Blowfish Study n' Reverse

{*Written by* **Evilcry**}
{evilcry@reteam.org}

[RET]
[http://www.reteam.org]
[http://evilcry.altervista.org]
[irc.azzurranet.org #crack-it]

{**Difficulty:** Advanced}

INTRODUCTION

In this tutorial, we will study the Blowfish cipher. We will explore how Blowfish works and see if there are some weaknesses. Obviously we will see Blowfish not only from a 'cryptographer point of view' but also and *mainly* from a reverser point. I tried to make this a detailed analysis. My intent is to make you able to quickly identify any code that implements blowfish.

TOOLS

IDA
OllyDbg
Good Basis of Cryptography

TARGET LOCATION

http://www.google.com

ABOUT THE TARGET

I have used the Raskgme1 crackme for an example.

Blowfish OVERVIEW

Blowfish is a cipher based on blocks which also uses a secret-simmetric key (so there is only a single key, used both for encryption and decryption). Blowfish is based on a Feistel-Network, for people that do not know what this means: A Feistel iterates a specific function a certain number of times and each cycle is called a round. Usually blowfish has a round number of 16.

In addition to the Feistel Network, Blowfish bases its action on 4 SBoxes (could be considered arrays), indipendant from the keys. There is also a OneWay function F() which is NOT reversible.

Previously we said that we're in front of a block cipher. Each block is 64 bits long, while the key can be as long as 448 bits. Until this moment there known effective attacks techniques. Blowfish is also really fast. I have a little curiosity with this cipher as it has some good features. Blowfish was also implemented in some microchips, one of which is the 68HC11.

Note for the attackers: the Sbox used "results to be resistant to attacks of differential cryptoanalysis". After this introduction, we can begin to study this algorithym deeply. It's important to say that the general fuctioning mechanism will be fundamental to recognizing blowfish in a protection scheme. The encryption procedure can be divided in two steps:

Expansion Step (or initialization function, this is the most complex phase).

In which the key, long max 448 bits, is "transformed" into an array of subkeys for a total of 4168 bytes

Encryption step:

The encryption process happens in 16 rounds and uses XOR and addition operations. The size used is the Dword. All this, is placed into some indicized arrays. From the key, are extracted two kinds of arrays: P-Array and S-Box. The P-Array is composed of 18 elements and each subkey is 4 bytes long:

P1,P2,P3.....P18

While the S-Box are 4, each of them is composed by 256 subkeys at 32 bits. Let's see now what happens in detail, in the expansion phase:

1. P-Array and the 4 S-Boxes are initialized, the most important thing to say, is that, the 4 Sboxes all begin with constant values.

2. There is a XOR between P-Array and the chosen key.

3. Repeat this procedure until all subkeys of P-array are "codified" into the form:

P-array= {xL1,xR1,xL2,xR2,.....,xL18,xR18}

4. With the "new" P-array, begin the substitution work of the S-Box. An important note: these S-Boxes are modified by each round that happens.

Now we know a bit about how Blowfish works in the encryption phase we can say that Blowfish bases its work on randomized sboxes. Indeed, the algorithm builds some big lookup tables with

pseudorandom data. The 'PSEUDO' casuality depends heavily on the key itself. As you can see this procedure is a bit complex. If we try to analyze the "implications" that coexist between sboxes obtained and used keys we will see that these relations are really complex!

The direct consequence of all this is that Blowfish seems to be not attackable (except in some cases) by differential and linear cryptoanalysis. The only well known attacks to Blowfish work essentially on the use of weak-keys, in which there is at minimum a collision into one of the four sboxes.

Now let's take a look at a generic protection routine that uses Blowfish:

- **Blowfish_Initialization( )**
- **Blowfish_encrypt( )**
- **Blowfish_decrypt( )**

An attack for Blowfish (addendum)

This section is not fundamentally important to understand the future "reverse approach" to Blowfish, but can be interesting for people who want to know something more on blowfish cryptoanalysis. The most successful attack to blowfish was the Vaundenay attack, which demonstrated the presence of some weak keys in the cipher. The condition of "weak key" occours when there is AT LEAST ONE collision into one of the 4 Sboxes. This condition can be expressed as:

Sbox1(X)=Sbox1(Y)

Where X and Y are two different bytes. Now if the attacker knows all Sboxes entrances (not P1 P2 etc.), or even better, knows the key described by the F() function, he can see the following relation:

d= X xor Y

"d" which is the key (in this case is obviously weak). As a result the P-arrays (P1 P2 etc.) are generated from the weak key, so if we assume at least one collision for S1, the probability that this collision happens is 2^(-7). Let's consider also that we're in front of a Feistel Network, so we can expect that things will grow round by round!. Indeed if we repeat the procedure (d X xor Y) three times, we will obtain 2^(-7)^3 in other words 2^(-21)!. Now we can suddenly perform a "chosen plaintext" attack, in which we have:

With: xor (0000d000), using a chosen couple (c,c') we land at xor(d000xyzt). c' can be considerated divided into two groups

c(L,R). Now if we apply this to F(), we will obtain the following equation:

F(L xor P10) xor F(L xor P10 xor [d000])=[xyzt]

And finally with a bruteforce of 2^32 we will obtain P10, that verifies the previous equation.

