Krypton 2 Patching the Beast!		
Data	by "**AndreaGeddon**"	
31/03/2001	***UIC's Home Page***	**Published by Quequero**
Efficiunt daemones, ut quae non sunt, sic tamen...	*Qualche mio eventuale commento sul tutorial :)))* *Se la smettessi di solarci te lo potrei far rivincere il titolo!!* 1) Sei un traditore, il krypton lo fai e lo strainer no...Vabbè questa la lego al dito 2) Come osi brutto leim scrivere nello spazio riservato a ME???? :) 3) La prossima volta se per il codice non usi SEMPRE un carattere a spaziatura fissa ti rompo i denti :) 4) Io non ti solo è che a volte sono eccessivamente impegnato e SOLARE vuol dire non presentarsi, mica chiamare in modo cordiale come faccio io e dire: "Andre scusa ma non mi è possibile stasera" ....Hehehehe beccati questa :)	...quasi sint, conspicenda hominibus exibehant.
....	**Home page se presente:** http://www.andreageddon.com **E-mail:** andreageddon@hotmail.com **irc.azzurra.it / irc.tin.it #crack-it**	....
Difficoltà	( )**NewBies** ( )**Intermedio** ( )**Avanzato** (x)**Master**	

Tutti ci abbiamo provato. Tutti ci abbiamo sbattuto sopra i denti. Tutti abbiamo consumato il tasto del reboot. Tutti abbiamo maledetto Yado e il K2. Il Krytpon fluttuava nel suo flat: inattaccabile, imperscrutabile, indisassemblabile, questa creatura rideva in faccia al softice e ai suoi ridicoli tentativi di attacco: ogni attacco significava inevitabilmente la morte in quel mare di bytes dominati dal K2, che sfidava ogni legge di reversing. Ma ecco un animo tenace armare a punto il suo debugger e sferrare finalmente un unico attacco: letale, continuo, da ogni lato, senza lasciare spazio a nessuna via di fuga. Il prode Krypton fu colpito a morte, gli fu stroncata la possibilità di fuggire a ring0 mentre il suo codice veniva violato sempre più a fondo... l'ultimo fugace tentativo fu di scappare sulla strada lastricata del globalalloc, ma il softice lo inseguì senza tregua alcuna, e ormai l'innalzamento delle SEH erano un ultimo disperato rantolo del K2, rantolo che ormai preannunciava l'inevitabilità. E la morte lo colpì. Quel fatal JLE fu trovato e il K2 non ebbe scampo: e lo videro sprofondare morente nel verde siliceo tra una segment e l'laltro, risucchiato dal vortice del paging . Poi nulla più. Ma la morte del K2 non ha segnato la fine... un nuovo mostro è in attesa, una beta embrionale che è già preludio di un crisalide: schiusa la sua vecchia pelle di K2 si prepara a ergersi maestoso e inarrestabile in quell'array di celle di memoria che diventeranno la sua casa; chissà, un giorno forse potrebbe prevalicare anche l'ultima realtà del pmode....

Krypton 2
Patching the Beast!

Written by **AndreaGeddon**

Introduzione

Dopo il meet da ZeroByte siamo ripartiti con 30 litri di vino in meno e un gran mal di testa in più. Eppure avevo il tarlo nell'orecchio: una promessa in chat fatta tempo fa... di sconfiggere il K2 una volta per tutte. Queste parole suscitarono l'ilarità, non per me che presi la cosa sul serio. Al momento ero impegnato a debellare safedisc, e intanto il k2 mi rodeva... una ferita sempre aperta. Ora è morto, e aspetto con ansia la release definitva del Krypter. Dalle beta che ho visto ho proprio paura che Yado non ci deluderà.

Tools usati

SoftIce per un pelo
Un HexEditor
Un PeDitor
Nervi Saldi!

URL o FTP del programma

quequero.cjb.net o www.lockless2k.com

Notizie sul programma

Lo avviamo senza softice e ci dice che il trial è spirato. Quindi un Time Trial, ma condito di belle cosette come codice criptato, modifica degli interrupt, anti-sice, controlli di integrità sul codice, decrypting dinamico, decrypting su più layer, uso e abuso di seh, anti bpx/bpm...

