UIC – Università Italiana Cracking

http://quequero.org

Bit Defender 9 Professional Plus Reversing

written by SatUrN

UIC New Year Pack 2 – 01/Jan/2007

BitDefender 9 Professional Plus Reversing (Build 9.5)

BitDefender è un pacchetto che contiene le principali difese contro tutto ciò che gira di maligno nella rete. Contiene antivirus, firewall, antispyware, antispam e controllo parentale.

Il programma è scaricabile dal sito www.bitdefender.com e purtoppo devo anticipare che la versione attualmente disponibile è la 10 (solo in inglese per il momento) e non più la 9.
Per motivi di tempo ho rinunciato a scrivere l'articolo su questa nuova versione, poiché al momento del rilascio ero già a uno stadio troppo avanzato con la 9.
Comunque le versioni vecchie si riescono a trovare a questo indirizzo in 23 lingue differenti:

http://download.bitdefender.com/windows/desktop/professional/final/

L'articolo si baserà sul reversing dell'applicazione per poter bypassare i controlli e creare un keygen in asm che ci permetterà di generare tutti i seriali validi che vogliamo.
Il controllo del seriale è abbastanza complicato perché fa uso dell'algoritmo di cifratura Blowfish in una forma modificata.
Per l'implementazione e la ricerca delle differenze dall'algoritmo standard mi sono appoggiato a un secondo programma chiamato Cryptool.

Visto che l'articolo è rivolto prevalentemente al reversing, tralascerò la descrizione dettagliata del funzionamento di Blowfish, ma ne spiegherò alcuni passaggi man mano che ce ne sarà bisogno.
Un primo passaggio importante da sapere è che Blowfish per eseguire la cifratura si appoggia su una private key (naturalmente sconosciuta) che dovremmo scovare nel codice e riutilizzare nel nostro keygen.

Dopo questa breve introduzione direi che possiamo cominciare.

Gli strumenti del mestiere:

- OllyDbg 1.10
- Cryptool 1.2 (chiamato anche Cryptocal 1.2)
- PEiD 0.94
- MASM32 (per il keygen)

Come prima cosa installiamo il programma e, una volta che tutto è a posto e funzionante, chiudiamolo tramite l'icona nel system tray. Carichiamo il modulo bdmcon.exe in olly e aspettiamo che l'analisi sia terminata. Adesso siamo pronti per far partire l'applicazione.
Qualche volta mi è capitato che prima di avviarsi passava una vita, quindi se dovesse succedere anche a voi non preoccupatevi e andate a prepararvi un

caffè!

Andiamo nel tab registrazione e clicchiamo su “Inserire nuova chiave”. Bene classica finestra di inserimento serial: 4 campi da 5 chars ciascuno e un pulsante “Registra” per il momento disattivato.
Inserite quello che vi pare (io ci ho messo 11111-22222-33333-44444) e vedrete che il pulsante si attiverà. Premiamolo e vediamo una bella finestrella di errore.
Bene, ci siamo fatti un'idea del funzionamento e siamo pronti per settare qualche BP in olly.
Ho cominciato con i classici GetDlgItemText, GetWindowText, MessageBox,... senza ottenere risultati, poi ho provato a guardare tra le string references se c'era qualcosa di interessante e malgrado ho trovato cose interessanti tipo CheckRegistration o InvalidKeyNag nessuna mi ha permesso di breakkare...

Probabilmente la gestione del controllo e gli errori non vengono fatti da bdmcon.exe, ma bensì da qualche altro modulo.
Guardando i moduli caricati ne troviamo alcuni interessanti tipo NAG.dll, agentreg.dll e popup.dll. Sono passato quindi a un'analisi delle varie export e ho trovato:

- CheckRegistration (esportata da agenreg.dll)
- ?OnButtonRegister@CNagDialog@@... e ?ShowPopup@CNagDialog@@... (esportate da NAG.dll
- ShowPopupMessage (esportata da Popup.dll)

Indovinate un po'? Nessuna di queste funziona...Porc **ç/**ç&%”*&%/&(&ç%
A questo ho chiuso tutto e sono andato a scaricarmi un seriale valido!

No scherzo ;-)
Ho analizzato altre librerie e ho trovato tx_messagebox_s e tx_messagebox esportate da TxTools.dll. Ho messo un BP sulle 2 funzioni e finalmente ho fatto centro!
Si breakka all'interno della TxTools.dll e uscendo dalla call della messagebox ci ritroviamo nella agentreg.dll. A questo punto ci sorge il sospetto che il controllo venga fatto proprio da quest'ultima libreria, quindi diamo uno sguardo alle string references. Ne troviamo un sacco interessanti:

```
IDS_INVALID_SERIAL
IDS_WRONG_SERIAL
IDS_EXISTING_SERIAL
IDS_SERIAL
IDS_BTN_REGISTER
TEST_AS: ValidateSerial()
TEST AS OnBnClickedOk()
```

Dopo qualche breve prova (ho messo BP su diverse stringhe...hehehe) sono arrivato dove mi interessava breakkando su TEST AS OnBnClickedOk, proprio prima della lettura dei campi.

```
00A85BDC  PUSH 00A96FC0                    ; ASCII "TEST AS OnBnClickedOk()"
00A85BE1  MOV EBP,ECX
00A85BE3  CALL 00A90B80
00A85BE8  ADD ESP,4
00A85BEB  PUSH 5
00A85BED  LEA ECX,DWORD PTR [EBP+B18]
00A85BF3  CALL DWORD PTR [<&MFC71.#2468>]   ; lettura prima parte serial
00A85BF9  LEA EBX,DWORD PTR [EBP+1180]
...
00A85C11  PUSH 5
00A85C13  LEA ECX,DWORD PTR [EBP+B1C]
00A85C19  CALL DWORD PTR [<&MFC71.#2468>]   ; lettura seconda parte serial
00A85C1F  MOV ECX,EAX
...
00A85C46  PUSH 5
00A85C48  LEA ECX,DWORD PTR [EBP+B20]
00A85C4E  MOV BYTE PTR [EBP+118A],0
00A85C55  CALL DWORD PTR [<&MFC71.#2468>]   ; lettura terza parte serial
00A85C5B  MOV ECX,EAX
...
00A85C86  PUSH 5
00A85C88  LEA ECX,DWORD PTR [EBP+B24]
00A85C8E  MOV BYTE PTR [EBP+118F],0
00A85C95  CALL DWORD PTR [<&MFC71.#2468>]   ; lettura quarta parte serial
00A85C9B  MOV ECX,EAX
```

E dopo la lettura dei campi si arriva a questa call:

```
00A85CE8  CALL 00A82FD0
```

Sarà la chiave di tutto e lo si può capire facilmente poiché da quel punto fino alla messagebox non c'è più niente di interessante.
Metteteci un bel BP e fate analizzare ad olly il modulo (CTRL+A) per fargli creare il file agentreg.udd altrimenti, avendo analizzato solo il file bdmcon.exe, la prossima volta che aprirete il programma il BP sarà andato a farsi fottere.

