

Pepsi 2.0 Unpacking



Luca D'Amico

<https://www.lucadamico.dev>

19-Aug-2023

Sommario

Abstract.....	3
Ambiente E Strumenti Usati.....	4
Analisi Iniziale	5
OSINT.....	5
Alto Tasso Di Detection Su Virus Total	7
Packer Detection.....	7
Unpacking	9
Come Funziona Pepsi 2.0?	9
Dumping: Metodo A - Fix Manuale	16
Dumping: Metodo B - Automatico	18
Dumping: Metodo C - Estrazione File Unpackato Dalla Memoria Temporanea	19
Conclusione	23
Ringraziamenti	23

Abstract

Questo documento è dedicato all'analisi e all'unpacking di Pepsi 2.0: un packer proveniente dalla scena underground di cui sono disponibili pochissime informazioni pubbliche.

Come descritto in seguito, questo packer ha alcune limitazioni e presenta vari problemi, ma utilizza alcune tecniche interessanti.

Il binario in esame è stato estratto dalla collezione di unpackme proveniente dal forum di tuts4you (<https://forum.tuts4you.com/files/file/1314-tuts-4-you-unpackme-collection-2016>), è disponibile al download gratuitamente e legalmente.

Ambiente E Strumenti Usati

L'analisi di Pepsi è stata effettuata su una macchina virtuale con Windows XP SP3. I seguenti strumenti sono stati impiegati:

- PEiD, DIE & CFF Explorer: ottenimento informazioni sul binario
- x32dbg & Scylla: debugging e unpacking
- HxD: modifiche al dump per fixare l'header nel primo metodo proposto
- Lord PE: per fixare il dump usando il terzo metodo proposto

L'unpackme ha il seguente SHA256:

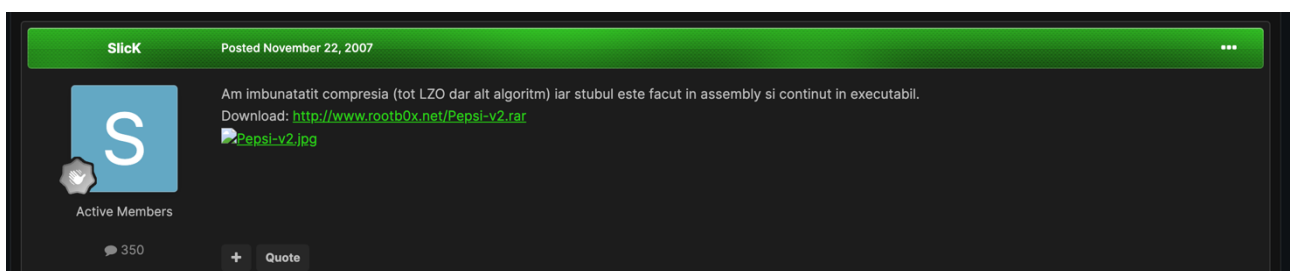
79288c042048afd61d3ddec9a75b8bedf1830adfc015c873d676ff4782d2a339

Analisi Iniziale

In questa sezione verranno descritte varie analisi effettuate sul binario al fine di ottenere informazioni utili.

OSINT

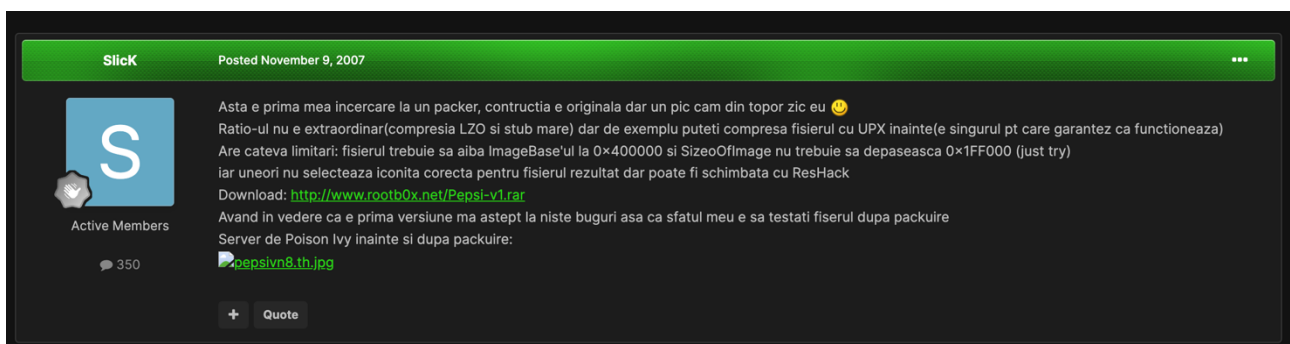
Su questo packer non si hanno molte informazioni. Facendo una ricerca mirata su Google usando come keywords “Pepsi” “Slick” e “packer”, l’unico riferimento significativo proviene da un forum di Cyber Security in lingua rumena inviato proprio dal creatore di Pepsi e risalente al 22 novembre 2007, con titolo “Pepsi Packer v2”:



Traducendo il post con Google Translate otteniamo:

“Ho migliorato la compressione (anche LZO ma un altro algoritmo) e lo stub è realizzato in assembly e contenuto nell'eseguibile.”

