encrypt+9a:    ashl    $3,r3,r3
```

```
_encrypt+9e:    addl2   r3,r2
_encrypt+a1:    cvtbl   119a[r11],r3
_encrypt+a9:    ashl    $2,r3,r3
_encrypt+ad:    addl2   r3,r2
_encrypt+b0:    cvtbl   119b[r11],r3
_encrypt+b8:    addl2   r3,r3
_encrypt+bb:    addl2   r3,r2
_encrypt+be:    cvtbl   119c[r11],r3
_encrypt+c6:    ashl    $0,r3,r3
_encrypt+ca:    addl2   r3,r2
_encrypt+cd:    addl2   r2,r1
_encrypt+d0:    ashl    $4,r0,r0
_encrypt+d4:    cvtbl   (r1)[r0],r9
_encrypt+d8:    ashl    $2,r10,r0
_encrypt+dc:    movl    r0,r11
_encrypt+df:    extzv   $3,$1,r9,r0
_encrypt+e4:    cvtlb   r0,1178[r11]
_encrypt+ec:    extzv   $2,$1,r9,r0
_encrypt+f1:    cvtlb   r0,1179[r11]
_encrypt+f9:    extzv   $1,$1,r9,r0
_encrypt+fe:    cvtlb   r0,117a[r11]
17a:            ashl    $0,r9,r0
17e:            bicl2   $fffffffe,r0
185:            cvtlb   r0,117b[r11]
18d:            incl    r10
18f:            cmpl    r10,$8
192:            bgeq    197
194:            brw     e3
197:            clrf    r10
199:            cvtbl   b64[r10],r0
1a1:            cvtbl   1177[r0],r0
1a9:            cvtbl   1118[r10],r1
1b1:            xorl2   r1,r0
1b4:            cvtlb   r0,1138[r10]
1bc:            aoblss  $20,r10,199
1c0:            clrf    r10
1c2:            movb    1158[r10],1118[r10]
1cf:            aoblss  $20,r10,1c2
1d3:            incl    fffffffc(fp)
1d6:            cmpl    fffffffc(fp),$10
1da:            bgeq    1df
1dc:            brw     9d
1df:            clrf    r10
1e1:            cvtbl   1118[r10],r11
1e9:            movb    1138[r10],1118[r10]
```

```
1f6:            cvtlb   r11,1138[r10]
1fe:            aoblss  $20,r10,1e1
202:            clrf    r10
204:            cvtbl   mcount+74[r10],r0
20c:            movb    1117[r0],*4(ap)[r10]
216:            aoblss  $40,r10,204
21e:            ret
21f:            halt
_crypt:
_crypt:         f00
_crypt+2:       subl2   $8,sp
_crypt+5:       clrf    r11
_crypt+7:       clrb    11c8[r11]
_crypt+e:       aoblss  $42,r11,_crypt+7
_crypt+16:      clrf    r11
_crypt+18:      brb     _crypt+4d
_crypt+1a:      cmpl    r11,$40
_crypt+21:      bgeq    _crypt+53
_crypt+23:      clrf    r10
_crypt+25:      subl3   r10,$6,r0
_crypt+29:      mnegl   r0,r0
_crypt+2c:      ashl    r0,r9,r0
_crypt+30:      bicl2   $fffffffe,r0
_crypt+37:      cvtlb   r0,11c8[r11]
_crypt+3f:      incl    r10
_crypt+41:      incl    r11
_crypt+43:      cmpl    r10,$7
_crypt+46:      blss    _crypt+25
_crypt+48:      incl    r11
_crypt+4a:      incl    4(ap)
_crypt+4d:      cvtbl   *4(ap),r9
_crypt+51:      bneq    _crypt+1a
_crypt+53:      clrf    r11
_crypt+55:      cvtbl   mcount+b4[r11],r0
_crypt+5d:      movb    11c7[r0],_errno+4[r11]
_crypt+6a:      cvtbl   mcount+d0[r11],r0
_crypt+72:      movb    11c7[r0],_errno+20[r11]
_crypt+7f:      aoblss  $1c,r11,_crypt+55
_crypt+83:      clrf    r11
_crypt+85:      clrf    r8
_crypt+87:      brb     _crypt+d1
_crypt+89:      cvtbl   _errno+4,fffffffc(fp)
_crypt+91:      clrf    r10
_crypt+93:      movb    _errno+5[r10],_errno+4[r10]
_crypt+a0:      aoblss  $1b,r10,_crypt+93
```