Vabbé entriamo in questa call e cominciamo a steppare:

```
00A82FD0  PUSH -1
00A82FD2  PUSH 00A94887             ; SE handler installation
...
00A82FFE  MOV ESI,ECX
00A83000  CALL 00A90B80             ; poco interessante
00A83005  ADD ESP,4
00A83008  LEA ECX,DWORD PTR [ESP+84]
00A8300F  CALL 00A82920             ; poco interessante
00A83014  MOV EAX,DWORD PTR [A9DB00]
00A83019  ADD EAX,4
00A8301C  PUSH EAX                                  ; /s2
00A8301D  LEA EDI,DWORD PTR [ESI+1180]              ; |
00A83023  PUSH EDI                                  ; |s1
00A83024  MOV DWORD PTR [ESP+D4],0                  ; |
00A8302F  CALL DWORD PTR [<&MSVCR71._stricmp>]      ; \verifica serial inserito
con quello attuale
00A83035  ADD ESP,8
00A83038  TEST EAX,EAX
00A8303A  JNZ SHORT 00A8305D
00A8303C  LEA ECX,DWORD PTR [ESP+84]
00A83043  MOV DWORD PTR [ESP+CC],-1
00A8304E  CALL 00A82950
```

```
00A83053  MOV EAX,5
00A83058  JMP 00A83787
00A8305D  LEA ECX,DWORD PTR [ESP+48]
00A83061  CALL 00A82920                          ; poco interessante
00A83066  PUSH EDI                               ; /Arg1
00A83067  LEA ECX,DWORD PTR [ESP+4C]             ; |
00A8306B  MOV BYTE PTR [ESP+D0],1                ; |
00A83073  CALL 00A82980                          ; \call controllo ultimi 4
chars + blowfish
00A83078  CMP EAX,1
00A8307B  JE SHORT 00A830AF
```

Come potete vedere dai commenti si incontra subito una call che si occupa di verificare se il seriale appena inserito corrisponde a quello attualmente in uso. Se non è così si passa oltre e si va sempre più a fondo verso i controlli seri. La prossima call che si incontra è quella a 00A83061 che si occuperà solo di inizializzare qualcosa, mentre la call seguente ci proietterà sempre più a fondo. Entriamoci e continuiamo a steppare allegramente. Apparentemente i soliti controlli di circostanza senonché prima di uscire troviamo ancora un paio di calls:

```
00A82A73  MOV ECX,DWORD PTR [ESP+14]
00A82A77  ADD ESP,4
00A82A7A  CALL 00A82760              ; calcoli e controllo ultimi 4 chars
00A82A7F  MOV ECX,DWORD PTR [ESP+20]
00A82A83  POP EBP
00A82A84  POP EDI
00A82A85  POP ESI
00A82A86  POP EBX
00A82A87  CALL 00A88DAE              ; poco interessante
00A82A8C  ADD ESP,14
00A82A8F  RETN 4
```

Visto che abbiamo ipotizzato che i calcoli e i controlli si trovano forzatamente all'interno della call a 00A85CE8 dobbiamo partire dal presupposto di trovare qualcosa. Speriamo di non sbagliarci!

Come vedete dai listati ho preso l'abitudine di aggiungere commenti direttamente in olly, mi servono per evitare di tracciare calls classificate come inutili diverse volte e di risparmiare parecchio tempo. D'altro canto bisogna fare attenzione a scartare a priori alcuni passaggi perché potrebbero rivelarsi fondamentali. Comunque non preoccupatevi perché in questo caso non é così visto che ho verificato tutto per benino.

Entriamo anche nella call a 00A82A7A e all'interno troviamo subito un piccolo ciclo:

```
00A827C1  /LEA EDX,DWORD PTR [ESP+10]
00A827C5  |PUSH EDX
00A827C6  |PUSH 00A96B44                           ; |format = "%02X"
00A827CB  |PUSH EBP                                ; |s
00A827CC  |MOV DWORD PTR [ESP+1C],EBX              ; |
00A827D0  |CALL DWORD PTR [<&MSVCR71.sscanf>]      ; \sscanf
00A827D6  |MOV AL,BYTE PTR [ESP+1C]
00A827DA  |ADD ESP,0C
00A827DD  |MOV BYTE PTR [ESP+EDI+2C],AL
00A827E1  |INC EDI
```

```
00A827E2  |ADD EBP,2
00A827E5  |CMP EDI,8                                ; numero di esecuzioni
00A827E8  \JL SHORT 00A827C1
```

Questa piccola sequenza prende a 2 a 2 i chars del seriale inserito e li “trasforma” in bytes. Il ciclo viene ripetuto 8 volte, quindi per 16 chars. Gli ultimi 4 vengono snobbati! Uhmmm

Proseguiamo e all'uscita del ciclo troviamo questa parte di codice:

```
00A827EA  PUSH 8
00A827EC  LEA ECX,DWORD PTR [ESP+30]
00A827F0  PUSH ECX
00A827F1  PUSH EBX
00A827F2  CALL 00A82680                         ; calcolo ultimi 4 chars corretti
00A827F7  LEA EDX,DWORD PTR [ESP+1C]
00A827FB  PUSH EDX
00A827FC  PUSH 00A96B3C                         ; |format = "%04X"
00A82801  OR EBP,FFFFFFFF                       ; |
00A82804  MOV DWORD PTR [ESP+24],EBP            ; |
00A82808  MOV EDI,EAX                           ; |
00A8280A  LEA EAX,DWORD PTR [ESI+14]            ; |
00A8280D  PUSH EAX                              ; |s
00A8280E  CALL DWORD PTR [<&MSVCR71.sscanf>]    ; \confronto ultimi 4 chars
00A82814  ADD ESP,18
00A82817  CMP WORD PTR [ESP+10],DI
00A8281C  LEA ECX,DWORD PTR [ESP+18]
00A82820  JE SHORT 00A82832                     ; controllo ultime 4 cifre
00A82822  MOV DWORD PTR [ESP+40],EBP
00A82826  CALL 00A816C0
00A8282B  XOR EAX,EAX
00A8282D  JMP 00A828DF                          ; Ritenta, Nulla di fatto!
```

I miei commenti per questa parte sono completamente esplicativi.
Comunque se osservate a runtime i parametri della prima call vedrete che ECX punta alla sequenza di bytes creata nel ciclo poco sopra.
Questa sequenza verrà utilizzata per calcolare il valore corretto degli ultimi 4 chars del seriale che, se vi ricordate, non erano presenti nella formattazione. Il risultato lo troveremo in EAX appena usciti dalla call.

L'analisi dei calcoli eseguiti l'ho tralasciata completamente, poiché non è strettamente necessario comprendere i passaggi per arrivare alla soluzione finale.
In generale vengono usati AND, XOR e SHIFT e la maschera fissa seguente:

```
Maschera  dd 00000000h, 0000CC01h, 0000D801h, 00001400h,\
             0000F001h, 00003C00h, 00002800h, 0000E401h,\
             0000A001h, 00006C00h, 00007800h, 0000B401h,\
             00005000h, 00009C01h, 00008801h, 00004400h
```

Noi copieremo il codice della call e lo riporteremo nel nostro keygen aggiustando qualche cosuccia tipo i puntatori alla maschera.

Andando avanti troviamo ancora una call a sscanf, ma stavolta viene utilizzata non tanto per la formattazione, ma bensì per il confronto tra gli ultimi 4 chars del nostro fake serial e quelli corretti appena calcolati.

Adesso per poter andare avanti o forziamo il salto a 00A82820 oppure modifichiamo gli ultimi 4 chars con quelli che si aspetta BitDefender.
Ho optato per la seconda possibilità e per il fake serial inserito da me (11111-22222-33333-4xxxx) ottengo E4D5.
Quindi con 11111-22222-33333-4E4D5 passerò tranquillamente questo primo controllo.
Viene preso il salto a 00A82820 e atterriamo qua:

```
00A82832  PUSH 38
00A82834  PUSH 00A9C100
00A82839  CALL 00A81AA0
00A8283E  PUSH 8
00A82840  LEA ECX,DWORD PTR [ESP+28]
00A82844  PUSH ECX
00A82845  LEA EDX,DWORD PTR [ESP+34]
00A82849  PUSH EDX
00A8284A  LEA ECX,DWORD PTR [ESP+24]
00A8284E  CALL 00A82260
```

Ancora 2 calls (Ufffffff) ma oramai siamo al piatto forte!
Provate a entrare nella prima e non spaventatevi, vi presento Blowfish!
Essendo stata la prima volta che ho avuto a che fare direttamente con Blowfish non l'ho identificato subito, ma ho dovuto ricorrere a PEiD.
Se ci caricate agentreg.dll e usate il plugin Krypto Analyzer salteranno fuori le referenze a Blowfish e alle cifre esadecimali di PI greco.
Se andate all'offset indicato per PI greco (RVA=1B078 in olly A9B078) ci troverete le P-Array iniziali (18 dwords):

```
P1  = 0x243f6a88, P2  = 0x85a308d3, P3  = 0x13198a2e, P4  = 0x03707344
P5  = 0xa4093822, P6  = 0x299f31d0, P7  = 0x082efa98, P8  = 0xec4e6c89
P9  = 0x452821e6, P10 = 0x38d01377, P11 = 0xbe5466cf, P12 = 0x34e90c6c
P13 = 0xc0ac29b7, P14 = 0xc97c50dd, P15 = 0x3f84d5b5, P16 = 0xb5470917
P17 = 0x9216d5d9, P18 = 0x8979fb1b
```

Subito sotto ci troverete anche le 4 S-boxes (256 dword per ogni S-box), per un totale di:
72 bytes (P-Array) + 4096 Bytes (S-Boxes) = 4168 Bytes.