Il link fornito alla fine del post purtroppo non è più raggiungibile, ma effettuando una nuova ricerca su Google limitando i risultati al forum in esame (usando la keyword site:<https://rstforums.com/>), possiamo trovare anche il relativo post che pubblicizza la prima versione del packer, intitolato “Pepsi Packer v1”:



Sempre ricorrendo a Google Translate, la traduzione è la seguente:

“Questo è il mio primo tentativo con un packer, la costruzione è originale, ma un po' fuori dal comune, direi :)”

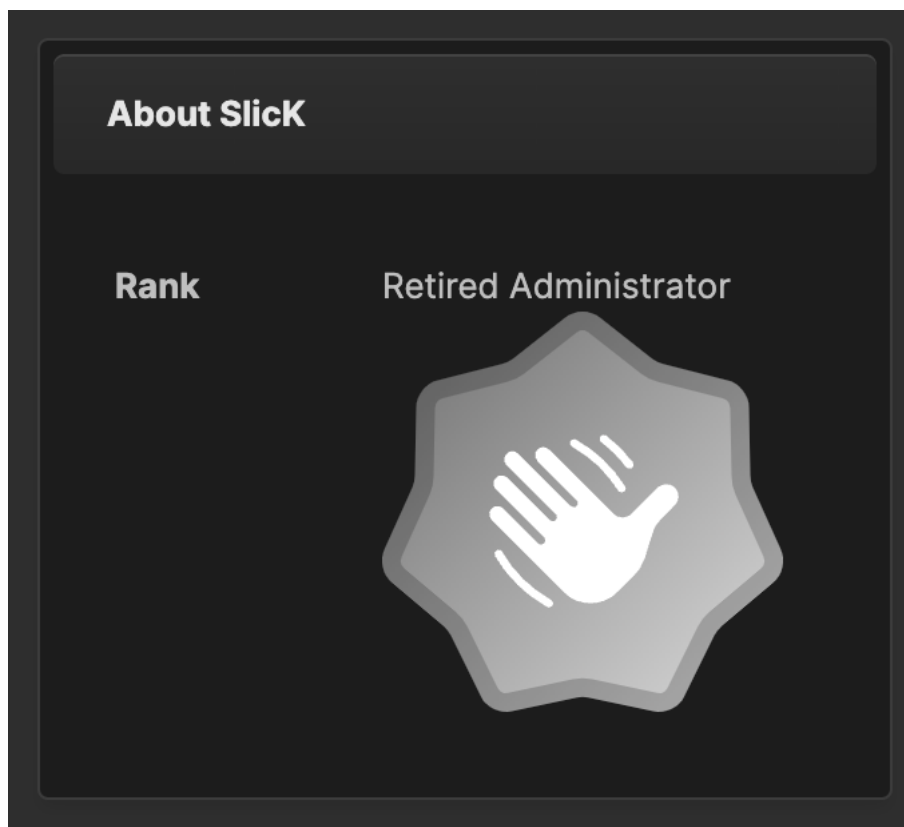
*Il rapporto non è straordinario (compressione LZO e big stub) ma ad esempio puoi comprimere il file con UPX prima (è l'unico che ti garantisco che funziona)
Ha alcune limitazioni: il file deve avere ImageBase a 0x400000 e SizeOfImage non deve superare 0x1FF000 (basta provare)
e talvolta non seleziona l'icona corretta per il file risultante, ma può essere modificata con ResHack*

Scarica: ...

Considerando che è la prima versione, mi aspetto qualche bug, quindi il mio consiglio è di testare il file dopo averlo impacchettato”

Questo documento prende in esame la versione V2 di Pepsi, ma ho ritenuto comunque interessante aggiungere queste informazioni sulla V1 in quanto tali problemi risultano ancora presenti.

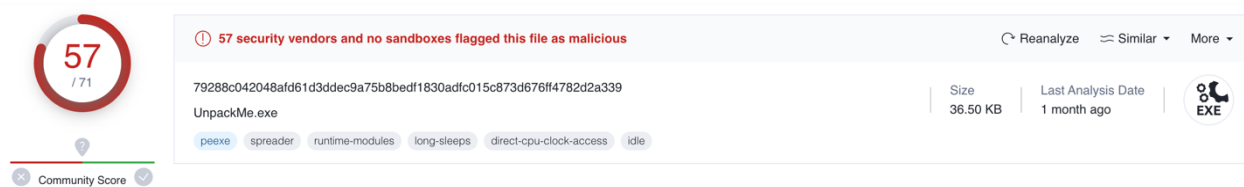
Un'ultima curiosità su SlicK, l'autore di Pepsi, la possiamo ottenere guardando il suo profilo sul forum:



A quanto pare è l'ex amministratore di questo forum e ha effettuato l'accesso l'ultima volta il 12 novembre 2011. Spero che stia bene e che un giorno legga questo paper, magari la versione inglese 😊.