Essay

Dopo innumerevoli tentativi di breaking ho convenuto che l'unica soluzione è quella di passare il K2 COMPLETAMENTE al setaccio. Vi sarete chiesti anche voi "ma come cazzo fa?", ora daremo una risposta a tutto. Questo crypter è un pò diverso da quelli convenzionali, in quanto il loader non si occupa di svolgere tutto il lavoro di decrypt, ma tale lavoro continua anche nella sezione di codice. Questo è un semplice crackme, non un programma con un corpo vero e proprio, quindi il codice stesso del crackme è stato riusato come loader. Iniziamo dalla prima analisi: caricate un PEditor e diamo una sondata al PE di questa belva. Guardiamo le imports: tutto a posto. C'è una IT valida. La IT in questione è quella del programma originale, non quella del loader. Beh cmq niente giochi strani sulle import. Ora guardiamo le sezioni:

SEZIONE	VSIZE	VOFFSET	RSIZE	ROFFSET	FLAGS
CODE	00001000	00001000	00001000	00000600	60000020
DATA	00001000	00002000	00000600	00001600	C0000040
.idata	00001000	00003000	00000200	00001C00	C0000040
.reloc	00001000	00004000	00000200	00001E00	50000040
.rsrc	00001000	00005000	00000A00	00002000	50000040
Yado	00002000	00006000	00002000	00003000	E0000080

anche qui per ora nulla di strano. Lo stile sembra quello di un programma compilato con TASM. Vediamo le caratteristiche delle sezioni, in particolare la prima. La CODE è la sezione principale di codice, ed è criptata, però ha come flags
00000020 + 40000000 + 60000000
cioè sezione di codice, eseguibile, di lettura. Spesso i crypter sostituiscono le caratteristiche delle sezioni con il solito E0000020, in modo da ottenere l'accesso in scrittura sulla sezione desiderata. La CODE così com'è non può essere sovrascritta (dal decrypting) normalmente, perchè se ci proviamo da ring3 finiremo in un fault. Questo ci potrebbe portare qualche problema, visto il metodo che ho scelto di seguire per l'attacco del k2, indi settiamo le caratteristiche delle sezioni tutte a E0000020. Ora non ci rimane che iniziare lo stepping. Come tutti sappiamo, il k2 fa di tutto per non lasciarci steppare, al minimo errore CRASH! il programma infatti sostituisce gli int 1 3 e 5, e li usa per scendere a ring0, da dove provoca i bei crash se si accorge che stiamo tentando di fregarlo. Quindi partiamo col presupposto di rimanere sempre a ring3 per evitare blocchi indesiderati, e cercheremo di lasciare sempre intatti i vettori degli interrupt 1 e 3, in modo da poter usare il softice normalmente. Okei, data questa piccola overview ora si comincia. Aprite il softice loader e carichiamo il K2. Eseguiamolo. Arriveremo sull'entry point a 00406000, proprio sul primo byte della sezione Yado. Iniziamo a studiarci il loader. Subito arriviamo al primo problema:

```
0040603C      PUSH ECX
```

```
            CALL 00406032
```

che succede qui? Se steppate la call con F10 avrete il solito effetto catastrofico. Quindi in 00406032 che succede? Entriamoci con F8 e vediamo:

```
POP ECX          ottiene i return address pushato dalla call
PUSHFD           pusha registri e flags
ADD ECX, -19         modifica il return address
POPFD            poppa registri e flags
JMP ECX          salta al return address modificato
```

non abbiamo un RET specifico, quindi in realtà la CALL è solo una finta call. Infatti subito dopo la call il return address viene preso dallo stack e messo in ECX, dove viene decrementato di 19h e poi il controllo viene spostato a questo valore dal jump ecx. Questa struttura non ha importanza ai fini del loader, serve solo per confondere le acque e per rendere più difficile le cose a noi che steppiamo. Ovviamente questa struttura è ripetuta praticamente in modo continuo, quindi armatevi di F8 e steppate sempre con quello. Vedrete passare sotto i vostri occhi parecchie linee, ma quelle che ci interessano in realtà sono poche. Ora le vediamo singolarmente.