Visto che ci siamo entriamo in questa prima call (mancano i primi address) e passiamo all'analisi:

```
00A81AE1  JMP SHORT 00A81AF0                    ;  sposta le P-Array iniziali
00A81AE3  MOV ESI,DWORD PTR [ESP+10]
00A81AE7  JMP SHORT 00A81AF0
00A81AE9  LEA ESP,DWORD PTR [ESP]
00A81AF0  MOV EBX,DWORD PTR [ECX+A9B078]
00A81AF6  MOV EDI,DWORD PTR [EBP+8]
00A81AF9  MOV DWORD PTR [ECX+EDI],EBX           ;  follow in dump (P1, P7, P13)
00A81AFC  MOV EBX,DWORD PTR [ECX+A9B07C]
00A81B02  MOV EDI,DWORD PTR [EBP+8]
00A81B05  LEA EAX,DWORD PTR [ECX+A9B07C]
00A81B0B  MOV DWORD PTR [ECX+EDI+4],EBX         ;  (P2, P8, P14)
00A81B0F  MOV EDI,DWORD PTR [EBP+8]
00A81B12  MOV EBX,DWORD PTR [EAX+4]
00A81B15  ADD EDI,EAX
```

```
00A81B17  MOV DWORD PTR [EDI+EDX],EBX          ;  (P3, P9, P15)
00A81B1A  MOV EDI,DWORD PTR [EBP+8]
00A81B1D  MOV EBX,DWORD PTR [EAX+8]
00A81B20  ADD EDI,EAX
00A81B22  MOV DWORD PTR [EDI+ESI],EBX          ;  (P4, P10, P16)
00A81B25  MOV ESI,DWORD PTR [EBP+8]
00A81B28  MOV EDI,DWORD PTR [EAX+C]
00A81B2B  MOV EBX,DWORD PTR [ESP+14]
00A81B2F  ADD ESI,EAX
00A81B31  MOV DWORD PTR [ESI+EBX],EDI          ;  (P5, P11, P17)
00A81B34  MOV ESI,DWORD PTR [EBP+8]
00A81B37  MOV EDI,DWORD PTR [ESP+18]
00A81B3B  ADD ESI,EAX
00A81B3D  MOV EAX,DWORD PTR [EAX+10]
00A81B40  ADD ECX,18                           ;  18*3 cicli=48
00A81B43  CMP ECX,48
00A81B46  MOV DWORD PTR [ESI+EDI],EAX          ;  P6, P12, P18)
00A81B49  JL SHORT 00A81AE3
```

In questa prima parte vengono semplicemente spostate le P-Array e se fate “follow in dump” --> “memory address”, esattamente nel punto che ho marcato nei commenti, vi troverete all'address di destinazione dello spostamento.
Nei commenti trovate il punto esatto di ogni spostamento (P1, P2, P3,...).
In ogni ciclo vengono spostate 6 dwords e in totale le P-Array sono composte da 18 dwords. ECX fa da counter e a ogni ciclo viene incrementato di 18, visto che la condizione di uscita è che ECX sia almeno 48, saranno necessari 3 cicli completi per eseguire tutto lo spostamento.

Andiamo oltre e continuiamo la nostra analisi:

```
00A81B5B  JMP SHORT 00A81B60                   ;  sposta le S-Boxes
00A81B5D  LEA ECX,DWORD PTR [ECX]
00A81B60  LEA ECX,DWORD PTR [EAX+C]
00A81B63  MOV EDX,40
00A81B68  JMP SHORT 00A81B70
00A81B6A  LEA EBX,DWORD PTR [EBX]
00A81B70  MOV EDI,DWORD PTR [EAX+A9B0C0]
00A81B76  MOV ESI,DWORD PTR [EBP+4]
00A81B79  MOV DWORD PTR [EAX+ESI],EDI          ;  follow in dump (sposta)
00A81B7C  MOV EDI,DWORD PTR [EAX+A9B0C4]
00A81B82  MOV ESI,DWORD PTR [EBP+4]
00A81B85  MOV DWORD PTR [EAX+ESI+4],EDI        ;  (sposta)
00A81B89  MOV ESI,DWORD PTR [ESP+18]
00A81B8D  MOV EDI,DWORD PTR [EBP+4]
00A81B90  MOV EBX,DWORD PTR [EAX+A9B0C8]
00A81B96  LEA ESI,DWORD PTR [ESI+EAX+A9B0C4]
00A81B9D  MOV DWORD PTR [ESI+EDI],EBX          ;  (sposta)
00A81BA0  MOV EDI,DWORD PTR [EAX+A9B0CC]
00A81BA6  MOV ESI,DWORD PTR [EBP+4]
00A81BA9  MOV DWORD PTR [ECX+ESI],EDI          ;  (sposta)
00A81BAC  ADD EAX,10
00A81BAF  ADD ECX,10
00A81BB2  DEC EDX
00A81BB3  JNZ SHORT 00A81B70
00A81BB5  CMP EAX,1000
00A81BBA  JL SHORT 00A81B60
```

Qua vengono invece spostate le S-Boxes. Stesso discorso di prima per sapere dove verranno copiate.

Come vedete la condizione di uscita è che eax sia almeno 1000h e visto che ogni volta viene incrementato di 10h, saranno necessari 100h cicli.
Se trasformiamo in base 10 abbiamo 100h = 256d e se guardate bene in ogni ciclo vengono spostate 4 dwords, per un totale di 1024 dwords = 4096 Bytes. I conti tornano!

Proseguendo nel codice (che non riporto completamente) c'è una prima inizializzazion sulle P-Array e qua cominciano i guai! Questa elaborazione dipende dalla private key che noi non conosciamo...
Chiaramente possiamo pescarla direttamente dal programma ed usarla nel nostro keygen, ma in fin dei conti vedremo che non è necessario, in quanto sarà più comodo usare direttamente il risultato finale delle P-array dopo l'inizializzazione. Questo ci permetterà, oltre a risparmiare codice da scrivere, anche di velocizzare il keygenning ;-)

```
00A81BE1  MOVZX EAX,BYTE PTR [EDX+EDI]  ; in eax secondo byte della private key
00A81BE5  MOV AH,BYTE PTR [EBX+EDI]     ; in ah primo byte della private key
00A81BE8  SHL EAX,8
00A81BEB  MOV DWORD PTR [ESP+24],EAX
00A81BEF  LEA EAX,DWORD PTR [EBX+2]
00A81BF2  CDQ
00A81BF3  IDIV ESI
00A81BF5  MOV EAX,DWORD PTR [ESP+24]
00A81BF9  MOVZX EDX,BYTE PTR [EDX+EDI]  ; in edx terzo byte della private key
00A81BFD  OR EAX,EDX
00A81BFF  SHL EAX,8
00A81C02  MOV DWORD PTR [ESP+24],EAX
00A81C06  MOV EDX,DWORD PTR [ESP+10]
00A81C0A  LEA EAX,DWORD PTR [EDX+3]
00A81C0D  CDQ
00A81C0E  IDIV ESI
00A81C10  MOVZX EAX,BYTE PTR [EDX+EDI]  ; in eax quarto byte della private key
00A81C14  MOV EDX,DWORD PTR [ESP+24]
00A81C18  OR EDX,EAX                    ; in edx prima dword della private key
00A81C1A  MOV EAX,DWORD PTR [ESP+20]
00A81C1E  XOR DWORD PTR [EAX],EDX       ; xora P1/P7/P13
```