Alto Tasso Di Detection Su Virus Total

Questo packer ha un elevatissimo tasso di detection su Virus Total:



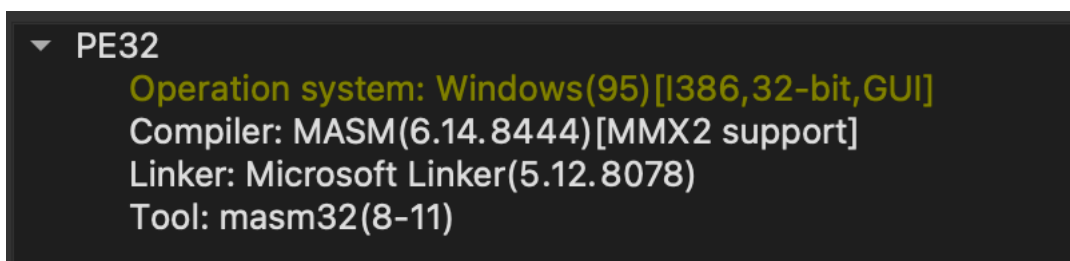
57 software antivirus lo rilevano come malware. VT ha assegnato il seguente nome alla possibile minaccia: trojan.barys/backdoorx.

Benché sia possibile che le tecniche usate dal packer possano causare falsi positivi, è anche probabile che Pepsi sia stato impiegato (non dal suo autore originale) nel corso del tempo, nel tentativo di nascondere qualche malware.

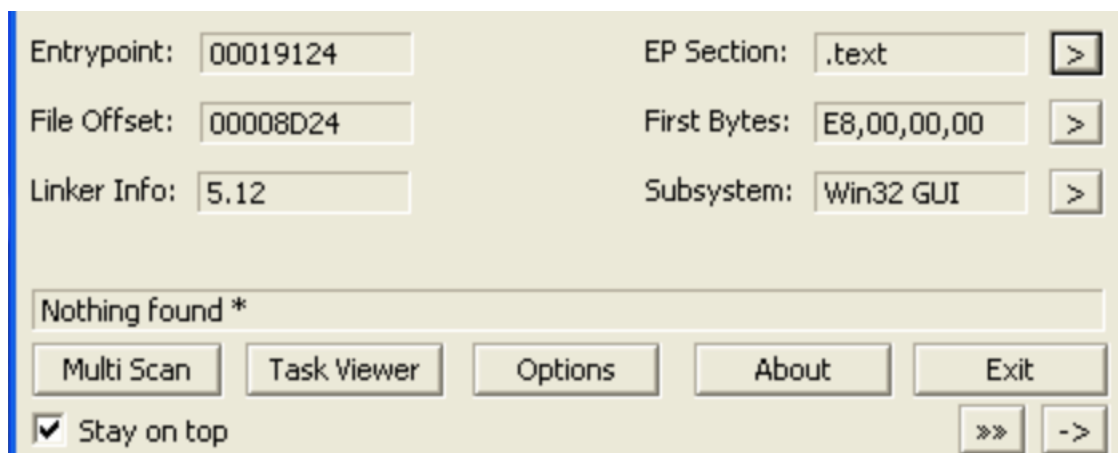
Packer Detection

Pepsi è completamente invisibile ai più noti Packer Detectors.

Detect It Easy rileva solo che il binario è stato scritto con l'ausilio di MASM:



PEiD non rileva nulla:



Un metodo rapido per rilevare la presenza di Pepsi è verificare che tra le sezioni del binario ne sia presente una chiamata “.pepsi”. In questo screenshot tale verifica è stata effettuata usando CFF Explorer:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.pepsi	00018000	00001000	00008891	00000200
.text	00001000	00019000	00000600	00008C00

Inoltre, sembra che Pepsi non sia in grado di mantenere l'icona originale dei programmi che comprime. I binari finali risultano privi di icona, sino a quanto non vengono unpackati.

Unpacking

In questa sezione attaccheremo il binario, debuggando e studiando il comportamento di Pepsi in fase di esecuzione ed infine estrarremo il binario originale unpackato.

Come Funziona Pepsi 2.0?