An analytic/reverse approach to Blowfish

Wow, we are ready to start a reversing session on Blowfish! First of all we will analyze a little portion of Raskgme1 crackme in which is implemented Blowfish. This crackme implements also bignums and other things that we will not consider.

```
004018A2 push 0040A0B8 ; ptr to the dtKey
004018A7 push ecx
004018A8 call 00401130 ; Blowfish_Init()
```

Let's see now what happens inside our initialization function:

```
00401130 push ecx

00401134 mov esi, dword ptr [esp+14]

00401138 push edi

00401139 mov eax, 00408118

0040113E lea ecx, dword ptr [esi+48]

00401141 mov edx, 00000100 ; Number of iterations (100h=256d)

00401146 mov edi, dword ptr [eax]

00401148 add eax, 00000004

0040114B mov dword ptr [ecx], edi ;copy in ecx (array), edi

0040114D add ecx, 00000004 ; Add a dword to the array index

00401150 dec edx ;decrase the counter

00401151 jne 00401146 ; Repeat this cycle 256 times

00401153 cmp eax, 00409118

00401158 jl 00401141 ; Repeat the previous cycle 4 times
```

This piece of code creates an array of 256 elements! and at each cycle a dword is copyed. We also have an external cycle that repeats the internal cycle 4 times. If you studied with attention

the theoretical part, you should have no problems in understandind what this routine does ;). Let's translate it to C:

```
int i, j, k;

unsigned long data, datal, datar;

for (i = 0; i < 4; i++) {

for (j = 0; j < 256; j++)

ctx->S[i][j] = ORIG_S[i][j]; }
```

Surely you will demand what ctx is?, easly from the ->, we understand that ctx is a struct, but what should this struct contain?:

```
typedef struct {

unsigned long P[16 + 2];

unsigned long S[4][256]; } BLOWFISH_CTX;
```

Syntatically it's a Blowfish context, inside which we can find the definition of the P-array and of the Sboxes. By returning to the previous for cycle, we can see that inside the 4 Sboxes are iterated. Each of them composed of 256 subkeys at 32 bits. In other words we're in the expansion phase.

```
0040115A mov ebp, dword ptr [esp+20] ;| this will be important because will contain xL and xR

0040115E mov edx, dword ptr [esp+1C] ;| or better contains "Data"

00401162 mov edi, 004080D0 ;puts a value in edi, its not important to know what it is

00401167 xor eax, eax ;

00401169 sub edi, esi

0040116B mov [esp+10], 00000012 ; 12h will work as index (12h=18d)

00401173 xor ecx, ecx ; Null the index (j=0) first to begin a cycle with the algorithm itself

00401175 mov [esp+20], 00000004 ;Other index

0040117D xor ebx, ebx

0040117F mov bl, byte ptr [eax+edx]

00401182 8 shl ecx, 08 ; data = (data << 8) OR key[j]

00401185 or ecx, ebx ;
```

```
00401187 inc eax ; j = j + 1;

00401188 cmp eax, ebp ;| if (j >= keyLen) ----> keyLen, represents the length of
the key

0040118A jl 0040118E

0040118C xor eax, eax ;J=0

0040118E mov ebx, dword ptr [esp+20]

00401192 dec ebx ; Ebx starts from 4 and lands to 0

00401193 mov dword ptr [esp+20], ebx

00401197 jne 0040117D ;Repeat until is 0

00401199 mov ebx, dword ptr [edi+esi]

0040119C add esi, 00000004

0040119F xor ebx, ecx ; Xor two values contained in ecx and ebx (ctx->P[i] =
ORIG_P[i] ^ data; )

004011A1 mov ecx, dword ptr [esp+10] ; In ecx the value 18

004011A5 mov dword ptr [esi-04], ebx

004011A8 dec ecx

004011A9 mov dword ptr [esp+10], ecx

004011AD jne 00401173 ;repeat the two 2 cycles, 18 times
```

This piece of code needs a little explanation. We have two cycles, one is iterated 4 times, the other 18 times (what do you remember about the value 18?...yes the P-array ;). So let's translate this piece of code to better understand how it works:

```
for (i = 0; i < N + 2; ++i) {

data = 0x00000000;

for (k = 0; k < 4; ++k) {

data = (data << 8) | key[j];

j = j + 1;

if (j >= keyLen) j = 0;

}

ctx->P[i] = ORIG_P[i] ^ data; } // ctx AKA Blowfish Context
```

The most intersting piece for us, is the last line, where we can see an Xor between the data and an element of an array (P-array). This is the most interesting operation between Key and P-array. If you do not remenber well, jump back to **2.**, and everything will be clear!