Come vedete dai commenti, verranno presi uno dopo l'altro i bytes della private key e swappati prima di andare a xorare le varie P-Array. Se sulla seconda istruzione riportata qua sopra seguite il puntatore, vi ritroverete proprio alla private key, composta in questo caso da 14 dwords (448 bits che è il massimo previsto da Blowfish).

```
Key1  = 0x680594ac, key2  = 0x4b28f988, key3  = 0xd22ded41, key4  = 0xfe701824
key5  = 0xf7e89702, key6  = 0xa14ff134, key7  = 0x6f834ddb, key8  = 0x7c2de26c
key9  = 0xd8a0ae6d, key10 = 0x1f046b79, key11 = 0x51b3bf8a, key12 = 0xdde3b378
key13 = 0xff8666bc, key14 = 0x4c6e037a
```

Le dwords da xorare sono però 18 (le P-Array), quindi una volta “utilizzata” tutta la private key si ricomincerà dall'inizio.

Il risultato delle P-Array dopo tutti gli xor sarà il seguente:

```
P1  = 0x88ab6fe0, P2  = 0x0D5a2098, P3  = 0x52f4a7fc, P4  = 0x276803ba
P5  = 0xA69ed0d5, P6  = 0x1D6e7e71, P7  = 0xD36379f7, P8  = 0x80ac41f5
P9  = 0x2886813e, P10 = 0x41bb1768, P11 = 0x34ebd59e, P12 = 0x4C5aefb1
```

```
P13 = 0x7Ccaaf48, P14 = 0xB37f3e91, P15 = 0x9310d0dd, P16 = 0x3Dbe215c
P17 = 0xD3fbf80b, P18 = 0xAD618be5
```

Se prendiamo l'esempio di P15 possiamo effettivamente verificare che una volta terminata la private key, si riprende dall'inizio:

```
P15 iniziale    = 0x3f84d5b5    xor
Key1 (swappata) = 0xac940568
                  ----------
                  0x9310d0dd
```

Dopo questa prima elaborazione delle P-Array si passa all'inizializzazione per costruire tutte le subkeys necessarie in seguito.
Proseguiamo nel codice e troveremo altre 2 calls che si occuperanno dell'inizializzazione:

```
00A81E1B  XOR ESI,ESI
00A81E1D  LEA ECX,DWORD PTR [ECX]
00A81E20  LEA ECX,DWORD PTR [ESP+20]
00A81E24  PUSH ECX
00A81E25  LEA EDX,DWORD PTR [ESP+28]
00A81E29  PUSH EDX
00A81E2A  MOV ECX,EBP
00A81E2C  CALL 00A816E0                     ;  inizializzazione P1-P18
00A81E31  MOV EAX,DWORD PTR [EBP+8]
00A81E34  MOV ECX,DWORD PTR [ESP+24]
00A81E38  MOV DWORD PTR [ESI+EAX],ECX       ;  sovrascrive una P-Array
00A81E3B  MOV EDX,DWORD PTR [EBP+8]
00A81E3E  MOV EAX,DWORD PTR [ESP+20]
00A81E42  MOV DWORD PTR [ESI+EDX+4],EAX     ;  sovrascrive una P-Array
00A81E46  ADD ESI,8                         ;  ESI = Counter
00A81E49  CMP ESI,48                        ;  48/8=9 cicli
00A81E4C  JL SHORT 00A81E20
```

Il ciclo è ripetuto 9 volte poiché ogni volta che viene eseguita la call verranno create 2 nuove dwords che andranno a sostituire quelle precedenti. Il totale sarà naturalmente 18 (come le P-Array).

```
00A81E4E  XOR ESI,ESI
00A81E50  MOV EDI,80                        ;  80h=128d
00A81E55  LEA ECX,DWORD PTR [ESP+20]
00A81E59  PUSH ECX
00A81E5A  LEA EDX,DWORD PTR [ESP+28]
00A81E5E  PUSH EDX
00A81E5F  MOV ECX,EBP
00A81E61  CALL 00A816E0                     ;  inizializzazione S1-S4
00A81E66  MOV EAX,DWORD PTR [EBP+4]
00A81E69  MOV ECX,DWORD PTR [ESP+24]
00A81E6D  MOV DWORD PTR [ESI+EAX],ECX       ;  follow in dump
00A81E70  MOV EDX,DWORD PTR [EBP+4]
00A81E73  MOV EAX,DWORD PTR [ESP+20]
00A81E77  MOV DWORD PTR [ESI+EDX+4],EAX
00A81E7B  ADD ESI,8                         ;  80h*8h=400h (1024d bytes)
00A81E7E  DEC EDI
00A81E7F  JNZ SHORT 00A81E55
00A81E81  CMP ESI,1000                      ;  1000/400=4 cicli (S1-S4)
00A81E87  JL SHORT 00A81E50
```

E dopo l'inizializzazione delle P-Array si passa alle S-Boxes usando sempre lo

stesso metodo, infatti le calls puntano entrambe allo stesso indirizzo, stavolta però il ciclo sarà ripetuto qualche volta in più.
In verità sono 2 cicli, uno annidato nell'altro, poiché le 4 S-Boxes vengono trattate separatamente. Avremo quindi il ciclo più interno eseguito 80h = 128 volte (in EDI) per poi passare al controllo del counter del ciclo più esterno (in ESI). ESI viene incrementato di 8 ad ogni esecuzione del ciclo interno, quindi dopo l'uscita dal ciclo interno varrà 128*8=1024. Questo significa che il ciclo più esterno verrà eseguito 4 volte. In totale verranno elaborati tutti i bytes dell S-Boxes (4096).

Se volete sapere esattamente come vengono create le subkeys vi consiglio di scaricare il sorgente di Blowfish di Bruce Schneier.

Bene l'inizializzazione è finita, ma se vi ricordate c'è ancora una call in ballo, infatti dopo aver bypassato il primo controllo (quello degli ultimi 4 chars) abbiamo incontrato 2 calls.
Vi riporto la piccola parte di codice:

```
00A82832  PUSH 38
00A82834  PUSH 00A9C100
00A82839  CALL 00A81AA0                ; inizializzazione
00A8283E  PUSH 8
00A82840  LEA ECX,DWORD PTR [ESP+28]
00A82844  PUSH ECX
00A82845  LEA EDX,DWORD PTR [ESP+34]
00A82849  PUSH EDX
00A8284A  LEA ECX,DWORD PTR [ESP+24]
00A8284E  CALL 00A82260                <-- manca questa!
```