Carichiamo pepsi.exe in x32dbg e raggiungiamo l'EntryPoint:

```
00419124 E8 00000000 call pepsi.419129
00419129 6A 00 push 0
0041912B E8 28040000 call <JMP.&GetModuleHandleA>
00419130 A3 04914100 mov dword ptr ds:[419104],eax
00419135 8B3D 04914100 mov edi,dword ptr ds:[419104]
0041913B 037F 3C add edi,dword ptr ds:[edi+3C]
0041913E 81C7 F8000000 add edi,F8
00419144 8B57 08 mov edx,dword ptr ds:[edi+8]
00419147 8915 0C914100 mov dword ptr ds:[41910C],edx
0041914D 8B57 0C mov edx,dword ptr ds:[edi+C]
00419150 0315 04914100 add edx,dword ptr ds:[419104]
00419156 8915 10914100 mov dword ptr ds:[419110],edx
0041915C 8B57 0C mov edx,dword ptr ds:[edi+C]
0041915F 8915 08914100 mov dword ptr ds:[419108],edx
00419165 8B57 10 mov edx,dword ptr ds:[edi+10]
00419168 8915 14914100 mov dword ptr ds:[419114],edx
0041916E 6A 04 push 4
00419170 68 00100000 push 1000
00419175 FF35 0C914100 push dword ptr ds:[41910C]
0041917B 6A 00 push 0
0041917D E8 EE030000 call <JMP.&VirtualAlloc>
```

Come è possibile notare, viene ottenuto l'handle del modulo attuale passando 0 (NULL) all'API GetModuleHandleA. Il registro eax conterrà adesso il valore 0x400000 (dove risiede l'eseguibile in memoria). Vengono quindi recuperate e salvate varie informazioni dall'header. Per semplificare la comprensione di questa parte, ho assegnato dei nomi significativi ai vari indirizzi di memoria e commentato il disassembly:

```
00419124 E8 00000000 call pepsi.419129 call $0
00419129 6A 00 push 0
0041912B E8 28040000 call <JMP.&GetModuleHandleA>
00419130 A3 04914100 mov dword ptr ds:[<imagebase>],eax
00419135 8B3D 04914100 mov edi,dword ptr ds:[<imagebase>]
0041913B 037F 3C add edi,dword ptr ds:[edi+3C]
0041913E 81C7 F8000000 add edi,F8
00419144 8B57 08 mov edx,dword ptr ds:[edi+8]
00419147 8915 0C914100 mov dword ptr ds:[<virtualsize_pepsi>],edx
0041914D 8B57 0C mov edx,dword ptr ds:[edi+C]
00419150 0315 04914100 add edx,dword ptr ds:[<imagebase>]
00419156 8915 10914100 mov dword ptr ds:[<virtualaddr_pepsi>],edx
0041915C 8B57 0C mov edx,dword ptr ds:[edi+C]
0041915F 8915 08914100 mov dword ptr ds:[<rvva_pepsi>],edx
00419165 8B57 10 mov edx,dword ptr ds:[edi+10]
00419168 8915 14914100 mov dword ptr ds:[<rawsize_pepsi>],edx
0041916E 6A 04 push 4
00419170 68 00100000 push 1000
00419175 FF35 0C914100 push dword ptr ds:[<virtualsize_pepsi>]
0041917B 6A 00 push 0
0041917D E8 EE030000 call <JMP.&VirtualAlloc>
```

[edi+3c] = address of PE header
start of first section header (.pepsi)
[edi+8] = virtual size of .pepsi

[edi+c] = RVA of .pepsi
RVA + imagebase = VA of .pepsi

[edi+c] = RVA of .pepsi
[edi+10] = SizeOfRawData

Notiamo a questo punto una chiamata a VirtualAlloc, all'indirizzo 0x41917D, che allocherà una porzione di memoria grande quando la virtual size del segmento .pepsi, ovvero in questo caso 0x18000.

Continuiamo analizzando la porzione di disassembly seguente:

```

0041917D E8 EE030000 call <JMP.&VirtualAlloc>
00419182 0BC0 or eax,eax
00419184 75 07 jne pepsi.41918D
00419186 6A 00 push 0
00419188 E8 C5030000 call <JMP.&ExitProcess>
0041918D A3 18914100 mov dword ptr ds:[419118],eax
00419192 6A 00 push 0
00419194 68 1C914100 push pepsi.41911C
00419199 FF35 18914100 push dword ptr ds:[419118]
0041919F FF35 14914100 push dword ptr ds:[419114]
004191A5 FF35 10914100 push dword ptr ds:[419110]
004191AB E8 C1010000 call pepsi.419371

```

Se l'allocazione di memoria non dovesse avvenire con successo, il programma terminerà chiamando ExitProcess. Altrimenti, verrà salvato l'indirizzo di inizio della memoria appena allocata (contenuto ovviamente nel registro eax) in 0x419118. Notiamo inoltre come questo dato venga pushato sullo stack ed usato come parametro nella chiamata all'indirizzo 0x4191AB.