In the next piece of code, we will see the encrypting function at work:

```
004011BD mov esi, ebx

004011BF mov edi, 00000009

004011C4 lea eax, dword ptr [esp+1C] ;DataR

004011C8 lea ecx, dword ptr [esp+20] ; DataL

004011CC push eax ; Data R (similar to xR, that can be found in the first part)

004011CD push ecx ;Data L (similar to xL, that can be found in the first part)

004011CE push ebx ; Ctx (is the Blowfish Context that previously analysed)

004011CF call 00401000 ; Blowfish_encryption(ctx,&DataL,&DataR ), take three
parameters
```

To better understand what happens here, let's trasform the asm code to C:

```
for (i = 0; i < N + 2; i += 2) {

Blowfish_Encrypt(ctx, &datal, &datar);

ctx->P[i] = datal; // Will be analyzed farther down

ctx->P[i + 1] = datar; // Will be analyzed farther down

}
```

Take note as to the number of iterations of this operation that will continue until all arrays are “expanded” or rather filled with new data. In this case, it's iterated for the same number as the P-array's length. Pay attention to the intersting fact that, when you're into the Blowfish's initialization routine, you fully meet the encrypt function. Now let's analyze the encryption function:

```
00401000 mov eax, dword ptr [esp+08] ;DataL in eax

00401004 mov ecx, dword ptr [esp+0C] ;DataR in ecx
```

```
0040100A mov eax, dword ptr [eax]

0040100C push esi

0040100D mov esi, dword ptr [ecx]

0040100F push edi

00401014 mov [esp+14], 00000010 ; Ready for a FOR Cycle(10h=16d)
```

In this easy piece of code, DataL and DataR are assigned to local variables of our function and registers are set up for a FOR Cycle:

```
0040101E xor eax, dword ptr [ebx] ; left ^= p[0];

00401020 push eax ;parameter that will be used by the call 00401060, let's call it Y

00401021 57 push edi ;parameter that will be used by the call 00401060, let's call it X

00401022 mov ebp, eax

00401024 call 00401060 ;let's call it Getbyte(X,Y)
```

Here the definition of GetByte(X,Y), asm code is not necessary:

```
#define GETBYTE(x, y) (unsigned int)(((x)>>(8*(y)))&255)
```

This is the famous function F, in this case it takes as input CTX and xL, let's continue with encrypt procedure:

```
00401029 mov ecx, dword ptr [esp+1C] ;Insert in ecx 10h, 16d that are the 16 round of the Feistel proc, do you rmember?

0040102D add esp, 00000008

00401030 xor eax, esi

00401032 add ebx, 00000004 ;go to the next dword

00401035 dec ecx ;decrease counter

00401036 mov esi, ebp

00401038 mov dword ptr [esp+14], ecx

0040103C jne 0040101E ;repeat 16 times
```

Hehe our Feistel Network at work!! :)..we are in the heart of encryption function..a thing jumps out at us immediately. The entire process is based on XOR operations, these XORs happen between two pieces of data (each piece being 32 bits).

```
for (i = 0; i < N; ++i) {

Xl = Xl ^ ctx->P[i]; // xor between DataL and the actual value pointed by P-array

Xr = F(ctx, Xl) ^ Xr;

temp = Xl;

Xl = Xr;

Xr = temp; }
```

In the last part of our function we can observe, other XORs

```
Xr = Xr ^ ctx->P[N];

Xl = Xl ^ ctx->P[N + 1];

*xl = Xl;

*xr = Xr;

}
```

And now, our encyption function is finished, we can come back to the initialization routine, and observe what happens immediately after the encryption:

```
004011D4 8B54242C mov edx, dword ptr [esp+2C] ; move the new DataL into edx

004011D8 8B442428 mov eax, dword ptr [esp+28] ; move the new DataR into eax

004011DC 8916 mov dword ptr [esi], edx ; Insert DataL , into ctx->P (that points to P-array)

004011DE 894604 mov dword ptr [esi+04], eax; ; Insert DataR , into the next ctx->P
```