```
_crypt+a4:      cvtlb   fffffffc(fp),_errno+1f
_crypt+ac:      cvtbl   _errno+20,fffffffc(fp)
_crypt+b4:      clrf    r10
_crypt+b6:      movb    _errno+21[r10],_errno+20[r10]
_crypt+c3:      aoblss  $1b,r10,_crypt+b6
_crypt+c7:      cvtlb   fffffffc(fp),_errno+3b
_crypt+cf:      incl    r8
_crypt+d1:      movl    r8,r0
_crypt+d4:      cvtbl   mcount+ec[r11],r1
_crypt+dc:      cmpl    r0,r1
_crypt+df:      blss    _crypt+89
_crypt+e1:      clrf    r10
_crypt+e3:      cvtbl   mcount+fc[r10],r0
_crypt+eb:      mull3   $30,r11,r1
_crypt+ef:      addl2   $_errno+3c,r1
_crypt+f6:      movb    _errno+3[r0],(r1)[r10]
31f:            cvtbl   91c[r10],r0
327:            mull3   $30,r11,r1
32b:            addl2   $_errno+3c,r1
332:            addl2   r10,r1
335:            movb    _errno+3[r0],18(r1)
33e:            incl    r10
340:            cmpl    r10,$18
343:            blss    _crypt+e3
345:            incl    r11
347:            cmpl    r11,$10
34a:            bgeq    34f
34c:            brw     _crypt+85
34f:            clrf    r11
351:            clrb    11c8[r11]
358:            aoblss  $42,r11,351
360:            clrf    r11
362:            movb    934[r11],10e8[r11]
36f:            aoblss  $30,r11,362
373:            clrf    r11
375:            cvtbl   *8(ap),r9
379:            incl    8(ap)
37c:            cvtlb   r9,120c[r11]
384:            cmpl    r9,$5a
38b:            bleq    390
38d:            subl2   $6,r9
390:            cmpl    r9,$39
393:            bleq    398
395:            subl2   $7,r9
398:            subl2   $2e,r9
```

```
39b:    clrf    r10
39d:    mnegl   r10,r0
3a0:    ashl    r0,r9,r0
3a4:    blbc    r0,3e2
3a7:    mull3   $6,r11,r0
3ab:    addl2   r10,r0
3ae:    cvtbl   10e8[r0],fffffff8(fp)
3b7:    mull3   $6,r11,r0
3bb:    addl2   r10,r0
3be:    mull3   $6,r11,r1
3c2:    addl2   r10,r1
3c5:    movb    1100[r0],10e8[r1]
3d2:    mull3   $6,r11,r0
3d6:    addl2   r10,r0
3d9:    cvtlb   fffffff8(fp),1100[r0]
3e2:    incl    r10
3e4:    cmpl    r10,$6
3e7:    blss    39d
3e9:    incl    r11
3eb:    cmpl    r11,$2
3ee:    blss    375
3f0:    clrf    r11
3f2:    pushl   $0
3f4:    pushaf  11c8
3fa:    calls   $2,_encrypt
401:    aoblss  $19,r11,3f2
405:    clrf    r11
407:    clrf    r9
409:    clrf    r10
40b:    ashl    $1,r9,r9
40f:    mull3   $6,r11,r0
413:    addl2   r10,r0
416:    cvtbl   11c8[r0],r0
41e:    bisl2   r0,r9
421:    aoblss  $6,r10,40b
425:    addl2   $2e,r9
428:    cmpl    r9,$39
42b:    bleq    430
42d:    addl2   $7,r9
430:    cmpl    r9,$5a
437:    bleq    43c
439:    addl2   $6,r9
43c:    cvtlb   r9,120e[r11]
444:    incl    r11
446:    cmpl    r11,$b
```

```
449:            blss    407
44b:            clrb    120e[r11]
452:            tstb    120d
458:            bneq    465
45a:            movb    120c,120d
465:            movaf   120c,r0
46c:            ret
```

もちろん、この部分以外にもデータ・テーブルがあることは言うまでもない。

アセンブラではアルゴリズムを理解しにくいのでＣ言語にコンバージョンを行った。

以下がそのプログラム及びデータ・テーブルである。

```
/*********************************************************************/
/*      crypt.c                                                      */
/*********************************************************************/
/*      This program does same work as unix/crypt                    */
/*      with easy to understand for person!                          */
/*                                                                   */
/*                      By K.Joe                                     */
/*********************************************************************/