Prima di continuare, diamo dei nomi significativi agli indirizzi per facilitarci la comprensione del disassembly:

```

0041917D E8 EE030000 call <JMP.&VirtualAlloc>
00419182 0BC0 or eax,eax
00419184 75 07 jne pepsi.41918D
00419186 6A 00 push 0
00419188 E8 C5030000 call <JMP.&ExitProcess>
0041918D A3 18914100 mov dword ptr ds:[<reserved_space_for_unpacking>],eax
00419192 6A 00 push 0
00419194 68 1C914100 push pepsi.41911C
00419199 FF35 18914100 push dword ptr ds:[<reserved_space_for_unpacking>]
0041919F FF35 14914100 push dword ptr ds:[<rawsize_pepsi>]
004191A5 FF35 10914100 push dword ptr ds:[<virtualaddr_pepsi>]
004191AB E8 C1010000 call pepsi.419371

```

Già guardando i parametri passati a questa call possiamo immaginare cosa accadrà. Infatti, eseguiamo la funzione senza entrarci dentro e poi guardiamo nel memory dump la porzione di memoria riservata in precedenza con la VirtualAlloc (che parte all'indirizzo salvato in [0x419118], in questo caso 0x330000, che a sua volta abbiamo rinominato in "reserved_space_for_unpacking"):

```

00330000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....yy..
00330010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00330020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00330030 00 00 00 00 00 00 00 00 00 00 00 00 B8 00 00 00 .....
00330040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..°...!!,Li!Th
00330050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00330060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00330070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.
00330080 FA B9 CD 45 BE D8 A3 16 BE D8 A3 16 BE D8 A3 16 ú'IE%of.%of.%of.
00330090 30 C7 B0 16 86 D8 A3 16 42 F8 B1 16 BC D8 A3 16 OÇ°..of.B±.%of.
003300A0 79 DE A5 16 BF D8 A3 16 52 69 63 68 BE D8 A3 16 yD%.of.Rich%of.

```

La memoria è stata riempita con un eseguibile! Possiamo già ipotizzare che questo sia il binario unpackato, ma per esserne sicuri dobbiamo continuare la nostra analisi.

Proseguiamo:

004191AB	E8 C1010000	call pepsi.419371
004191B0	0BC0	or eax,eax
004191B2	74 0C	je pepsi.4191C0
004191B4	83F8 F8	cmp eax,FFFFFFF8
004191B7	74 07	je pepsi.4191C0
004191B9	6A 00	push 0
004191BB	E8 92030000	call <JMP.&ExitProcess>
004191C0	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
004191C6	E8 C5000000	call pepsi.419290

Troviamo un altro controllo che in caso di errore forza l'uscita del programma con la solita ExitProcess. Altrimenti se non sono presenti problemi, l'indirizzo di memoria dove riede il nuovo eseguibile viene pushato sullo stack per essere passato come parametro alla funzione chiamata a 0x4191C6.

Questa volta entriamo dentro questa call cliccando su Step Into.
Ci troveremo qui:

00419290	55	push ebp
00419291	8BEC	mov ebp,esp
00419293	83C4 FC	add esp,FFFFFFFC
00419296	8B7D 08	mov edi,dword ptr ss:[ebp+8]
00419299	037F 3C	add edi,dword ptr ds:[edi+3C]
0041929C	8B57 28	mov edx,dword ptr ds:[edi+28]
0041929F	0315 04914100	add edx,dword ptr ds:[419104]
004192A5	8915 20914100	mov dword ptr ds:[419120],edx
004192AB	8B97 80000000	mov edx,dword ptr ds:[edi+80]
004192B1	0355 08	add edx,dword ptr ss:[ebp+8]
004192B4	8BFA	mov edi,edx
004192B6	E9 80000000	jmp pepsi.41933B
004192BB	BB 00000000	mov ebx,0
004192C0	035D 08	add ebx,dword ptr ss:[ebp+8]
004192C3	035F 0C	add ebx,dword ptr ds:[edi+C]
004192C6	53	push ebx
004192C7	E8 8C020000	call <JMP.&GetModuleHandleA>
004192CC	0BC0	or eax,eax
004192CE	75 0C	jne pepsi.4192DC
004192D0	53	push ebx
004192D1	E8 8E020000	call <JMP.&LoadLibraryA>
004192D6	0BC0	or eax,eax
004192D8	75 02	jne pepsi.4192DC
004192DA	EB 72	jmp pepsi.41934E

Sappiamo, come abbiamo appena detto, che l'indirizzo dove risiede il nuovo eseguibile (probabilmente unpackato) è stato passato come parametro a questa funzione, quindi si trova in [ebp+8]. La mov situata a 0x419296 sposta tale indirizzo nel registro edi. Le istruzioni che seguono recuperano prima l'indirizzo dell'header del PE ([edi+3C]) e subito dopo l'RVA dell'entry point ([edi+28]). In [419104] è presente l'indirizzo dell'ibase di Pepsi (0x400000), e l'istruzione add (a 0x41929F) somma tale indirizzo a quello dell'RVA dell'entry point appena recuperato, in modo tale da ottenerne l'indirizzo assoluto. Tale indirizzo viene

salvato in [419120] e probabilmente usato in futuro per effettuare il magic jump passando il controllo all'eseguibile unpackato.

Non meravigliatevi se questo non ha attualmente molto senso poiché, come avrete notato, l'indirizzo assoluto dell'entry point appena calcolato si trova all'interno segmento .pepsi del packer. La spiegazione più logica è che quell'area di memoria verrà presto sovrascritta, probabilmente con i dati dell'eseguibile unpackato. Lo scopriremo tra poco.

