TorresCrack

Visual protect 3.5.4 en vb-pcode

Twitter: @TorresCrack248
11/09/2013

INTRODUCCION

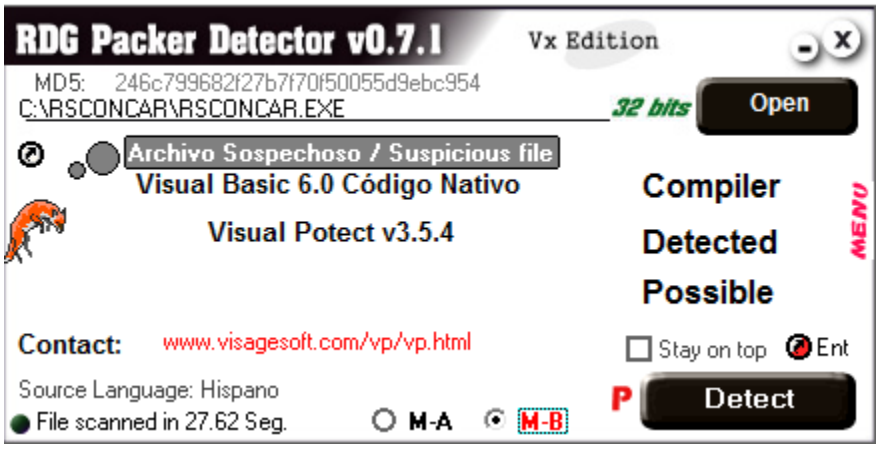
En esta ocasión escribiré algo corto y muy sencillo para no dejar tanto tiempo sin escribir. Veremos una "mejora" a la protección de un software del cual escribí hace un tiempo, únicamente han agregado el packer "Visual Protect" el cual es algo viejo, basándome en la información que encontré de ello:

"Visual Protect is a software based protection, e-commerce, and license management tools. With Visual Protect you can protect any Windows 95/98/NT 32-bit executable from piracy, illegal distribution, and hacking. Protection of your application with Visual Protect requires no source code editing. It allows potential customers to try your complete product before buying, using a simple, smart and user-friendly interface. In addition, thru Visual Protect you can distribute license files to your registered customers as it incorporates a mechanism to manually or automatically generate and send by E-Mail the required registration files."

En un visual basic p-code + dongle, el cual después de todo se logra fácil el desempaquetado que es el punto principal del escrito, decidí escribir únicamente acerca del packer y no del método para registrar la aplicación por algunos problemas con el dueño del software y la lista *"CracksLatinos"* de los cuales me menciono Ricardo. Así que el escrito se acorto mucho pero igual sirve de ejemplo en el unpacking de este sencillo packer.

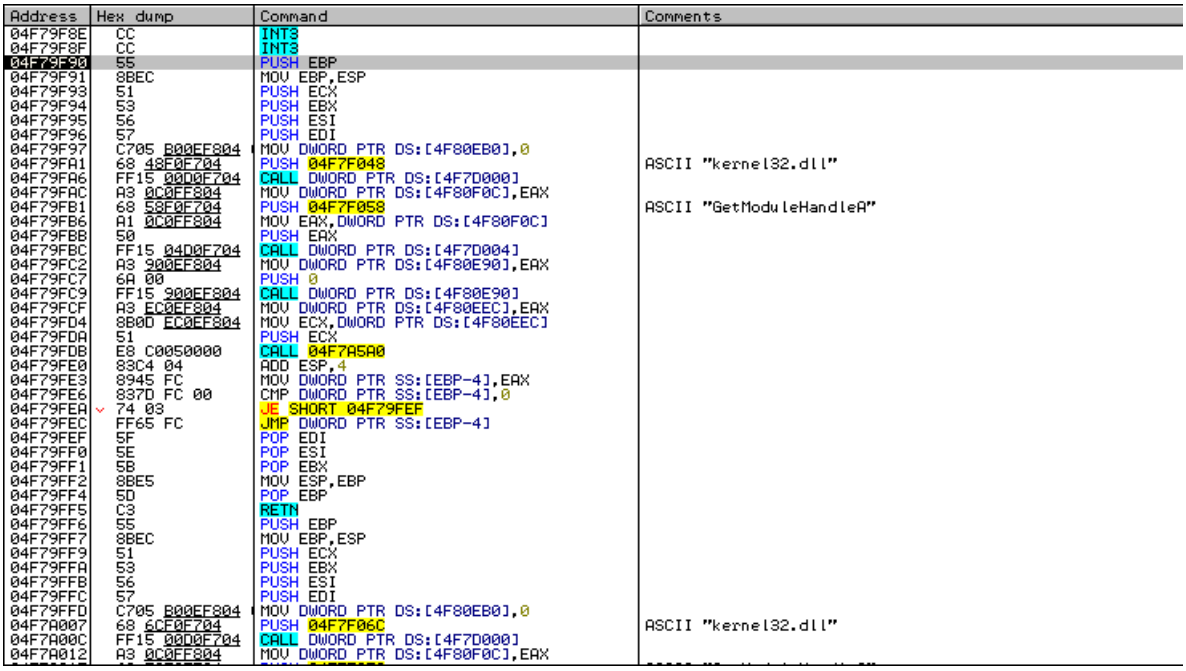
Comenzando con el objetivo:

Antes de iniciar vamos a mostrar lo que nos devuelve el analizador "RDG packer detector" y no está de más agradecer a RDGMAX por su nueva versión.



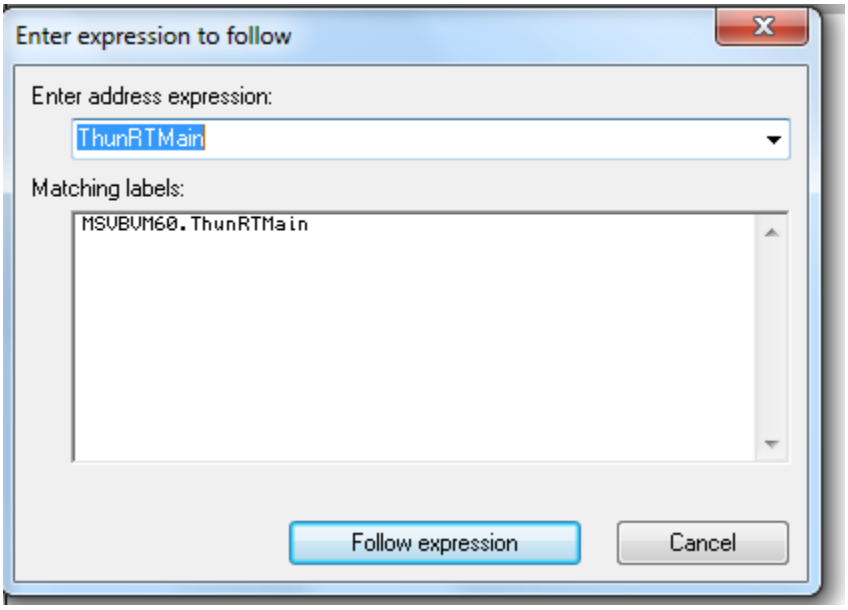
Como vemos, detecto el "visual protect" y un visual basic nativo, lo cual es incorrecto pues se trata de un vb p-code, seguimos