00401620   CLI

questo disabilita l'interrupt flag e quindi i relativi maskable hardware interrupts. A livello immediato non comporterà problemi, potremo cmq usare gli int 1 3 e 5, solo che il cli ci potrebbe dare problemi con il paging, sempre legati al fatto che non andremo a ring0. Quindi per precauzione è meglio evitare di eseguire i vari cli. Le righe che non vogliamo eseguire non dovete nopparle, ma semplicemente digitiamo in EIP l'indirizzo della riga successiva. Proseguiamo.

00406224   MOV [EBX], EAX
...
0040626E   MOV [EBX+6], EAX

queste due righe sostituiscono il vettore dell'int 1 con un vettore del krypton2, precisamente il vettore che si trova alla linea 00406BBA. Questa modifica è assolutamente da evitare dato che l'int 1 è l'interrupt del SingleStep, cioè viene generato per l'esecuzione di ogni riga mentre steppiamo. Se provate a eseguire le 2 righe qui sopra steppandole col softice riceverete una schermata blu :-). Se invece le righe vengono eseguite senza stepping, allora l'int 1 viene ridefinito e saranno cavoli perchè ci manderà al diavolo il softice. Quindi anche queste due righe vanno saltate.

00406303   CLI
...
00406407   MOV [EBX], EAX
...
00406451   MOV [EBX+6], EAX

anche queste righe vanno saltate. Il CLI lo abbiamo già visto. Le altre due righe stavolta sono la sostituzione dell'int 3 con un handler del k2, handler che sta alla riga 00406C44. L'interrupt 3 è usato per settare i breakpoint, quindi se lasciamo modificare questo interrupt non possiamo più usufruire dei break e delle funzioni annesse (tipo HERE). Se provate a breakare e l'int 3 è modificato, al momento del break invece di invocare il softice finiremo nell'handler del K2 che potrà fare quello che vuole (tipo freezarci :-)). Quindi saltiamo anche queste.

004065F6   CLI
...
0040671C   MOV [EBX], EAX

...
00406766 MOV [EBX+6], EAX

ancora! Il cli lo saltiamo, le altre due istruzioni servono per redirigere l'INT 5. L'int 5 non è fondamentale per il debugger o per il sistema, quindi in teoria non avremmo problemi a lasciar sostituire l'handler. Certo però che tramite l'handler dell'int finiremo a ring0, e da lì un errore qualsiasi potrebbe seccarci, quindi ci segnamo l'handler nuovo dell'int 5 (alla riga 00406D12) e saltiamo queste due righe. Per ora lasciamo sempre la idt intatta.

004067AE INT 5
004067B0

eccolo qui! La chiamata all'int 5 ci dovrebbe spedire nell'handler del k2 e ci dovrebbe mandare a ring0. Non possiamo eseguire l'int 5 perchè non glielo abbiamo lasciato modificare, quindi arrivati sulla riga dell'int 5 cambiamo a mano l'EIP e ci mettiamo l'address 00406D12. Ricordate che il codice dove ci troviamo ora era stato pensato per essere eseguito a ring0, mentre ora siamo a ring3: infatti vedrete l'uso dei Debug Register, il sice potrebbe comportarsi in modo strano, tipo salta qualche riga. Non spaventatevi, è normale visto che l'accesso ai DR e i CR è consentito solo da ring0 (dove avremmo dovuto essere:-P). Ma perchè accede ai DR? Nei DR vengono salvate le informazioni dei BPM (DR0-DR3 = 4 bpm possibili). Con l'accesso ai DR semplicemente il k2 controlla se sono stati settati dei bpm, quindi dopo ogni riga del tipo:

MOV dest, DRx

assicuratevi che il registro dest sia a zero. In tal caso il sice non troverà bpm istallati e continuerà senza scocciare (non che io ne avessi istallati di bpm...). Steppate e arrivate a qualche riga sospetta:

00406F5E MOV EAX, 00401000
...
00406FA7 MOV ECX, 00402000

cosa possono essere se non gli estremi della sezione CODE? Quindi ci aspettiamo un decrypt. Proseguiamo fino al ciclo:

0040707D XOR [EAX], EBX ebx = 01012000
...
          MOV EDX, [EAX]
...
00407109 ROR EDX, 02
...
00407196 XOR [EAX], EBX
...
00407223 CMP EAX, ECX
...
00407269 JLE 00407039
...
004072B3 IRETD

il ciclo di decrypt del codice. EAX contiene 00401000 e viene incrementato di 1 dword alla volta, ecx contiene 00402000, ebx contiene la xormask 01012000. Una volta finito il decrypt arriviamo all'IRETD, cioè Interrupt Return, che ci fa ritornare alla riga successiva a quella in cui veniva invocato l'int 5 (già, proprio come una call :-)). Ma noi l'int 5 non l'avevamo invocato, quindi adesso che siamo sull'iretd cambiamo l'EIP a mano e ci mettiamo 004067B0.