Entriamoci e troveremo questa parte di codice:

```
00A82288  /MOV EAX,DWORD PTR [ESP+14]
00A8228C  |TEST EAX,EAX
00A8228E  |JE SHORT 00A822A1
00A82290  |LEA ECX,DWORD PTR [ESI+4]
00A82293  |PUSH ECX
00A82294  |PUSH ESI
00A82295  |MOV ECX,EBX
00A82297  |CALL 00A81EA0                ; non viene eseguita
00A8229C  |ADD ESI,8
00A8229F  |JMP SHORT 00A822E0
00A822A1  |MOV DL,BYTE PTR [ESI]        ;  char 1 e 2 del seriale in DL
00A822A3  |MOV BYTE PTR [EDI],DL        ;  salva DL in [EDI]
00A822A5  |MOV AL,BYTE PTR [ESI+1]      ;  char 3 e 4 del seriale in AL
00A822A8  |MOV BYTE PTR [EDI+1],AL      ;  salva AL in [EDI+1]
00A822AB  |MOV CL,BYTE PTR [ESI+2]      ;  char 5 e 6 del seriale in CL
00A822AE  |MOV BYTE PTR [EDI+2],CL      ;  salva CL in [EDI+2]
00A822B1  |MOV DL,BYTE PTR [ESI+3]      ;  char 7 e 8 del seriale in DL
00A822B4  |MOV BYTE PTR [EDI+3],DL      ;  salva DL in [EDI+3]
00A822B7  |MOV CL,BYTE PTR [ESI+4]      ;  char 9 e 10 del seriale in CL
00A822BA  |LEA EAX,DWORD PTR [EDI+4]
00A822BD  |MOV BYTE PTR [EAX],CL        ;  salva CL in [EDI+4] ([EAX])
00A822BF  |MOV DL,BYTE PTR [ESI+5]      ;  char 11 e 12 del seriale in DL
00A822C2  |MOV BYTE PTR [EDI+5],DL      ;  salva DL in [EDI+5]
00A822C5  |MOV CL,BYTE PTR [ESI+6]      ;  char 13 e 14 del seriale in DL
00A822C8  |MOV BYTE PTR [EDI+6],CL      ;  salva CL in [EDI+6]
00A822CB  |MOV DL,BYTE PTR [ESI+7]      ;  char 15 e 16 del seriale in DL
00A822CE  |PUSH EAX
```

```
00A822CF  |PUSH EDI
00A822D0  |MOV ECX,EBX
00A822D2  |MOV BYTE PTR [EDI+7],DL       ;  salva DL in [EDI+7]
00A822D5  |CALL 00A81EA0                 ;  cifratura
00A822DA  |ADD ESI,8
00A822DD  |ADD EDI,8
00A822E0  |DEC EBP
00A822E1  \JNZ SHORT 00A82288
```

Per prima cosa vengono salvati i chars che compongono il nostro seriale all'indirizzo puntato da [EDI] poi si entra nella call a 00A822D5 che eseguirà diverse operazioni sul nostro seriale con l'aiuto delle subkeys create poco fa! Entriamo anche in questa call, tanto ormai...

```
00A81EA0  PUSH ECX
00A81EA1  PUSH EBX
00A81EA2  MOV EAX,DWORD PTR [ESP+C]
00A81EA6  PUSH EBP
00A81EA7  PUSH ESI
00A81EA8  MOV ESI,DWORD PTR [EAX]           ; primi 8 chars in [EAX]
00A81EAA  PUSH EDI
00A81EAB  MOV EDI,DWORD PTR [ECX+8]
00A81EAE  MOV EDX,DWORD PTR [EDI+44]        ; [EDI+44] contiene P18
00A81EB1  XOR EDX,ESI
00A81EB3  MOV ESI,DWORD PTR [ECX+4]
00A81EB6  MOV EAX,EDX
00A81EB8  SHR EAX,10
00A81EBB  MOVZX EAX,AL
00A81EBE  MOV EAX,DWORD PTR [ESI+EAX*4+400] ; [ESI+EAX*4+400] = zona X-Boxes
00A81EC5  MOV EBX,EDX
00A81EC7  SHR EBX,18
00A81ECA  ADD EAX,DWORD PTR [ESI+EBX*4]     ; [ESI+EBX*4] = zona X-Boxes
00A81ECD  MOVZX EBX,DH
00A81ED0  XOR EAX,DWORD PTR [ESI+EBX*4+800] ; [ESI+EAX*4+800] = zona X-Boxes
00A81ED7  MOV EBX,EDX
00A81ED9  AND EBX,0FF
00A81EDF  MOV EBP,DWORD PTR [ESI+EBX*4+C00] ; [ESI+EAX*4+C00] = zona X-Boxes
00A81EE6  MOV EBX,DWORD PTR [EDI+40]        ; [EDI+44] contiene P17
00A81EE9  ADD EAX,EBP
00A81EEB  XOR EAX,EBX
00A81EED  MOV EBX,DWORD PTR [ESP+1C]
00A81EF1  XOR EAX,DWORD PTR [EBX]           ;  secondi 8 chars in [EBX]
00A81EF3  MOV EBX,EAX
00A81EF5  SHR EBX,10
```

Qua entrano in gioco le subkeys (P-Array e S-Boxes) e il nostro seriale, però c'è un piccolo problema, poiché la prima P-Array utilizzata è la P18 mentre normalmente dovrebbe essere la P1!!!
Penso che quelli della Softwin abbiano usato volontariamente la procedura di decifratura al posto della cifratura, poiché tra le 2 procedure cambia soltanto l'ordine con cui vengono prese le P-Array.
Per poter capire cosa stava succedendo realmente ho dovuto appoggiarmi a un programma che usasse correttamente l'algoritmo e ho optato per Cryptool 1.2.

Per chiarezza sono costretto a riportare anche codice di questo programma, cosicché possiate capire tutti i passaggi che ho seguito per poter scrivere il keygen.

Dopo aver caricato in olly cryptocal.exe dobbiamo arrivare al punto giusto per poter fare i confronti a runtime dell'algoritmo.
Vi risparmio la procedura e vi riporto direttamente il punto esatto con alcune parti tagliate:

```
0045180C  PUSH 56                                  ; /Count = 56 (86.)
0045180E  PUSH 0045D96E                            ; |Buffer = cryptoca.0045D96E
00451813  PUSH 13B0                                ; |ControlID = 13B0 (5040.)
00451818  PUSH DWORD PTR [EBP+8]                   ; |hWnd
0045181B  CALL <JMP.&USER32.GetDlgItemTextA>       ; \lettura private key
00451820  OR EAX,EAX
00451822  JNZ SHORT 0045182D
...
0045182D  PUSH 0045D96E
00451832  CALL 00452C1F
00451837  PUSH EAX
00451838  PUSH 0045D96E
0045183D  CALL 00401083                            ;  inizializzazione blowfish
00451842  PUSH 4000                                ; /Length = 4000 (16384.)
...
00451870  PUSH 800                                 ; /Count = 800 (2048.)
00451875  PUSH 0046196E                            ; |Buffer = cryptoca.0046196E
0045187A  PUSH 13AF                                ; |ControlID = 13AF (5039.)
0045187F  PUSH DWORD PTR [EBP+8]                   ; |hWnd
00451882  CALL <JMP.&USER32.GetDlgItemTextA>       ; \lettura testo da criptare
...
004518CA  /PUSH EBX
004518CB  |MOV EAX,DWORD PTR [EBP-8]
004518CE  |ADD EAX,EBX
004518D0  |PUSH EAX
004518D1  |PUSH 0046116E
004518D6  |CALL 00401121                           ;  cifratura
004518DB  |LEA EDI,DWORD PTR [46116E]
004518E1  |PUSH 2
```

A 0045183D troverete la call che inizializzerà le P-Array e le S-Boxes se ci entrate troverete la stessa struttura presente in BitDefender:

```
004010C9  /MOV EAX,DWORD PTR [ECX+4678D4]     ; prime operazioni sulle P-Array
004010CF  |BSWAP EAX                           ; Swap delle dwords che
004010D1  |XOR DWORD PTR [ECX+46688C],EAX      ; compongono la private key
004010D7  |SUB ECX,4                           ; e xor con P1, P2, P3,...
004010DA  \JNZ SHORT 004010C9
004010DC  XOR EAX,EAX
004010DE  MOV EDI,00466890
004010E3  MOV DWORD PTR [4678D8],EAX
004010E8  MOV DWORD PTR [4678DC],EAX
004010ED  MOV ECX,9
004010F2  PUSH 004678D8
004010F7  JMP SHORT 004010FA
004010F9  /PUSH EAX
004010FA  |PUSH EDI
004010FB  |CALL 00401121                       ;  inizializzazione P1-P18
00401100  |MOV EAX,EDI
00401102  |ADD EDI,8
00401105  |DEC ECX
00401106  \JNZ SHORT 004010F9
00401108  MOV ECX,200                          ;  copiare le P-Array
0040110D  /PUSH EAX
0040110E  |PUSH EDI
0040110F  |CALL 00401121                       ;  inizializzazione S1-S4
```

```
00401114  |MOV EAX,EDI
00401116  |ADD EDI,8
00401119  |DEC ECX
0040111A  \JNZ SHORT 0040110D
```

Anche qua troviamo 2 calls che puntano allo stesso indirizzo e sono quelle che rispettivamente inizializzano le P-Array e le S-Boxes.
Tra le 2 calls c'è anche un mio commento “copiare le P-Array”. Esattamente in quel punto andavo a iniettare le P-Array calcolate da BitDefender per poter far costruire correttamente tutte le S-Boxes.
Avrei anche potuto copiare la private key prima dell'inizializzazione e avrei ottenuto lo stesso risultato, ma visto che nel keygen useremo le P-Array già inizializzate (risparmiamo sul tempo di esecuzione) ho fatto lo stesso anche nelle mie prove.