Subito dopo, a 0x4192AB viene recuperato l'RVA della ImportDirectory ([edi+80]) dall'area di memoria dove risiede l'eseguibile unpackato e grazie all'add viene sommato alla sua imagebase per ottenerne l'indirizzo assoluto. Questo è necessario per costruire la IAT, infatti il codice che segue non fa altro che recuperare le librerie e le funzioni necessarie al programma unpackato per poter funzionare correttamente, usando GetModuleHandleA, LoadLibraryA e GetProcAddress.

Da notare bene: la IAT viene ricostruita all'interno della memoria dove risiede l'eseguibile unpackato!

Abbiamo scoperto cosa fa questa funzione: calcolare l'indirizzo assoluto dell'entry point e ricostruire la IAT.

Proseguendo l'analisi, vediamo che è presente anche in questo caso un controllo per assicurarsi che la funzione appena eseguita abbia completato correttamente il suo compito. Subito dopo, all'indirizzo 0x4191E3, viene effettuata un'altra call passando come parametri l'indirizzo della solita area di memoria dove risiede l'eseguibile unpackato (che adesso ha pure la IAT valida) e l'imagebase del packer, ovvero 0x400000 (contenuto in [419104]).

004191CB	83F8 01	cmp eax,1
004191CE	74 07	je pepsi.419107
004191D0	6A 00	push 0
004191D2	E8 7B030000	call <JMP.&ExitProcess>
004191D7	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
004191DD	FF35 04914100	push dword ptr ds:[419104]
004191E3	E8 63000000	call pepsi.41924B

Entriamo nella funzione per poterla analizzare:

0041924B	55	push ebp
0041924C	8BEC	mov ebp,esp
0041924E	83C4 FC	add esp,FFFFFFFC
00419251	8B75 08	mov esi,dword ptr ss:[ebp+8]
00419254	0376 3C	add esi,dword ptr ds:[esi+3C]
00419257	8B5D 0C	mov ebx,dword ptr ss:[ebp+C]
0041925A	035B 3C	add ebx,dword ptr ds:[ebx+3C]
0041925D	83BB 88000000 00	cmp dword ptr ds:[ebx+88],0
00419264	74 26	je pepsi.41928C
00419266	8D45 FC	lea eax,dword ptr ss:[ebp-4]
00419269	50	push eax
0041926A	6A 04	push 4
0041926C	6A 08	push 8
0041926E	56	push esi
0041926F	E8 08030000	call <JMP.&VirtualProtect>
00419274	8B93 8C000000	mov edx,dword ptr ds:[ebx+8C]
0041927A	8996 8C000000	mov dword ptr ds:[esi+8C],edx
00419280	8B93 88000000	mov edx,dword ptr ds:[ebx+88]
00419286	8996 88000000	mov dword ptr ds:[esi+88],edx
0041928C	C9	leave
0041928D	C2 0800	ret 8

Dai parametri passati alla funzione, sappiamo che l'area di memoria dove risiede l'eseguibile unpackato inizia all'indirizzo contenuto in [ebp+C], mentre in [ebp+8] abbiamo l'ibase del packer Pepsi che stiamo analizzando.

Quindi, prima di chiamare la VirtualProtect, avremo nel registro esi il valore 0x400080 (indirizzo dell'header PE del packer) e in ebx il valore 0x3300B8 (indirizzo dell'header PE del binario unpackato).

La VirtualProtect ha quindi il compito di cambiare la protezione di accesso alla memoria all'indirizzo 0x400080 che grazie al parametro 0x4 (PAGE_READWRITE) adesso sarà modificabile.

Il registro edx viene caricato con la dimensione della Resource Directory dell'eseguibile unpackato ([ebx+8C]). Tale valore viene scritto sull'header di Pepsi ([esi+8C]). Il registro edx viene adesso caricato con il VA della Resource Directory sempre dell'eseguibile unpackato ([ebx+88]) e anche in questo caso tale valore viene sovrascritto a quello presente nell'header di Pepsi ([edi+88]).

Quindi questa funzione non fa altro che copiare i dettagli relativi alla Resource Directory (dimensione e virtual address) dall'header del binario unpackato e sovrascriverli a quelli presenti nell'header del packer.

Una volta usciti da questa chiamata, ne troveremo subito un'altra:

004191E8	8B15 08914100	mov edx,dword ptr ds:[<rva_pepsi>]
004191EE	0115 18914100	add dword ptr ds:[<reserved_space_for_unpacking>],edx
004191F4	2915 1C914100	sub dword ptr ds:[41911C],edx
004191FA	FF35 1C914100	push dword ptr ds:[41911C]
00419200	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
00419206	FF35 10914100	push dword ptr ds:[<virtualaddr_pepsi>]
0041920C	E8 46010000	call pepsi.419357

Grazie ai nomi significativi che abbiamo dato agli indirizzi di memoria, capire i parametri che vengono passati alla funzione adesso è facile.