0040688B MOV [EBX], DX ripristina int 1

```
...
004068D5  MOV  [EBX+6], DX      ripristina int 1
...
00406992  MOV  [EBX], DX         ripristina int 3
...
004069DC   MOV  [EBX+6], DX     ripristina int 3
...
00406A99  MOV  [EBX], DX         ripristina int 5
...
00406AE3   MOV  [EBX+6], DX      ripristina int 5
...
00406B30   JMP   EAX       eax = 00401000
```

dopo aver decriptato il codice, il k2 ripristina gli interrupt modificati (anche qui saltate le righe del ripristino) e poi arriviamo al classico JMP EAX, dove eax guarda caso è 00401000. Quindi il loader a questo punto è terminato. Almeno la sezione Yado :-). Ora andiamo nel codice vero e proprio del crackme, ma come vedremo i nostri guai non sono affato finiti! Intanto dobbiamo aspettarci ancora il decrypt della sezione DATA, quindi steppiamo:

```
00401697  MOV  EAX, 00002000
```

e già questo è un campanellino di allarme: viene presa la base della sezione dei dati. Proseguiamo...

```
004016A8  XOR  [EAX], EBX
...
004016B3  JGE   004016A8
```

sull'algoritmo di decrypt dei dati non ci soffermiamo molto, visto che per i nostri scopi non è fondamentale.

```
0040178F   CLI
...
00401871   MOV   [EBX],    EAX
...
004018BB   MOV     [EBX+6], EAX
```

ecco l'ennesima sostituzione dell'int 3 con un handler che si trova a 00401D7F. Saltiamo le due righe così siamo a posto.

```
00401908   CALL  x       GetModuleHandle  -> user32.dll
...
0040197E   CALL  x      GetModuleHandle  ->   kernel32.dll
```

ecco che vediamo qualcosa di interessante. Qui viene preso l'handle delle due librerie tramite GetModuleHandle, quindi dobbiamo aspettarci che ora verrà preparata la IT del loader. La x come argomento della call sta a significare che la call non era in chiaro, credo che fosse un [eax], non ricordo :-(. Non abbiamo incontrato il LoadLibrary perchè i due moduli erano già presenti nella IT del K2 (ricordate all'inizio lo sguardo col PEditor??), quindi ora sono già mappati nello spazio del processo del K2. Ora aspettiamo solo il GetProcAddress.

```
00401A1C  CALL  x   call a GetProcAddress per ottenere l'address di GetSystemTime
...
00401ABA  CALL x   come sopra ma per la funzione MessageBoxA
...
00401B58  CALL x    funzione MessageBoxExA
```

...
00401BF6 CALL x funzione ExitProcess
...
00401C94 CALL x funzione GlobalAlloc

queste sono tutte chiamate a GetProcAddress per ottenere l'address delle relative funzioni scritte. Tutti gli address vengono messi in un array alla locazione 0040214C dal quale poi il k2 le richiamerà all'occorrenza. Bene, finiti i preparativi per il loader della sezione code ora arrivano i trick più bastardi.

```
0040163E      MOV   AH, 43
            INT   68
            CMP   EAX, F386
            JZ      00401D2B
```