Dopo l'inizializzazione viene letto il testo da criptare e si entra nella call a 004518D6.
Troviamo il codice seguente:

```
00401121  PUSH EBP
00401122  MOV EBP,ESP
00401124  PUSHAD
00401125  MOV EDI,DWORD PTR [EBP+C]
00401128  MOV EAX,DWORD PTR [EDI]
0040112A  MOV EDX,DWORD PTR [EDI+4]
0040112D  XOR EBX,EBX
0040112F  XOR ECX,ECX
00401131  XOR EDI,EDI
00401133  /XOR EAX,DWORD PTR [EDI*4+466890]   ; 466890 = address inizio P-Array
0040113A  |ROL EAX,10
0040113D  |MOV CL,AL
0040113F  |MOV BL,AH
00401141  |MOV ESI,DWORD PTR [ECX*4+466CD8]   ; [ECX*4+466CD8] = zona X-Boxes
00401148  |ROL EAX,10
0040114B  |ADD ESI,DWORD PTR [EBX*4+4668D8]   ; [ECX*4+4668D8] = zona X-Boxes
00401152  |MOV CL,AH
00401154  |MOV BL,AL
00401156  |XOR ESI,DWORD PTR [ECX*4+4670D8]   ; [ECX*4+4670D8] = zona X-Boxes
0040115D  |ADD ESI,DWORD PTR [EBX*4+4674D8]   ; [ECX*4+4674D8] = zona X-Boxes
00401164  |XOR EDX,ESI
00401166  |INC EDI
00401167  |XCHG EAX,EDX
00401168  |CMP EDI,10
0040116B  \JNZ SHORT 00401133
0040116D  MOV ESI,DWORD PTR [EBP+8]
00401170  XOR EAX,DWORD PTR [4668D0]
00401176  XOR EDX,DWORD PTR [4668D4]
0040117C  MOV DWORD PTR [ESI+4],EAX
0040117F  MOV DWORD PTR [ESI],EDX
00401181  POPAD
00401182  LEAVE
00401183  RETN 8
```

Come potete notare abbiamo diverse analogie con il ciclo di cifratura trovato in BitDefender, solo che in Cryptocal è implementato molto più elegantemente.
Questa stessa routine la ritroverete esattamente uguale nel keygen (puntatori a parte).

Facendo un confronto a runtime dei 2 programmi ho notato subito la differenza nell'ordine delle P-Array, solo che prima di poter fare un confronto coerente ho dovuto sudare parecchio...

Ok siamo a buon punto:

- sappiamo che vengono controllati gli ultimi 4 chars in funzione dei primi 16
- sappiamo che per calcolare gli ultimi 4 chars viene usata una maschera
- conosciamo la private key utilizzata
- conosciamo il valore delle P-Array dopo l'inizializzazione
- sappiamo che nella cifratura del nostro seriale le P-Array vengono usate in ordine inverso

Adesso ci rimangono solo i controlli che dipendono dal risultato della cifratura. Il risultato ottenuto dalla cifratura del mio seriale (11111-22222-33333-4xxxx) è:

60 75 96 3B BE 99 1A 40

Lo potete trovare seguendo il puntatore [EAX] alla fine della routine di cifratura:

```
00A82248  MOV DWORD PTR [EAX],ECX       ;  muove il testo criptato
00A8224A  MOV ECX,DWORD PTR [ESP+14]
00A8224E  POP EBP
00A8224F  MOV DWORD PTR [ECX],EDX       ;  muove il testo criptato
00A82251  POP EBX
00A82252  POP ECX
00A82253  RETN 8
```

e subito il ret, uscendo dalla cifratura, troviamo questo:

```
00A82853  MOV AL,BYTE PTR [ESP+29]      ; mette in AL il sesto byte = 99
00A82857  MOV CL,BYTE PTR [ESP+2A]      ; mette in CL il settimo byte = 1A
00A8285B  MOV DL,BYTE PTR [ESP+2B]      ; mette in DL l'ottavo byte = 40
00A8285F  MOV DWORD PTR [ESI+20],EBX    ; azzera [ESI+20]
00A82862  MOV BYTE PTR [ESI+22],AL      ; salva il sesto byte
00A82865  MOV AL,BYTE PTR [ESP+24]      ; mette in AL il primo byte = 60
00A82869  MOV BYTE PTR [ESI+21],CL      ; salva il settimo byte
00A8286C  MOV CL,BYTE PTR [ESP+25]      ; mette in CL il secondo byte = 75
00A82870  MOV DWORD PTR [ESI+28],EBX    ; azzera [ESI+28]
00A82873  MOV BYTE PTR [ESI+29],AL      ; salva il primo byte
00A82876  MOV AL,BYTE PTR [ESP+26]      ; mette in AL il terzo byte = 96
00A8287A  MOV BYTE PTR [ESI+28],CL      ; salva il secondo byte
00A8287D  MOV CL,BYTE PTR [ESP+27]      ; mette in CL il quarto byte = 3B
00A82881  MOV DWORD PTR [ESP+14],EBX    ; azzera
00A82885  MOV BYTE PTR [ESP+15],AL      ; salva il terzo byte
00A82889  MOV BYTE PTR [ESP+14],CL      ; salva il quarto byte
00A8288D  MOV ECX,DWORD PTR [ESP+14]    ; mette in ECX il terzo e il quarto byte
00A82891  MOV EAX,ECX                   ; li sposta in EAX
00A82893  SHR EAX,0C                    ; li shifta a destra di 0C (ris = 9)
00A82896  MOV BYTE PTR [ESI+20],DL      ; salva l'ottavo byte
00A82899  MOV DL,BYTE PTR [ESP+28]      ; mette in DL il quinto byte = BE
00A8289D  AND EAX,0F                    ; AND di EAX (=9) con 0F
00A828A0  MOV DWORD PTR [ESI+1C],EBX    ; azzera
00A828A3  MOV BYTE PTR [ESI+1C],DL      ; salva il quinto byte
00A828A6  MOV DWORD PTR [ESI+34],EAX    ; salva EAX
```

Dopo aver speso diverso tempo a cercare di capire il significato di ogni singolo byte, sono giunto alla conclusione che servono solo i primi 4. Gli altri sembra che non vengano mai richiamati, ma ammetto che posso sbagliarmi...
A parte questo anche gli altri bytes mi hanno dato parecchio filo da torcere, ma arrivo a dire con certezza che il primo byte deve essere 0 (anche se non c'è un controllo esplicito) e che il terzo e il quarto sono legati alla data di scadenza del seriale.
Il secondo byte invece mi lascia un po' perplesso... poiché può assumere diversi valori, ma in alcuni casi dove ero sicuro che dovesse funzionare, ho ricevuto picche...
Non sono riuscito ad approfondire completamente il discorso, ma sembra che in base al seriale in “funzione” dipenda il funzionamento di quello nuovo.