#include <stdio.h>
#include "crypt.h"

main()
{
char    passwd[10],salt[3];
char    *crypt();

        while (1)
        {       printf("¥nInput Passwd :");
                gets(passwd);
                printf("¥nInput Salt :");
                gets(salt);
                crypt(passwd,salt);
                printf("¥nCrypt is : %s¥n",result_pass);
        }
}

char *crypt(passwd,salt)
char passwd[],salt[];
{
int     i,j,k;

/*入力されたパスワードの８文字をそれぞれ６、５、４、３、２、１、０*/
/*ビットの順番で配列pdataにストア                                   */
        i = -1;
        while ((++i < 64)&&(k = passwd[i/8]))
                for (j = 0; j <7; j++)
                        pdata[i++] = (k << (j-6)) & 1;
/*鍵の作成*/
```

```
/*選択転置*/
        for(i=0;i<0x1c;i++)
        {       C0[i] = pdata[PC1a[i]-1];
                D0[i] = pdata[PC1b[i]-1];
        }

        for (i=0;i<16;i++)
        {       for(j=0;j<LS[i];j++)
                {       rotation(C0,28,1);/*シフト*/
                        rotation(D0,28,1);
                }
                for (j=0;j<0x18;j++)
                {       k = PC2a[j];    /*選択転置*/
                        ((char *)((int)KA + i*48))[j] =
                                (k <= 28) ? C0[k-1] : D0[k-28-1];
                        k = PC2b[j];
                        ((char *)((int)KA + i*48 + j))[0x18] =
                                (k <= 28) ? C0[k-1] : D0[k-28-1];
                }
        }       /*１６個の鍵ができる*/

        for (i=0;i<0x42;i++)
                pdata[i] = 0;

        for (i=0;i<0x30;i++)
                Esub[i] = E[i];

        for (i=0;i<2;i++)       /*saltにより拡大転置を変更*/
        {       salt_key[i] = k = salt[i];
                if (k > 0x5a)
                        k -= 6;
                if (k > 0x39)
                        k -= 7;
                k -= 0x2e;
                for (j=0;j<6;j++)
                        if ((k >> j) & 1)
                                swap(&Esub[6*i+j],&Esub[6*i+j+24]);
        }
        for (i= 0;i<0x19;i++)
               encrypt(0,pdata);/*ＤＥＳのアルゴリズムによる*/
                                /*暗号生成を２５回繰り返す  */

        for (i=0;i<0xb;i++)     /*印字可能に変換*/
        {       k = 0;
                for (j=0;j<6;j++)
```

```
                        k =  pdata[6*i+j] | (k <<= 1);
                k += 0x2e;
                if (k > 0x39)
                        k += 7;
                if (k > 0x5a)
                        k += 6;
                cdata[i] = k;
        }
        cdata[i] = 0;

        if (salt_key[1] == 0)
                salt_key[1] = salt_key[0];
        strcpy(result_pass,salt_key);
        strcat(result_pass,cdata);
        return result_pass;
}


encrypt(a,block)        /*ＤＥＳのアルゴリズム*/
int     a;
char    block[];
{       int     i,j,k,m;
        char    LORO[0x40],LRsub[0x20],W1[0x20],
                W2[0x30];

        for (i=0;i<64;i++)  /*初期転置*/
                LORO[i] = block[ IP[i]-1 ];

        /*１６個の鍵により転置と換字を繰り返す*/
        for (m=0;m<16;m++)
        {       for (i=0;i<32;i++)
                        LRsub[i] = LORO[32+i];
        /*修正された拡大転置を用いていることに注意*/
                for (i=0;i<0x30;i++)
                        W2[i] = LORO[Esub[i]+32-1]
                                ^ ((char *)(((int)KA) + 48 * m))[i];
                for (i=0;i<8;i++)
                {       k =  S[   W2[6*i+4]
                                + W2[6*i+3] * beki2(1)
                                + W2[6*i+2] * beki2(2)
                                + W2[6*i+1] * beki2(3)
                                + W2[6*i+5] * beki2(4)
                                + W2[6*i]   * beki2(5)
                                + i         * beki2(6)  ] ;
                        for (j=0;j<4;j++)
```