ecco la prima cosa banale finora! La chiamata all'int 68 con funzione 43 è uno dei modi per determinare la presenza del debugger, e se il debugger è presente viene restituito in eax il valore F386 (ma vaaaa). Quindi è chiaro che al JZ non dovete saltare. Mi raccomando non noppatelo, cambiate solo il flag z a runtime (capirete perchè). Cmq ora possiamo prendere una boccata d'aria, infatti questo int è una debolezza che ci permette di "salvare la partita", una specie di checkpoint :-). Ormai tutti sanno il trucco del
bpx exec_int if al == 68
che ci fa breakare direttamente all'int 68 senza doverci steppare tutto il loader come ho fatto io. Quindi in qualsiasi momento potrete riprendere la vostra analisi da qui. C'è un problema però. A breakare breaka, ma l'handler dell'int 3 è stato sostituito, quindi in pratica vi ritroverete fregati in men che non si dica. Per soprassedere a questo inconveniente è sufficiente che prima di lanciare il k2 apriamo il softice, digitiamo IDT e andiamo a salvarci i bytes del descrittore dell'int 3. Basterà prendere la Base della idt (800A7000 per me) e aggiungergli 18h (24), infatti i descrittori sono tutti da 8 bytes. Così saltiamo gli int 0 1 2 e vediamo l'int3. Scrivete gli 8 byte da qualche parte, poi quando breakate con l'exec_int andateli a ripristinare nell'idt. Ora scrivete un IDT 3 e assicuratevi che l'handler sia quello corretto del win. Ora si può proseguire.
Innanzitutto guardiamo cosa succede nel caso in cui il programma trova il softice in memoria:

```
00401E09      MOV   EAX, 0040214C    mette in eax l'array della iat
            ADD   EAX, 08            va all'address di messageboxa
            CMP   [EAX], CC        cerca se è stato installato un bpx
```

controlla se è presente il byte CCh nei primi bytes all'entry point della funzione MessageBoxA, il che vuol dire che cerca se per caso abbiamo inserito un bpx su tale funzione. Che malvagità! Non basta che l'int3 sia manomesso, controlla ancora che non ci sia un break sulla MessageBox... da notare che quando eseguite il crackme col softice presente, dopo la messagebox di errore l'int 3 rimane alterato, così crasherete a qualsiasi tentativo di break: che bastardo. Superato questo int 68 troviamo:

```
00401655     CLI
0040166B     MOV   [EBX], AX
00401671     MOV   [EBX+6], AX
```

la solita redirezione dell'int 5 verso l'handler a 004016B6. Saltiamo tutto.

```
00401675  INT 5
00401677
```

ora viene chiamato l'int 5, quindi come al solito spostate l'eip a mano verso il relativo handler visto poco fa e segnatevi il return address. Ecco cosa fa questo int 5:

```
004016C2    ADD  ECX, [EAX]
          INC  EAX
```

```
          CMP  EAX, EBX
          JLE  004016C2
```

dove eax = 00401634 e ebx = 00401695. Un semplice checksum su una piccola regione di codice, quella che si estende da 00401634 a 00401695. Il checkusm viene salvato in eax. Non viene controllato, ma più infidamente viene usato per il prossimo ciclo:

```
004016C9  MOV    ESI, 004011AF   Importante! Ricordate questo address
          MOV  EBX, 0482
004016D3  XOR    [EAX], ECX
          ADD  EAX, 04
          SUB  EBX, 04
          CMP  EBX, 00
          JGE  004016D3
          IRETD
```

il checksum viene usato per xorare 482h bytes di codice a 004011AF. Ricordate questo indirizzo perchè fra poco lo ritroveremo. Btw, ecx deve essere 2F2ED552 per xorare correttamente il codice. Ecco perchè vi ho fatto semplicemente saltare le linee bastarde invece di nopparle :-9. Dopo l'iretd settiamo l'eip al return point e passiamo alla seconda facciata dei miei appunti, cioè alla riga

```
00401686   MOV    [EBX], DX
...
0040168C   MOV    [EBX+6], DX
```

e qui viene ripristinato l'int 5. Saltiamo tutto, non si sa mai.

```
00401DE8  MOV    EAX, 00401000
          MOV  EBX, 004011A7
          XOR  ECX, ECX
00401DF4  ADD    ECX, [EAX]
          INC  EAX
          CMP  EAX, EBX
          JLE  00401DF4
          CMP  ECX, 421C4B97
          JNZ  00401D2F
```

ecco un altro checksum, che se sbagliato ci manda alla stessa routine di errore vista prima per la presenza del softice. Anche qui se non avete toccato niente il programma dovrebbe funzionare a dovere.

```
004016E1    PUSH  00401D06
          PUSH  dword ptr FS:[0000]
          MOV   FS:[0000], ESP
```

mi stavo giusto chiedendo quando sarebbe arrivato! Ecco il nostro bel SEH. L'handler è alla riga 00401D06, segnatevelo che poi vi tornerà molto utile.

```
0040100E  CLI
...
00401024   MOV  [EBX], AX
...
0040102A   MOV  [EBX+6], AX
...
```