Immediately we understand that the encryption function returns 2 DWORDs (DataL and DataR), that next will go to fill the P-array. This is the situation of P-array, after the entire For Cycle:

P-array {DataL1,DataR1,....,DataL18,DataR18}

Now into blowfish_init(): The encryption function is called two times, we have seen only the first, now let's study the second call:

```
004011F2 mov edi, 00000080 ;counter index is set to 80h (128d)

004011F7 ecx, dword ptr [esp+1C] ;DataR

004011FB lea edx, dword ptr [esp+20] ;DataL
```

```
004011FF push ecx ;DataR

00401200 push edx ;DataL

00401201 push ebx ;context

00401202 call 00401000 ;Blowfish_encrypt()

00401207 mov eax, dword ptr [esp+2C] ;DataL into ctx->S[i][j]

0040120B mov ecx, dword ptr [esp+28] ;DataR ctx->S[i][j+1] (in other words the
next element)
```

As usual, we have DataL and DataR as return values of the encrypt that go into another array..let's understand which array... The first thing that we see, is that For Cycle is set up to 128 (in other words 256/2), this for is placed inside another for (index 4), that i've not inserted.

This should make you suspicious..these two values(4 and 128) are associative of the 4 S-Boxes!!! :), indeed we can see that:

S-box[1] {DataL1,DataR1,....,DataL128,DataR128}

Same thing for the other three S-Boxes.

Wow! and this is really the end of our Encryption Function. This will help you to identify Blowfish with all its variants:) Now let's talk about Blowfish_Decrypt() function:

Blowfish_Decrypt(BLOWFISH_CTX *ctx, unsigned long *xl, unsigned long *xr)

The encryption function takes as input CTX, DataL and DataR, and it returns DataL and DataR decrypted. But you have to pay very big attention, because as you can see, there is a massive use of the XOR operation, and as you should know, to obtain the inverse (in this case decrypt) you have to apply the XOR again. So encrypt() and decrypt() are truly similar!. Essentially you can understand the decrypt() principally for:

```
Xr = Xr ^ ctx->P[1];

Xl = Xl ^ ctx->P[0];
```

Many times in protection schemes, to recognize some famous algorithm you check for some constant values (for example in MDx, TEA, Ghost, etc.). In Blowfish we have some starting values, for P-array:

0x243F6A88L

and for S-box1 the first element is:

0xD1310BA6L

Another useful element to recognize Blowfish scheme, is the number of iterations and how they are identified.

Keygenning of Blowfish

Writing a keygenerator for blowfish is not a hard task. Instead, we only have to use a copy of the various sources around the internet (for example Paul Kocher blowfish.c and blowfish.h). There is nothing of particular to pay attention to with these. Usually we need the key used for the encryption process, for example we could write

```
unsigned char szSetKey[]="BAPCRHJTET5DF4TR5YEWERKTY879432";
```

Many times an XorMask is used that we can code in this way:

```
 unsigned char szxortable[8]={0x6B, 0x74, 0x17, 0x69, 0x65, 0x1D, 0x67, 0x52};
```

Then the Initialization function, that we can code as:

```
BLOWFISH_CTX blowfishctx;

Blowfish_Init(&blowfishctx, &szSetKey, 0x1f);
```

Where the last parameter is the length of our key, and immediately after we have to create a little procedure to insert into DataL and DataR correct data (this does not occur many times)

```
 __asm{

  lea esi,szName

 mov eax,dword ptr [esi]

 mov     dtA,eax

mov eax,dword ptr [esi+4]

 mov     dtB,eax

          }
```

Immediately our Encrypt (obviously case by case there are different implementations, this is one of the most common protection patterns)

```
Blowfish_Encrypt(&blowfishctx, &dtA, &dtB);
```

It's truly important to specify that there are not, unflexible coding patterns, because we can meet a large amount of different cases. Here I have shown you the most easy, general and common

case that you'll meet. The worst case that you can see is the Modified Blowfish, in other words protection schemes that do not implement the canonic blowfish, but a version changed in some aspect. Usually the modifications that you'll meet are not many, here the most common: :

- Altered number of Feistel rounds (common is 32)
- S-boxes have different starting values, this is also an anti-detect system for automated recognition software
- F function is modified and frequently returns S[2][d]-S[1][b])^(S[0][c]+S[3][a]

Hope that the article is clear. I'm here for clarifications! :)

GREETS

Regards to All UIC: Quequero, AndreaGeddon, LonelyWolf, Zairon, LittleLuck, and RET: Devine9, Black-Eye, Big-S, LibX, Aimless..and all others :)