```
                        if (k & beki2(3-j))
                                W1[i*4+j] = 1;
                        else    W1[i*4+j] = 0;
                }
                for (i = 0;i<32;i++)
                        LORO[i+0x20] = W1[P[i]-1] ^ LORO[i];

                for (i=0;i<32;i++)
                        LORO[i] = LRsub[i];
        }

        for (i=0;i<32;i++)
                swap(&LORO[i],&LORO[i+32]);

        for (i=0;i<64;i++)      /*逆初期転置*/
                block[i] = LORO[IPD[i]-1];
}


rotation(array,dim,direction)
char    array[];
int     dim,direction;
{
        int     i;
        char    tmp;

        if (direction == 1)
        {       tmp = array[0];
                for (i=0;i<dim-1;i++)
                        array[i] = array[i+1];
                array[dim-1] = tmp;
        }
        else if (direction == -1)
        {       tmp = array[dim-1];
                for (i=(dim-1);i>=0;i--)
                        array[i] = array[i-1];
                array[0] = tmp;
        }
}

swap (a,b)
char    *a,*b;
{
        char    c;
```

```
        c = *a;
        *a = *b;
        *b = c;
}

beki2(n)
{
        return (n == 0)? 1 : 2*beki2(n-1);
}


/*****************************************************************/
/*      Crypt.h                                                  */
/*****************************************************************/