00401030 INT 5
00401032

eccoci di nuovo alla solita struttura. Il CLI saltatelo, al contrario stavolta non facciamo saltare le due righe successive che servono a istallare un handler sull'int 5 per la irga 004010E1. Qui avevo sbagliato e avevo saltato anche la riga che salvava l'address dell'int originale, ed ero poi occorso in un fault. Quindi alla successiva esecuzione avevo lasciato modificare l'handler dell'int 5, quindi adesso proseguiamo con l'int 5 modificato. Tuttavia il proseguimento con l'handler originale non dovrebbe comportare nessuna differenza. Quando siete sull'int 5 anche se l'handler dell'int è modificato è meglio andarci a finire cambiando l'eip a mano, così rimarremo nel nostro sicuro ring3.

```
004010A6  DEC    ECX
          JZ   004010D2
          MOV  EBX, [EAX]
          ADD  EAX, 04
          CMP  EBX, 544E4950 -> "TNIP"
          JNZ  004010A6
```

ecco qui il primo casino. Questo ciclo itera per un numero ecx di volte, ora non ricordo il valore esatto ma erano circa 450'000 cicli. Possiamo provare a mettere una riga in tutte le possibili vie di fuga di questo ciclo, cioè 004010D2 e 004010C8, infatti non ho scritto le ultime 2/3 righe, cmq non cambia di molto. Il ciclo può evadere solo in quelle 2 righe. Settiamo quindi i break (occhio che l'int 3 sia quello corretto), lasciamo runnare... DING! Your trial is expired. Azzo, il programma ha proseguito normalmente e ci ha fregati. Notate ora che anche avendo il softice il programma ha proseguito come se in realtà sice non ci fosse. Questo vuol dire che siamo sulla strada giusta. Solo che questo ciclo è un pò strano. Ricordate il SEH istallato poco fa? Bene. Torniamo a quel ciclo e settiamo un breakpoint sul seh, cioè alla riga 00401D06. Lasciamo correre il programma e il softice breakerà subito sulla riga indicata. Il ciclo in qualche modo genera una eccezione, anche se non ho ancora capito come. Cmq ora abbiamo capito come uscirne, quindi proeguiamo.

00401145 CALL [EAX] call a GlobalAlloc

con questo GlobalAlloc viene allocata un'area di memoria all'address 0051000C. Vediamo perchè:

00401162 REPZ MOVSD
...
0040116F IRETD

con ESI = 004011AF, EDI = 0051000C ed ECX = 0F82.
Viene spostato del codice dalla riga 004011AF alla nuova area di memoria appena allocata a 0051000C. Poi esce dall'handler dell'int 5, quindi tornate a 00401032

0040105A ripristino int 5

viene ripristinato l'int 5 con le 2 solite righe, visto che prima l'avevamo fatto cambiare ora facciamoglielo rimettere a posto.

00401193 PUSH EAX

questa riga e le due seguenti istallano un nuovo SEH: con EAX = 0051000C viene istallato un seh sul codice allocato poco fa, che in realtà era solo un alias di quello presente alla riga 004011AF. Questo codice era stato xorato prima (vi dicevo di tenerlo a mente!) con il valore del checksum, quindi se aveste fatto cazzate il checksum sarebbe stato errato e il codice nell'handler della eccezione sarebbe risultato non valido.

00401069 INVALID

come invalid?? Non ho riportato la riga, ma da qualche parte c'era un
MOV [00401069], FFFF
ed FFFF è l'opcode di un invalid opcode :-) Cosa può essere questa riga se non una chiamata diretta al seh???
Stavolta il break sull'entry point del seh non funziona, però visto che sappiamo che andremo a finire lì cambiamoci direttamente l'eip a mano prima di eseguire l'FFFF. Il seh stava a 0051009C.

```
0051018D    MOV  EAX, 0040214C    prendi iat
          ADD  EAX, 04            e prendi l'address di GetSystemTime
          MOV  ECX, 14
          CMP  byte ptr [EAX], CC       cerca un breakpoint
          jz   00510209
```