Proseguiamo nel nostro codice e vediamo cosa succede:

```
00A828A9  JE SHORT 00A828BE    ; controlla l'operazione di poco fa su EAX
00A828AB  MOV EDX,ECX
00A828AD  SHR EDX,7
00A828B0  AND EDX,1F
00A828B3  AND ECX,7F
00A828B6  MOV DWORD PTR [ESI+38],EDX
00A828B9  MOV DWORD PTR [ESI+30],ECX
00A828BC  JMP SHORT 00A828CD
00A828BE  AND ECX,0FFF
00A828C4  MOV DWORD PTR [ESI+38],EBX
00A828C7  MOV DWORD PTR [ESI+30],EBX
00A828CA  MOV DWORD PTR [ESI+2C],ECX
00A828CD  LEA ECX,DWORD PTR [ESP+18]
00A828D1  MOV DWORD PTR [ESP+40],EBP
00A828D5  CALL 00A816C0                ; nulla di interessante
00A828DA  MOV EAX,1
00A828DF  MOV ECX,DWORD PTR [ESP+38]
00A828E3  POP EDI
00A828E4  POP ESI
00A828E5  POP EBP
00A828E6  MOV DWORD PTR FS:[0],ECX
00A828ED  MOV ECX,DWORD PTR [ESP+28]
00A828F1  POP EBX
00A828F2  CALL 00A88DAE                ; nulla di interessante
00A828F7  ADD ESP,34
00A828FA  RETN
```

Niente di speciale a parte un controllo sul terzo e quarto byte dopo lo SHIFT e l'AND. Per ora non ce ne preoccupiamo e andiamo avanti, basta che teniamo a mente che è presente un controllo. Passiamo il RET e troviamo un'altra call poco interessante... Ignoriamola e continuiamo fino al prossimo RET.
Ritorneremo abbastanza a monte e precisamente qua:

```
00A83073  CALL 00A82980                 ; controllo ultimi 4 chars + blowfish
00A83078  CMP EAX,1
00A8307B  JE SHORT 00A830AF             ; preso se ultimi 4 chars corretti
00A8307D  LEA ECX,DWORD PTR [ESP+48]
00A83081  MOV BYTE PTR [ESP+CC],0
00A83089  CALL 00A82950
00A8308E  LEA ECX,DWORD PTR [ESP+84]
00A83095  MOV DWORD PTR [ESP+CC],-1
00A830A0  CALL 00A82950
```

```
00A830A5  MOV EAX,3
00A830AA  JMP 00A83787
00A830AF  MOV ECX,DWORD PTR [A9DB00]     ; agentreg.00A9C280
00A830B5  MOV EAX,DWORD PTR [ECX+210]    ; valore per lo switch
00A830BB  ADD EAX,-29                    ; Switch (cases 29..74)
00A830BE  CMP EAX,4B
00A830C1  PUSH EBX
00A830C2  PUSH EBP
00A830C3  JA 00A8370C
00A830C9  MOVZX EDX,BYTE PTR [EAX+A837C4]
00A830D0  JMP DWORD PTR [EDX*4+A837AC]
```

Uscendo dalla call viene controllato subito il valore di EAX. Viene settato all'interno della call da cui siamo appena usciti e dipende dalla correttezza degli ultimi 4 caratteri. I nostri sono già corretti, poiché venivano controllati anche in precedenza. Se dovessero essere sbagliati, vengono eseguite 2 calls e si salta lontanissimo.
Non è il nostro caso, quindi continuiamo da 00A830AF. Poche istruzioni che servono a pescare un valore presente a [ECX+210], a sottrargli 29, a compararlo con 4B ed eventualmente usarlo per determinare la destinazione dello switch.
Questo è il punto che mi ha fatto un po' impazzire (come se non lo fossi già abbastanza!), poiché se non è già presente un seriale viene pescato il numero 37, se invece è già presente un seriale il valore pescato dipende dal secondo byte (di cui abbiamo parlato poco sopra) ottenuto dalla cifratura.
Quindi in funzione di seriali “vecchi” cambierà la destinazione dello switch dando vita a molteplici controlli condizionati che non starò ad esaminare nel dettaglio.
Diamo per scontato che viene passato il 37, di conseguenza atterreremo qua:

```
00A8370C  MOV EDX,DWORD PTR [A9DB00]           ; Default case of switch
00A83712  PUSH 208                             ; /maxlen = 208 (520.)
00A83717  ADD EDX,4                            ; |
00A8371A  PUSH EDI                             ; |src
00A8371B  PUSH EDX                             ; |dest
00A8371C  CALL DWORD PTR [<&MSVCR71.strncpy>]  ; \strncpy
00A83722  MOV ECX,DWORD PTR [A9DB00]           ;  agentreg.00A9C280
00A83728  ADD ESP,0C
00A8372B  PUSH 0                               ; /Arg2 = 00000000
00A8372D  LEA EAX,DWORD PTR [ECX+4]            ; |
00A83730  PUSH EAX                             ; |Arg1
00A83731  CALL 00A87120                        ; \agentreg.00A87120
00A83736  MOV ECX,DWORD PTR [A9DB00]           ;  agentreg.00A9C280
00A8373C  CMP DWORD PTR [ECX+20C],3            ; cmp dopo una call (sospetto)
00A83743  JNZ SHORT 00A83750
```

Nella prima call viene semplicemente copiato il nostro seriale in un'altra locazione. Nella seconda non sappiamo ancora cosa viene fatto, ma subito dopo è presente un controllo, quindi vale la pena entrarci e vedere che succede.
All'inizio ci saranno 4 calls di poca importanza e poi un salto che ci porterà qua:

```
00A872C8  PUSH ESI                             ; /Arg1
00A872C9  LEA ECX,DWORD PTR [ESP+68]           ; |
00A872CD  CALL 00A82980                        ; \agentreg.00A82980
```

```
00A872D2  CMP EAX, 1
00A872D5  MOV EDI, DWORD PTR [<&KERNEL32.GetPrivateProfileIntA>]
00A872DB  JE SHORT 00A87305
00A872DD  PUSH 3
00A872DF  PUSH 00A972DC
00A872E4  MOV DWORD PTR [ESI+21C], EBX
00A872EA  MOV DWORD PTR [ESI+208], 3
```

Questa call è la stessa usata per calcolare e controllare gli ultimi 4 chars e dove è presente blowfish.
Subito dopo c'è il controllo su EAX come in precedenza. E' una verifica che effettivamente non stiamo barando. Visto che per ora siamo in regola, possiamo passarla tranquillamente e prendere il salto condizionale automaticamente ;-)
Date comunque uno sguardo al codice se il salto non dovesse venir preso. Verrebbe messo un 3 in [ESI+208] e se vi ricordate all'uscita della call in cui ci troviamo c'è:

```
00A8373C  CMP DWORD PTR [ECX+20C],3
```

Ok i puntatori non sono uguali, ma la cosa puzza parecchio...
Teniamolo a mente e andiamo avanti:

```
00A87305  |>  8D4C24 64     LEA ECX,DWORD PTR [ESP+64]
00A87309  |.  E8 42B4FFFF   CALL 00A82750
00A8730E  |.  83F8 71       CMP EAX,71
00A87311  |.  8986 0C020000 MOV DWORD PTR [ESI+20C],EAX
00A87317  |.  74 37         JE SHORT 00A87350
00A87319  |.  83F8 72       CMP EAX,72
00A8731C  |.  74 32         JE SHORT 00A87350
00A8731E  |.  83F8 31       CMP EAX,31
00A87321  |.  74 2D         JE SHORT 00A87350
00A87323  |.  83F8 32       CMP EAX,32
00A87326  |.  74 28         JE SHORT 00A87350
```

Un'altra call e 4 controlli in sequenza. Dalla call usciamo con EAX = 00006075
Ve li ricordate questi 2 bytes? Sono rispettivamente il primo e il secondo valore della sequenza creata dalla cifratura.
Se date uno sguardo generale al codice, troverete parecchi controlli identici e vengono sempre confrontati questi 2 bytes.
Purtroppo, come dicevo prima, il flusso è condizionato dallo stato di registrazione. Non sono in grado di prevedere il percorso per ogni situazione e non posso analizzare singolarmente ogni caso, altrimenti non finirei più!