char    result_pass[14],
        Esub[0x30],
        pdata[0x42],salt_key[2],cdata[0xb],C0[0x1c],D0[0x1c],
        KA[768],/*鍵*/
        IP[0x40]/*初期転置規約*/
= {072,062,052,042,032,022,012,02,074,064,054,044,034,024,014,04,076,066,056,046,036,
026,016,06,0100,070,060,050,040,030,020,010,071,061,051,041,031,021,011,01,073,063,
053,043,033,023,013,03,075,065,055,045,035,025,015,05,077,067,057,047,037,027,017,07
        },
        IPD[0x40]/*最終転置規約*/
= {050,010,060,020,070,030,0100,040,047,07,057,017,067,027,077,037,046,06,056,016,066
,026,076,036,045,05,055,015,065,025,075,035,044,04,054,014,064,024,074,034,043,03,053
,013,063,023,073,033,042,02,052,012,062,022,072,032,041,01,051,011,061,021,071,031
        },
        PC1a[0x1c]/*選択転置規約*/
= {071,061,051,041,031,021,011,01,072,062,052,042,032,022,012,02,
   073,063,053,043,033,023,013,03,074,064,054,044
        },
        PC1b[0x1c]
= {077,067,057,047,037,027,017,07,076,066,056,046,036,026,016,06,
   075,065,055,045,035,025,015,05,034,024,014,04
        },
        LS[0x10]/*シフト数*/
= {01,01,02,02,02,02,02,02,01,02,02,02,02,02,02,01
                },
        PC2a[0x18]
= {016,021,013,030,01,05,03,034,017,06,025,012,027,023,014,04,
   032,010,020,07,033,024,015,02
        },
        PC2b[0x18]
```

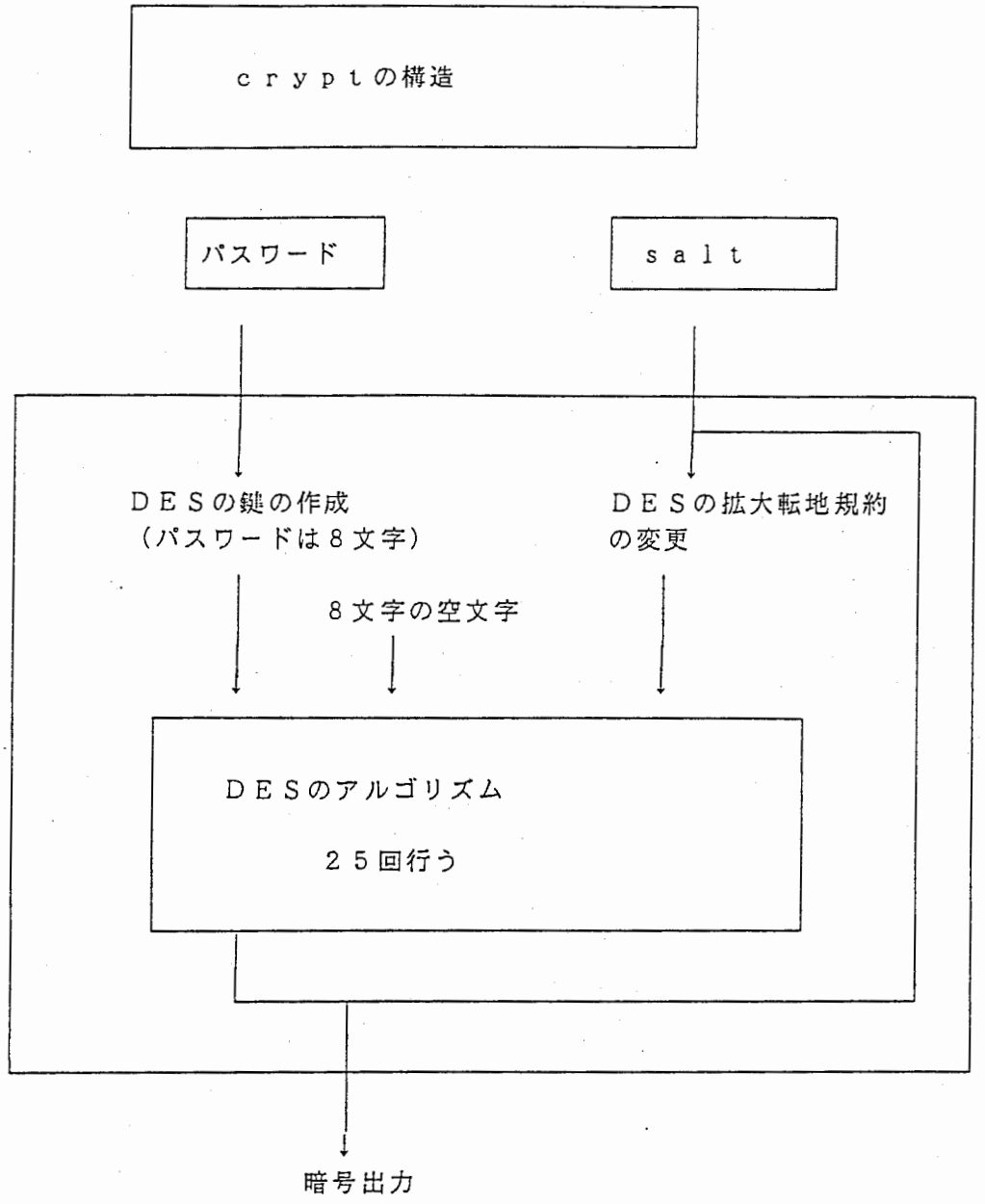
```
= {051,064,037,045,057,067,036,050,063,055,041,060,054,061,047,070,
   042,065,056,052,062,044,035,040
        },
        E[0x30]/*拡大転置規約*/
= {040,01,02,03,04,05,04,05,06,07,010,011,010,011,012,013,014,015,014,015,016,017,020
,021,020,021,022,023,024,025,024,025,026,027,030,031,030,031,032,033,034,035,034,035,
036,037,040,01
        },
        S[0x200] = {   /*選択関数規約*/
016,004,015,001,002,017,013,010,003,012,006,014,005,011,000,007,000,017,007,004,016,0
02,015,001,012,006,014,013,011,005,003,010,004,001,016,010,015,006,002,013,017,014,01
1,007,003,012,005,000,017,014,010,002,004,011,001,007,005,013,003,016,012,000,006,015
,017,001,010,016,006,013,003,004,011,007,002,015,014,000,005,012,003,015,004,007,017,
002,010,016,014,000,001,012,006,011,013,005,000,016,007,013,012,004,015,001,005,010,0
14,006,011,003,002,017,015,010,012,001,003,017,004,002,013,006,007,014,000,005,016,01
1,012,000,011,016,006,003,017,005,001,015,014,007,013,004,002,010,015,007,000,011,003
,004,006,012,002,010,005,016,014,013,017,001,015,006,004,011,010,017,003,000,013,001,
002,014,005,012,016,007,001,012,015,000,006,011,010,007,004,017,016,003,
013,005,002,014,007,015,016,003,000,006,011,012,001,002,010,005,013,014,004,017,015,0
10,013,005,006,017,000,003,004,007,002,014,001,012,016,011,012,006,011,000,014,013,00
7,015,017,001,003,016,005,002,010,004,003,017,000,006,012,001,015,010,011,004,005,013
,014,007,002,016,002,014,004,001,007,012,013,006,010,005,003,017,015,000,016,011,016,
013,002,014,004,007,015,001,005,000,017,012,003,011,010,006,004,002,001,013,012,015,0
07,010,017,011,014,005,006,003,000,016,013,010,014,007,001,016,002,015,006,017,000,01
1,012,004,005,003,014,001,012,017,011,002,006,010,000,015,003,004,016,007,005,013,012
,017,004,002,007,014,011,005,006,001,015,016,000,013,003,010,011,016,017,005,002,010,
014,003,007,000,004,012,001,015,013,006,004,003,002,014,011,005,017,012,013,016,001,0
07,006,000,010,015,004,013,002,016,017,000,010,015,003,014,011,007,005,012,006,001,01
5,000,013,007,004,011,001,012,016,003,005,014,002,017,010,006,001,004,013,015,014,003
,007,016,012,017,006,010,000,005,011,002,006,013,015,010,001,004,012,007,011,005,000,
017,016,002,003,014,015,002,010,004,006,017,013,001,012,011,003,016,005,000,014,007,0
01,017,015,010,012,003,007,004,014,005,006,013,000,016,011,002,007,013,004,001,011,01
4,016,002,000,006,012,015,017,003,005,010,002,001,016,007,004,012,010,015,017,014,011
,000,003,005,006,013
                        },
        P[0x20]  /*出力転置規約*/
={  020,007,024,025,035,014,034,021,001,017,027,032,005,022,037,012,002,010,030,016,0
40,033,003,011,023,015,036,006,026,013,004,031 };
```

ｃｒｙｐｔがＤＥＳアルゴリズムを用いていることは明かである。
ＤＥＳ（Data Encryption Standard）アルゴリズムは米国において標準化された暗号規格の一つで、１ブロック６４ビットの通信文を６４ビットの鍵を用いることにより（正確には５６ビットの鍵と８ビットのパリティ）暗号文を作成したりその復元を行ったりするものである。通常、ハード・ウェアで使われることが多く、ソフト・ウェアでインプリメントすると、その計算量の多さ故に生成・解読に時間がかかることが知られている。ＤＥＳのアルゴリズムの詳細についてはここでは述べない。

さて、ｃｒｙｐｔのアルゴリズムだが、面白いことにｃｒｙｐｔでは原文を使わない。あたかも原文であるかのように思えるパスワードは、実は鍵として使われているのだ。（ここにＵｎｉｘのパスワードは最初の８文字しか有効でない理由がある。８文字＝６４ビット）原文は６４ビットの０として与えられる。ＤＥＳでは、このような場合にも６４ビットの暗号文を出力するように作成されている。ｃｒｙｐｔはこの暗号文を原文と見なしてさらに暗号文を生成する。この操作が２５回繰り返され、目的の暗号文が作られる。では、もう１つのｃｒｙｐｔの引数であるｓａｌｔは何をするのだろうか？ＤＥＳの暗号生成・復元がハード・ウェアでインプリメントされていることを思いだして欲しい。ＤＥＳのままではこのようなハード・ウェアを利用することによりパスワードが破られてしまうかも知れないのだ。ＤＥＳのアルゴリズムの中心をなすのは原文６４ビットを２つに分け、それぞれの３２ビットに対し拡大転置をほどこし４８ビットに拡張してから鍵の４８ビットとＸＯＲをとって選択関数を使用するところであるが、ｃｒｙｐｔでは、この拡大転置規約にｓａｌｔを使って変更を行っている。
次に示すように、ｓａｌｔの値に従って拡大転置の列の前半と後半とで交換を行っている。