qui viene preso l'address di GetSystemTime e vengono scannati 14h bytes alla ricerca del famelico int 3 che indica la presenza di un breakpoint. Ci stiamo avvicinando al punto che ci interessa, visto che viene controllata la funzione GetSystemTime.

```
005101B2    CALL  [EAX]     call GetSystemTime
          MOV   EAX, [00402030]  metti in eax il time ottenuto
          AND   EAX, FFFF       tienine solo l'anno (2001)
          XOR   ECX, ECX        azzera ecx
          MOV   EBX, 0740       ebx = 1856
          CMP   EAX, EBX        eccolo!
          JLE             se abbiamo superato il 1856 il trial è scaduto :-)
```

ecco il cuore del controllo! Come vedete se l'anno attuale è superiore al 1856 allora il trial è scaduto. Cambiate il flag sul JLE, lasciate eseguire il programma, et voilà! Niente errori, siamo nella finetra principale! Non sono in molti ad averla vista!
Ma ora viene la parte più brutta! Abbiamo effettuato le modifiche in memoria, ora dobbiamo patchare l'eseguibile per renderle definitive. Come già immaginiamo Yado non ci permetterà di patchare il codice tanto facilmente, ma basterà un pò di attenzione e tutto filerà liscio :-). Innanzitutto visto che il file è criptato dobbiamo sapere DOVE andare a patchare e COME. La routine di decrypt del codice l'abbiamo vista (0040707D), anche il secondo layer di decrypt l'abbiamo incontrato (004016C9), quindi sappiamo come ma non dove. La linea da patchare si trova a 005101B2, ma questa era una parte di memoria ottenuta con il GlobalAlloc e nella quale è stato copiato il codice che sta a 004011AF, quindi è lì che dobbiamo patchare il jle. Andiamo alla riga 004011AF e cerchiamo il cmp che determina la fine del time limit, e troviamo il relativo jump alla linea

0040136A JLE xxxxx

ora con un FLC qualsiasi calcoliamoci l'offset: 96A. Bene. Ora sappiamo anche DOVE, quindi non ci resta che modificare i bytes fisici affinchè quando vengano decriptati diano il risultato che vogliamo noi. Prendiamo una manciata di bytes a partire da

00401368 BB 17 86 75 06 E2 46 B2

ora vediamo le trasformazioni dei due layer cosa fanno:

```
BB 17 86 75 06 E2 46 B2
----------------------- primo layer: xor 01012001 - ror 2 - xor 01012001
EE ED 20 DC 81 D0 D0 AD
----------------------- secondo layer: xor con 2F2ED552 (checksum)
      0F 8E 54 FE FF FF  -> opcode del JLE
        (52 D5 2E 2F)    -> (xormask)
```

bene, solo che al posto del JLE (0F8E) vogliamo un bel JG (0F8F). Quindi ora rifacciamoci i calcoli al contrario. Le dword sono colorate alternatamente per ricordarvi di rispettare l'allineamento dei calcoli. Ricordate anche di non fare casini con la notazione intel (bytes invertiti). Vediamo ora il calcolo al contnrario:

al posti di 8E mettiamo 8F, quindi -> 8F xor52 = DD
EE ED 20 DD xor 01012001 / rol 2 /xor 01012001 = BB 17 86 71

ecco quindi che all'offset fisico dobbiamo sostituire il 75 con un bel 71. Notate il ROL, invertito rispetto a ROR perchè abbiamo effettuato il calcolo a ritroso. Ma questo ancora non basta! Lanciamo il K2 senza softice caricato e ci dà errore. Evidentemente deve esserci qualche altro controllo. Se infatti torniamo a steppare dal JLE che abbiamo appena modificato ecco i passi che troviamo:

005100EB CLI
...
00510101 MOV [EBX], AX
...
00410107 MOV [EBX+6], AX
...
0051010B INT 5

okei il solito INT 5 chiamato, con l'handler rediretto a 004012CE.