Abrimos nuestro olly debugger y podemos ver el *"Entry Point"*

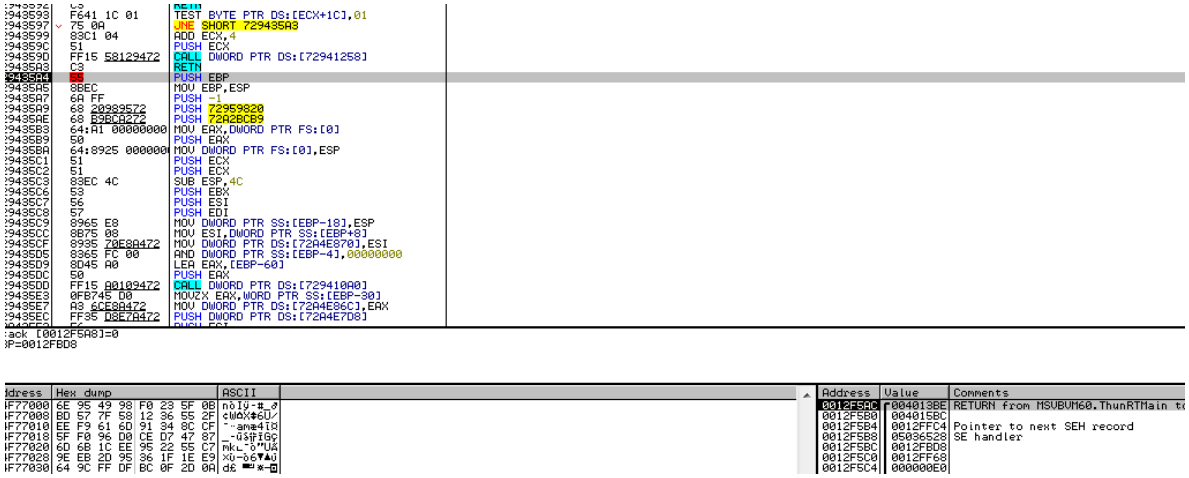


Address	Hex dump	Command	Comments
04F79F8E	CC	INT3	
04F79F8F	CC	INT3	
04F79F90	55	PUSH EBP	
04F79F91	8BEC	MOV EBP,ESP	
04F79F93	51	PUSH ECX	
04F79F94	53	PUSH EBX	
04F79F95	56	PUSH ESI	
04F79F96	57	PUSH EDI	
04F79F97	C705 B00EF804	MOV DWORD PTR DS:[4F80EB0],0	
04F79FA1	68 48F0F704	PUSH 04F7F048	ASCII "kernel32.dll"
04F79FA6	FF15 00D0F704	CALL DWORD PTR DS:[4F7D000]	
04F79FAC	A3 0C0FF804	MOV DWORD PTR DS:[4F80F0C],EAX	
04F79FB1	68 58F0F704	PUSH 04F7F058	ASCII "GetModuleHandleA"
04F79FB6	A1 0C0FF804	MOV EAX,DWORD PTR DS:[4F80F0C]	
04F79FBB	50	PUSH EAX	
04F79FBC	FF15 04D0F704	CALL DWORD PTR DS:[4F7D004]	
04F79FC2	A3 900EF804	MOV DWORD PTR DS:[4F80E90],EAX	
04F79FC7	6A 00	PUSH 0	
04F79FC9	FF15 900EF804	CALL DWORD PTR DS:[4F80E90]	
04F79FCF	A3 EC0EF804	MOV DWORD PTR DS:[4F80EEC],EAX	
04F79FD4	8B0D EC0EF804	MOV ECX,DWORD PTR DS:[4F80EEC]	
04F79FDA	51	PUSH ECX	
04F79FDB	E8 C0050000	CALL 04F7A5A0	
04F79FE0	83C4 04	ADD ESP,4	
04F79FE3	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
04F79FE6	837D FC 00	CMP DWORD PTR SS:[EBP-4],0	
04F79FEA	74 03	JE SHORT 04F79FEF	
04F79FEC	FF65 FC	JMP DWORD PTR SS:[EBP-4]	
04F79FEF	5F	POP EDI	
04F79FF0	5E	POP ESI	
04F79FF1	5B	POP EBX	
04F79FF2	8BE5	MOV ESP,EBP	
04F79FF4	5D	POP EBP	
04F79FF5	C3	RETN	
04F79FF6	55	PUSH EBP	
04F79FF7	8BEC	MOV EBP,ESP	
04F79FF9	51	PUSH ECX	
04F79FFA	53	PUSH EBX	
04F79FFB	56	PUSH ESI	
04F79FFC	57	PUSH EDI	
04F79FFD	C705 B00EF804	MOV DWORD PTR DS:[4F80EB0],0	
04F7A007	68 6CF0F704	PUSH 04F7F06C	ASCII "kernel32.dll"
04F7A00C	FF15 00D0F704	CALL DWORD PTR DS:[4F7D000]	
04F7A012	A3 0C0FF804	MOV DWORD PTR DS:[4F80F0C],EAX	

Aquí usaremos un método muy simple para encontrar el Original Entry Point, y es que cuando se trata de un Visual Basic Nativo o p-code sabemos que ambos usan la API *"ThunRTMain"* al iniciar la aplicación cerca del Original Entry Point, así que sin dar más vueltas ejecutamos dentro de nuestro debugger la aplicación para que este pueda cargar la librería y poder buscar la dirección:



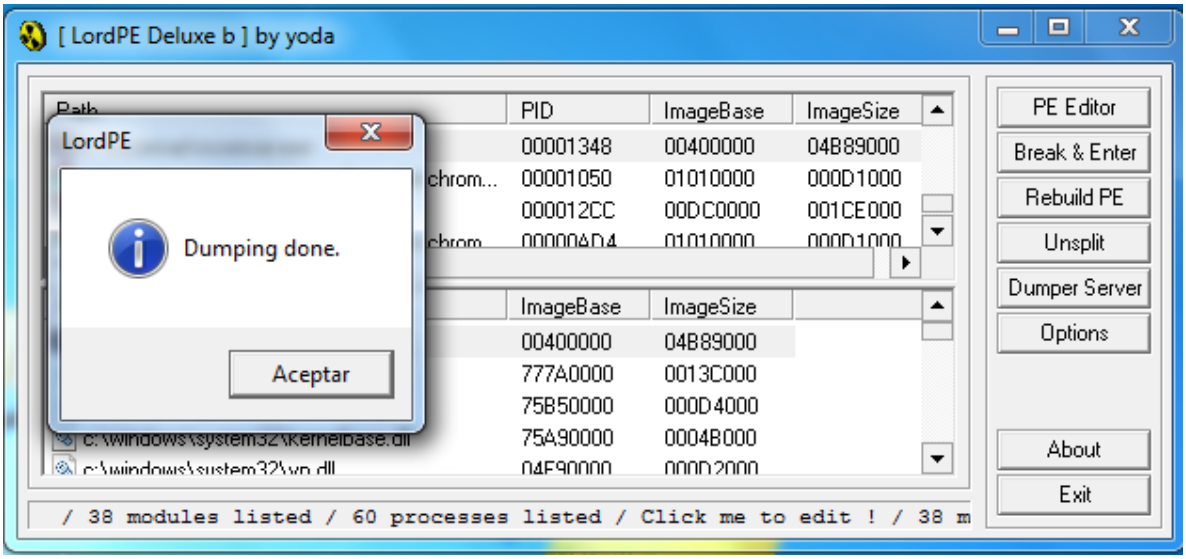
colocamos un Hardware BreakPoint on execution en *"ThunRTMain"* y reiniciamos nuestro debugger y corremos nuestro binario hasta que para su ejecución en el breakpoint, si vemos el stack y nos muestra la dirección de retorno y si la seguimos:



Nos encontramos debajo del Original Entry Point “push 004015bc”, pues es el típico Entry Point de los vb nativo y p-code, podemos poner un Hardware BreakPoint en el OEP:

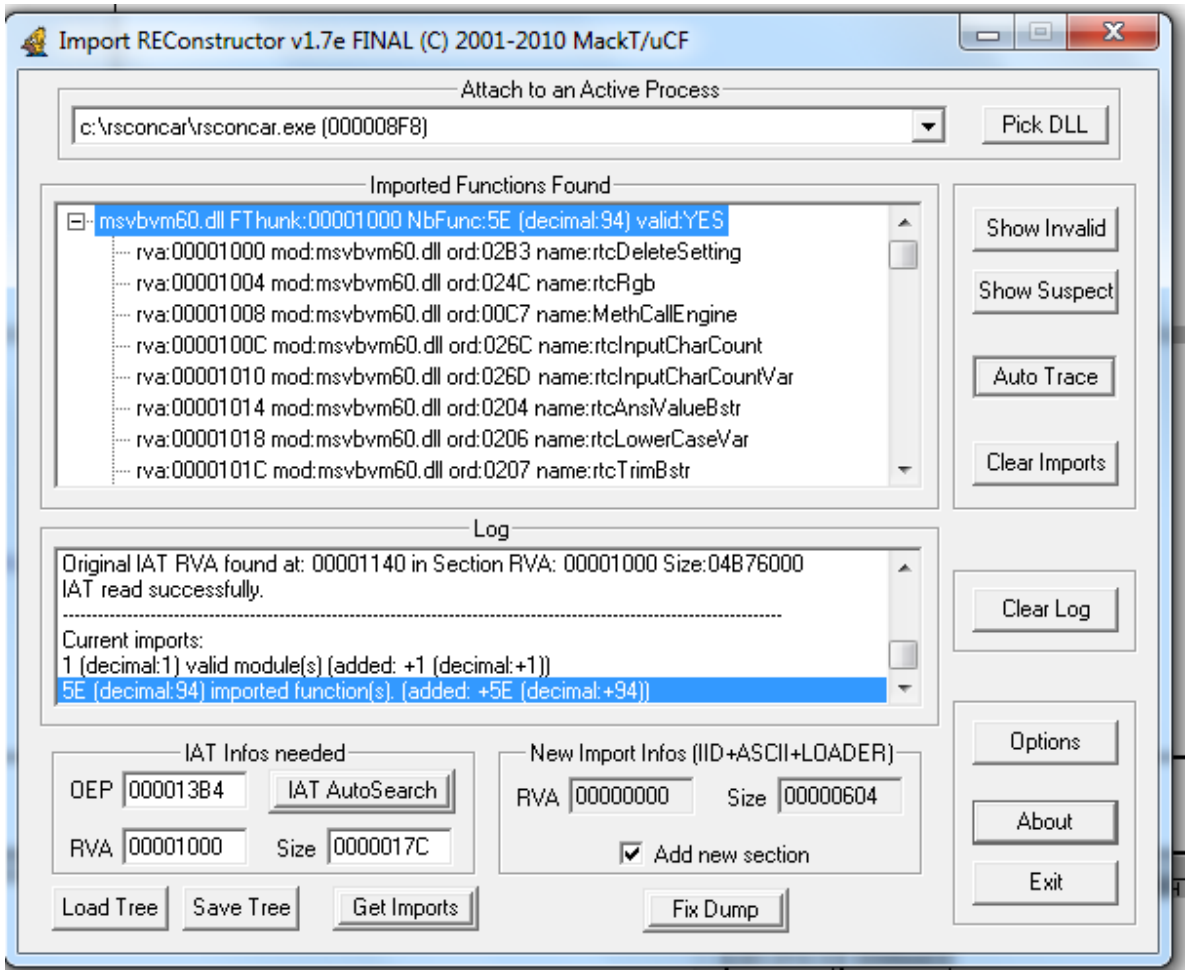
```
013384 - FF25 4C104000 JMP DWORD PTR DS:[40104C]
01338A - FF25 74104000 JMP DWORD PTR DS:[401074]
013390 - FF25 D8104000 JMP DWORD PTR DS:[4010D8]
013396 - FF25 E4104000 JMP DWORD PTR DS:[4010E4]
01339C - FF25 F0104000 JMP DWORD PTR DS:[4010F0]
0133A2 - FF25 F8104000 JMP DWORD PTR DS:[4010F8]
0133A8 - FF25 04114000 JMP DWORD PTR DS:[401104]
0133AE - FF25 40114000 JMP DWORD PTR DS:[401140]
0133B4   68 BC154000   PUSH 004015BC
0133B9   E8 F0FFFFFF   CALL 004013AE                    Jump to MSVBVM60.ThunRTMain
0133BE   0000          ADD BYTE PTR DS:[EAX],AL
0133C0   48            DEC EAX
0133C1   0000          ADD BYTE PTR DS:[EAX],AL
0133C3   0030          ADD BYTE PTR DS:[EAX],DH
0133C5   0000          ADD BYTE PTR DS:[EAX],AL
0133C7   0040 00       ADD BYTE PTR DS:[EAX],AL
0133CA   0000          ADD BYTE PTR DS:[EAX],AL
```

Con esa información, podemos continuar con el dumpeado y la reparación de la IAT, para eso podemos usar “LordPe”:



(dumping full)

Ahora procedemos a corregir el entry-point y si lo requiere, reparar la IAT, usaremos ImportReconstructor, metemos lo datos (OEP,RVA,Size):



Como mencione anteriormente, es un packer sencillo y espero sirva de ayuda para algunos, en el próximo escrito (espero sea pronto) espero tener listo un escrito que no he podido terminar por falta de tiempo.

AGRADECIMIENTOS:

Debo agradecer a apuromafo, mcksys, por animarme a escribir este tutorial y a ti por haber llegado hasta aquí.

Saludos

TorresCrack