```
for (i=0;i<2;i++)        /*saltにより拡大転置を変更*/
{       salt_key[i] = k = salt[i];
        if (k > 0x5a)
                k -= 6;
        if (k > 0x39)
                k -= 7;
        k -= 0x2e;
        for (j=0;j<6;j++)
                if ((k >> j) & 1)
                        swap(&Esub[6*i+j],&Esub[6*i+j+24]);
```

このようにして作られた暗号文１１文字（６４ビットを６ビットずつ取ってきて１文字としている）にｓａｌｔをあわせた１３文字をｃｒｙｐｔは返す。
以上がｃｒｙｐｔのアルゴリズムである。


ｃｒｙｐｔの構造
パスワード
ｓａｌｔ
ＤＥＳの鍵の作成
（パスワードは８文字）
ＤＥＳの拡大転地規約
の変更
８文字の空文字
ＤＥＳのアルゴリズム
２５回行う
暗号出力

第３章　　パスワード解読ルーチンの可能性

ｃｒｙｐｔの全容が明らかになったところで、次にパスワード解読ルーチンの可能性について考察してみよう。

Ｕｎｉｘのパスワード解読ルーチンとは以下のことを行うルーチンである。

『／ｅｔｃ／ｐａｓｓｗｄのパスワード・フィールドの暗号から元のパスワードを復元する。』

ここで注意することは、パスワード・フィールドの先頭２文字はｓａｌｔになっていることだ。言い変えれば次のようになる。

『パスワード・フィールドの先頭２文字をｓａｌｔと考え、残り１１文字を暗号として、原文が空文字（０が６４ビット続く）であるようなＤＥＳの鍵を見つける。』

これは即ちＤＥＳ破りの特殊なケースに等価である。なぜなら、ｃｒｙｐｔ中の変更された拡大転置規約はｓａｌｔが分かれば一意に定まるからだ。

ＤＥＳ破りについてはM．Ｅ．Ｈｅｌｌｍａｎ等により各種の問題点が指摘されているが、依然として高性能のパラレル・プロセッサが開発されなければ計算安全性が保証されているようである。

ＤＥＳの鍵とは６４ビットの列とそれを１５通りにシフトした計１６の配列からなっているが、このシフトの方法は一意であるから最後の１つが決定すればもとの鍵も決定する。