00401307 XOR [EAX], E3D42DF6

questo è un cicletto che decripta la sezione delle risorse, quindi se volete cambiare il look del crackme tenete a mente queste righe.

```
005102CD   MOV  EAX, 0040133A
           MOV  EBX, 0040146F
           XOR  ECX, ECX
           ADD  ECX, [EAX]
           INC  EAX
           CMP  EAX, EBX
           JLE  005102D9
           CMP  ECX, 45536EFF
           JNZ  00510BDE
```

ecco qualcosa di interessante! Un checksum che comprende la riga 0040136A che noi abbiamo patchato. Il checksum risulterà alterato visto che abbiamo modificato unn byte. A runtime possiamo modificare al volo il flag z per evitare di saltare alla riga 00510BDE, ma se lasciamo correre il programma avremo lo stesso un errore, probabilmente perchè c'è qualche altro meccanismo di controllo. Se andiamo avanti troviamo le seguenti righe degne di nota:

0051030C di nuovo modifica l'int 5, ora l'handler è 00401502
...
00510316 INT 5 niente di particolarmente importante
...
00510350 PUSH EAX con eax = 5102B3, istallazione di un nuovo seh
...
00510380 altro ciclo di cmpare con "TNIP" che ci manda dritti nel seh (proprio come quello di prima)
...

```
00510032   PUSH  00                    inizialization value
           PUSH  00401215              dialog func
```

```
PUSH  00                        hwnnd parent
PUSH  67                        template name
PUSH  dword ptr [004020BC]      hinstance
CALL  DialogBoxParamA
```

e qui succedono i casini. Infatti la call a DialogBoxParamA ci visualizza il messaggio di errore se abbiamo patchato il byte, se invece abbiamo trickato solo a runtnime allora ci visualizza la finestra vera e propria del crackme. In entrambi i casi i parametri passati alla API sono uguali. Notate il parametro Dialog Funz che punta alla rocedura del dialog box. Probabilmente è li che il codice fa un ulteriore controllo sull'integrità del programma. Il problema è che inte realtà alla riga 00401215 non c'è nulla di interessante!! Il tracing qui risulta estremamente complicato, ho provato a settare BPR su tutto il codice possibile del K2 per vedere di beccare qualche routine di checksum, ma niente da fare... allorchè ho pensato: chi se e frega! Se infatti i controlli di integrità si basano su checksum allora siamo a cavallo, in quanto il checksum è debolissimo. Analizziamo meglio la situazione. Modificando il byte da 8E a 8F abbiamo incremetato di 1, e il checksum non dà più lo stesso valore. Essendo il checksum appunto basato sulla semplice somma possiamo modificare il byte successivo al nostro appena creato JG: tale byte non sarà un problema perchè non viene MAI eseguito se c'è il JG, quindi se gli sottraiamo 1 (da B8 a B7) il delta creatosi col precedente patching verrà riassorbito e alla fine il checksum risulterà corretto :-). Insomma, facciamo una cosa di questo genere:

da x + y + z a x + (y+1) + (z-1)

così il totale è sempre lo stesso. Quindi i bytes decriptati a runtime sono:

```
0F 8F 54 FE FF FF B7 4C 21 40 00 B3       con B8 cambiato in  B7, quindi
applichiamo la xormask (layer 2°)
      D5 2E 2F 52 D5 2E 2F 52
-----------------------------------
            D0 AD 62 62 0E 12          ora applichiamo lo xor / rol / xor  (layer 1°)
-----------------------------------
            46 B2 88 29 3C 4D
```

ecco fatto! Abbiamo ottenuto un bell' 88 al posto di B4. Fate anche questa modifica, e ora il krypton2 partirà senza problemi. Ricordatevi che non dovete avere softice caricato!!!

Byez
AndreaGeddon

Note finali

In primis un saluto a Yado che ha fatto il crackme, poi un saluto a ZeroByte che è davvero troppo forte! Grazie per il logo Zero! Un saluto a Acid_Leo che mi intrattiene sempre con argomenti interessanti, poi un saluto a tutti quelli che stavano da Zero, tranne al pervertito di Risk che ha dormito con me e voleva violentarmi. Un saluto ad Athena che cucina benissimo e che ci ha sopportati per 2 giorni :). Infine un non-saluto al Que che non solo ci ha solati da Zero, ma ci ha solati pure la sera prima (vediamo se stasera ci sola di nuovo...)
ADDENDUM: come prevedibile la sola è avvenuta!

Disclaimer

consumatori i migliori prodotti possibili.

Noi reversiamo al solo scopo informativo e di miglioramento del linguaggio Assembly.

Capitooooooooo????? Bhè credo di si ;))))