Continuando per il flusso, che spero sia quello “normale”, si arriverà a un'altro switch ed entreranno in gioco ancora questi 2 bytes:

```
00A8773E  MOV EAX,DWORD PTR [ESI+20C]          ; i 2 famosi bytes in EAX
00A87744  ADD EAX,-28                          ; Switch (cases 28..72)
00A87747  CMP EAX,4A
00A8774A  JA SHORT 00A87792
00A8774C  MOVZX EAX,BYTE PTR [EAX+A881A8]
00A87753  JMP DWORD PTR [EAX*4+A881A0]
00A8775A  CMP DWORD PTR [ESI+21C],EBX          ; Cases 28,29,2A,2B,2C,2F,30
00A87760  JLE SHORT 00A8777A                   ; 31,32,68,69,6A,6B,6C,6D,6E
00A87762  PUSH 1                               ; 6F,70,71,72
```

```
00A87764  PUSH 00A972DC
00A87769  MOV DWORD PTR [ESI+208],1
00A87773  CALL 00A90B80
00A87778  JMP SHORT 00A877AE
00A8777A  PUSH 2
00A8777C  PUSH 00A972DC
00A87781  MOV DWORD PTR [ESI+208],2
00A8778B  CALL 00A90B80
00A87790  JMP SHORT 00A877AE
00A87792  PUSH 3
00A87794  PUSH 00A972DC
00A87799  MOV DWORD PTR [ESI+208],3
00A877A3  CALL 00A90B80
```

Vengono messi i 2 bytes in EAX e preparati per lo switch. Dopo la sottrazione avremo EAX = 0000604D quindi andremo a finire forzatamente a 00A87792. E che ci troviamo?

```
MOV DWORD PTR [ESI+208],3
```

Ancora una volta possiamo pensare che sarà il valore controllato all'uscita dalla call, quindi la direzione apparentemente non è quella giusta!
Dobbiamo riuscire ad arrivare nella condizione di finire in uno dei “Cases” e per riuscirci i 2 bytes della sequenza devono valere: il primo 00 (per forza!) e il secondo uno qualsiasi dei valori presenti nei “Cases”.

KEYGENNING

E' arrivato il momento di cominciare a scrivere il keygen, altrimenti non ce ne tiriamo più fuori!
Sappiamo già abbastanza per poter cominciare a scrivere qualcosa, poi saremo sempre in tempo a modificare delle parti per poter studiare meglio i risultati.

Per cominciare vediamo di cosa abbiamo bisogno e come strutturare il programma.
Principalmente abbiamo bisogno di qualcosa per programmare.
Ho scelto MASM32!
I motivi sono che mi piace scrivere in assembly, perché non ho trovato un'implementazione di blowfish in asm e volevo complicarmi la vita e perché non sono un gatto con gli altri linguaggi...
Perlomeno l'assembly lo conosco discretamente bene quindi, a parte qualche grande svarione nel codice (ce ne saranno), in un modo o nell'altro riesco sempre a cavarmela.

Cominciamo a pensare alla struttura.
Dunque, una semplice dialog può bastare, ci sarà bisogno delle S-Boxes prima dell'inizializzazione, delle P-array già inizializzate (per non far sprecare tempo al programma), della maschera per il calcolo degli ultimi 4 chars del seriale e di una routine per il calcolo, della routine di inizializzazione e della routine di cifratura.

Le S-Boxes le ho recuperate dal file Blowfish.dat presente nel sorgente di

Bruce Schneier, poi l'ho riformattato escludendo le P-Array (che non mi servivano) e cambiando la sequenza dei bytes delle dwords (ordine invertito), altrimenti era un lavoro che dovevo far eseguire al programma facendogli perdere tempo.
Il percorso standard che ho scelto per Blowfish.dat è “C:\”. Se volete cambiarlo vi basta mettere mano al sorgente.

Le P-Array dobbiamo solo copiarle da BitDefender e la maschera pure.
La routine di calcolo per gli ultimi 4 caratteri l'ho ripresa tale e quale da BitDefender, ma ho dovuto aggiustare i puntatori.
La routine di inizializzazione l'ho copiata tale e quale da Cryptool (puntatori a parte). Stesso discorso per la routine di cifratura.

Durante l'apertura della Dialog eseguiremo:

- la lettura delle S-Boxes dal file blowfish.dat
- l'inizializzazione delle S-Boxes

Quando premeremo il bottone per generare il seriale eseguiremo:

- la generazione di un seriale
- la cifratura
- il calcolo degli ultimi 4 chars
- la visualizzazione

Cerchiamo di avere una visione globale del problema.
Sappiamo che dopo la cifratura dobbiamo avere una sequenza di bytes di questo tipo:

00 xx yy yy zz zz zz zz

dove 00 è fisso, xx dovrà corrispondere a uno dei valori presenti nel “Case”, yy sono legati alla data di scadenza del seriale, zz sono dei valori casuali.

Il mio primo tentativo è stato quello di creare un seriale totalmente casuale, invertire l'ordine delle P-Array (come fa BitDefender), farlo criptare e controllare che la sequenza creata soddisfasse i requisiti. Praticamente un bruteforcing.

Metodo efficace, ma alquanto distruttivo, poiché prima di trovare qualcosa di “buono” passavano minuti!!!!

Poi mi è venuto il colpo di genio! Hehehe

Sappiamo che la decifratura dell'algoritmo è tale e quale alla cifratura, solo che le P-Array vengono utilizzate in ordine inverso. Allora perché non fare lo stesso?
Parto dal risultato che ho bisogno, eseguo la decifratura e ottengo il seriale che

genererà il risultato di cui ho bisogno.
Tra l'altro non devo neanche invertire l'ordine delle P-Array! Perfetto!

La routine di poche righe che ho usato per creare il seriale è la seguente:

```
mov dword ptr [serial], 0yyyyxx00h
invoke GetTickCount
mov dword ptr [serial+4], eax
ret
```

Forzo la prima parte con valori che mi serviranno per passare i controlli i BitDefender, mentre la seconda parte la lascio al caso.
Questo mi ha permesso di giocare allegramente con i valori e verificare le conseguenze nel programma abbastanza facilmente. Malgrado ciò non sono riuscito a determinare completamente il funzionamento.
Posso solo dirvi che se xx = 2B (o 28 o 29 o 69) vi riuscirete sempre a registrare.
yyyy invece corrispondono ai giorni di validità della licenza.

Se passate per esempio 50 00 2B 00h la vostra licenza durerà 0050h = 80 giorni.
Se passate per esempio 26 02 2B 00h la vostra licenza durerà 0226h = 550 giorni e così via

Questo controllo non l'ho capito direttamente dal debugging del programma, ma l'ho intuito a furia di far prove!

Un' ultima cosa che dovete sapere è che i dati di registrazione sono salvati in win.ini nella sezione [internal].
Se provate a generare dei seriali mettendo 28 al posto di xx non avrete più la possibilità di immettere altri seriali, poiché scompare la voce. Vi basterà cancellare la sezione [internal] e i sottovalori per ripristinare il tutto.

CONCLUSIONE

Che dire? Indicativamente ho finito...
Mi rendo conto che è tutto abbastanza incasinato, ma per fregare questo target ho dovuto lavorare parecchio in sintonia tra debug di Bitdefender, debug di Cryptool e scrittura del keygen.
Descrivere la procedura seguendo un filo logico, che neanch'io ho seguito, non è per niente facile...
Inoltre, malgrado Quequero ci ha avvisato in anticipo del new year pack 2007, il tempo è sempre tiranno e non basta mai!

In un prossimo futuro conto di metter mano alla nuova versione (l'ho già guardata nel dettaglio!)e magari di partire dalla base di questo articolo pe rscrivere qualcosa di più dettagliato e meno dispersivo.

Concludo augurando un ottimo 2007 a tutti quanti e vi ricordo che, anche se

l'articolo non vi è piaciuto, noi abbiamo comunque investito tempo perché vi vogliamo bene! Hihihi

SatUrN