```
for (m=0;m<16;m++)
{  for (i=0;i<32;i++)
      LRsub[i] = LORO[32+i];
   for (i=0;i<0x30;i++)                                    (E)
      W2[i] = LORO[Esub[i]+32-1] ^ ((char *)(((int)KA) + 48 * m))[i];
   for (i=0;i<8;i++)
   {  k =  S[ W2[6*i+4] + W2[6*i+3]*beki2(1) + W2[6*i+2]*beki2(2) +W2[6*i+1]*beki2(3)
           + W2[6*i+5]*beki2(4) + W2[6*i]*beki2(5)+ i*beki2(6)  ] ;   (D)
      for (j=0;j<4;j++)                                    (C)
        if (k & beki2(3-j))
           W1[i*4+j] = 1;
        else    W1[i*4+j] = 0;
   }
   for (i = 0;i<32;i++)
        LORO[i+0x20] = W1[P[i]-1] ^ LORO[i];      (B)

   for (i=0;i<32;i++)
        LORO[i] = LRsub[i];       (A)
        }
```

このプログラムにおいて最後の鍵は ((char *)(int)KA+48*15)[i] (0<=i<16) に相当する。
それではプログラムを逆に追ってみよう。
まず、（A）において、暗号は LORO にもとまっているので、LRsub は決定する。
次に、（B）において LORO の後半３２要素はもとまっているが LORO の前半３２要素は（A）で置き変わる前の値であるから一意に決まらない。即ち、要素数３２の配列 W1 は各要素が０または１の値となる。即ち、2^{32}とおりの W1 が考えられる。
さらにこの W1 を用いて（C）で K が一意に決定される。
（D）においては、この K に対して、選択関数Ｓの逆関数をほどこした値の各ビットに対応して W2 がもとまる。しかしながら、このＳは多値関数であり、逆関数として使うと８とおりの値が出てくる。（ここがＤＥＳの中核をなすところである）
W2 をすべてもとめるためにはこの操作を８回繰り返さなければならないので、結局考えられる場合の数は$2^{32} * 8^8 = 2^{56}$となる。
（E）では、このようにしてもとまった W2 と LORO から目的の鍵が一意に算出される。
つまり、第１６番目の鍵を算出するには2^{56}の可能性をすべて調べなければならないことになる。
（この2^{56}はＤＥＳのもともとの鍵６４ビットからパリティの８ビットを引いた５６ビットの表しうるパターンと一致する。）

従って、ｃｒｙｐｔの逆関数を作ることは実際上不可能である。

補足

本資料ではＵｎｉｘのｃｒｙｐｔルーチンの解析とその逆関数の可能性について述べているが、単にシステムを破ることのみを目的とするだけならば、いろいろな方法がある。

暗号生成のルーチンがいくらよくできていても肝心のパスワード自体が安直な物であれば，これだけ安全なはずのｃｒｙｐｔは意味をなさないのだから....

参考文献

Ｖａｘ／Ｕｌｔｒｉｘ　Ｍａｎｕａｌ

謝辞

有益な御討論をして頂いたＡＴＲ視聴覚機構研究所の諸氏に感謝致します。