L'RVA del segmento .pepsi (0x1000) viene aggiunto all'indirizzo di inizio della memoria dove risiede l'eseguibile unpackato. Subito dopo viene anche sottratto alla grandezza del suo stesso segmento (contenuta in [41911C]): quindi adesso in [41911C] ci sarà il valore 0x17000, che è il primo ad essere pushato sullo stack. Il secondo valore ad essere pushato è l'indirizzo di inizio della memoria dove risiede l'eseguibile unpackato con l'aggiunta effettuata precedentemente di 0x1000: probabilmente l'autore del packer sta cercando di referenziare il primo segmento dopo l'header dell'eseguibile unpackato! Il terzo valore pushato è il VA del segmento .pepsi.

Già con questi dati possiamo ipotizzare quello che succederà appena entreremo nella chiamata: il segmento .pepsi del packer verrà sovrascritto con i dati dell'eseguibile unpackato partendo da 0x1000.

Scopriamo se abbiamo ragione entrando nella funzione:

00419357	55	push ebp
00419358	8BEC	mov ebp, esp
0041935A	56	push esi
0041935B	57	push edi
0041935C	FC	cld
0041935D	8B75 0C	mov esi, dword ptr ss:[ebp+C]
00419360	8B7D 08	mov edi, dword ptr ss:[ebp+8]
00419363	8B4D 10	mov ecx, dword ptr ss:[ebp+10]
00419366	D1E9	shr ecx, 1
00419368	F366:A5	rep movsw
0041936B	5F	pop edi
0041936C	5E	pop esi
0041936D	C9	leave
0041936E	C2 0C00	ret C

ESATTO! Abbiamo proprio ragione! L'opcode cld imposta la direzione di avanzamento della copia. Nel registro esi abbiamo l'indirizzo 0x331000 ovvero il primo segmento dell'eseguibile unpackato. Nel registro edi viene spostato il valore 0x401000, ovvero il VA del segmento .pepsi del packer. In ecx abbiamo 0x17000, ovvero la dimensione dei dati da copiare. L'opcode "rep movsw" avvia la copia.

Usciti da questa funzione possiamo dire di avere le idee più chiare: l'eseguibile unpackato con la IAT ricostruita è stato copiato sul segmento .pepsi e l'header del packer è stato opportunamente modificato per avere la Resource Directory corretta in relazione all'eseguibile originale!

Usciti da questa funzione resta ben poco da analizzare:

00419211	2915 18914100	sub dword ptr ds:[<reserved_space_for_unpacking>],edx
00419217	FF35 0C914100	push dword ptr ds:[<virtualsize_pepsi>]
0041921D	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
00419223	E8 42030000	call <JMP.&RtlZeroMemory>
00419228	68 00400000	push 4000
0041922D	FF35 0C914100	push dword ptr ds:[<virtualsize_pepsi>]
00419233	FF35 18914100	push dword ptr ds:[<reserved_space_for_unpacking>]
00419239	E8 38030000	call <JMP.&VirtualFree>
0041923E	FF15 20914100	call dword ptr ds:[419120]
00419244	6A 00	push 0
00419246	E8 07030000	call <JMP.&ExitProcess>

Abbiamo due chiamate, rispettivamente a RtlZeroMemory e VirtualFree che non fanno altro che rimuovere qualsiasi traccia dell'eseguibile unpackato dalla memoria (furbi eh?! 😊), seguite da una call a [419120].

Sappiamo già che a [419120] vi è l'indirizzo assoluto dell'entry point calcolato in precedenza: questa call non fa altro che passare il controllo all'eseguibile unpackato!

Infatti, una volta entrati dentro questa call ci troviamo qui:

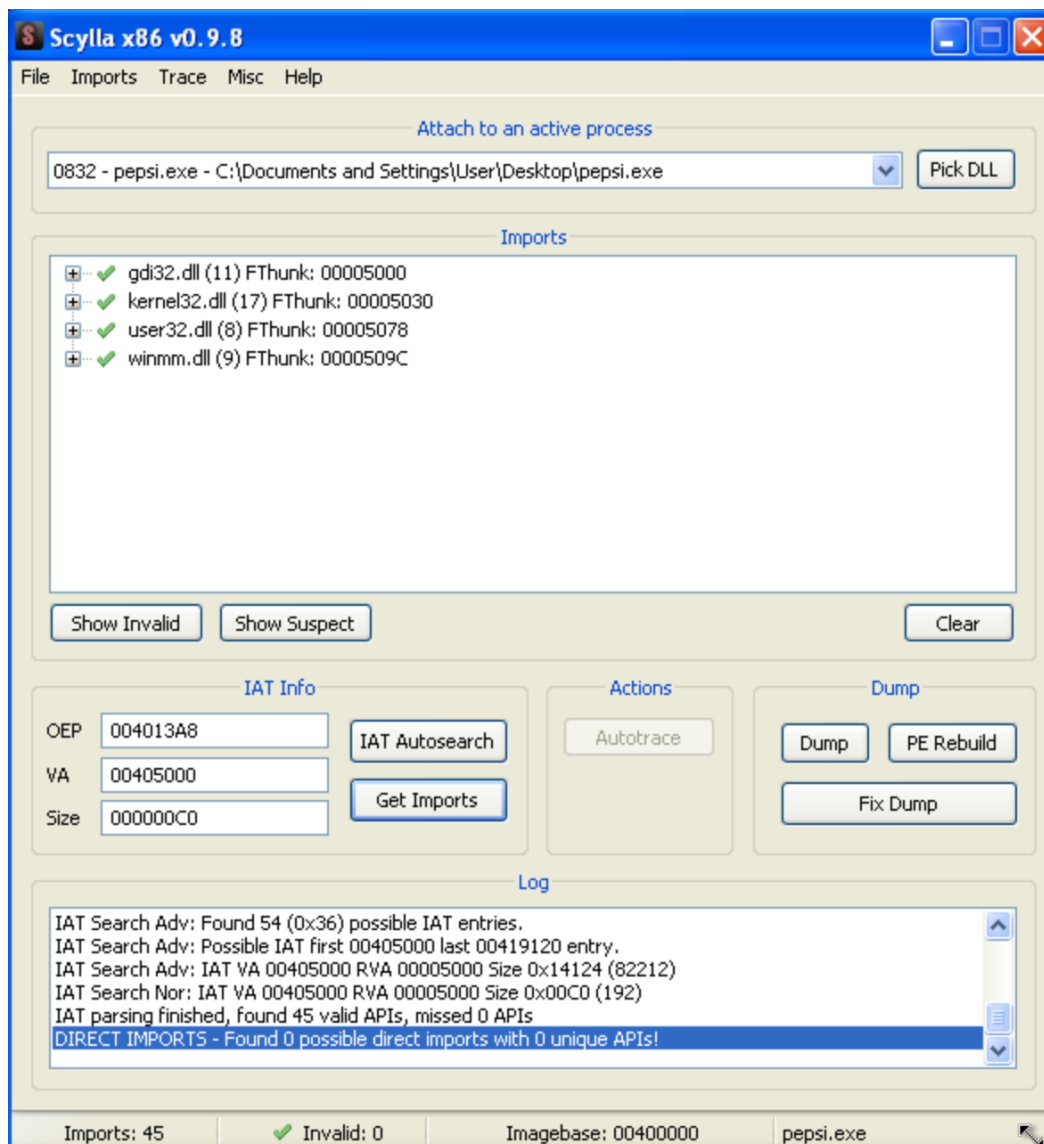
004013A8	56	push esi
004013A9	6A 0A	push A
004013AB	68 F4010000	push 1F4
004013B0	FF35 6D634000	push dword ptr ds:[40636D]
004013B6	E8 97000000	call <JMP.&FindResourceA>
004013BB	50	push eax
004013BC	50	push eax
004013BD	FF35 6D634000	push dword ptr ds:[40636D]
004013C3	E8 B4000000	call <JMP.&SizeofResource>
004013C8	A3 71634000	mov dword ptr ds:[406371],eax
004013CD	58	pop eax
004013CE	50	push eax

Siamo arrivati all'OEP (original entry point)!! Adesso possiamo dumpare con Scylla.

Poiché saranno presenti delle difficoltà, legate alla configurazione di Scylla, ho deciso di scrivere tre sotto capitoli separati sul dumping, per illustrare tutte e tre le possibilità.

Dumping: Metodo A - Fix Manuale

Arrivati all'OEP apriamo Scylla (pulsante con icona a forma di 'S' dalla barra degli strumenti di x32dbg), inseriamo l'indirizzo dell'OEP e clicchiamo su "IAT Autosearch". Scegliamo "no" quando ci viene proposto di usare il metodo avanzato di ricerca. Ci ritroveremo con questa configurazione:



Perfetto, clicchiamo su "Dump" e quindi su "Fix Dump".

Proviamo ad eseguire il nostro programma unpackato e.... NON FUNZIONA!!

Perché non funziona? La risposta è piuttosto semplice: nella configurazione standard di Scylla, l'header del dump viene copiato dal binario presente sul disco e non da quello in memoria. Se vi ricordate l'header di Pepsi è stato modificato quando i valori relativi alla grandezza e al VA della Resource Directory sono stati modificati! Il nostro

dump quindi non ha questi valori aggiornati e dobbiamo procedere modificandoli manualmente.

La strada più veloce è aprire il nostro dump (quello con la IAT fixata ovviamente) in un editor esadecimale come HxD e spostarci all'offset dove risiedono i dati relativi alla grandezza e al VA della resource directory.

Nel nostro caso avremo il VA della Resource Directory a 0x109 e la sua grandezza a 0x10C.

Per ottenere i valori corretti riavviamo il debugger e mettiamo un breakpoint dove avviene la modifica di tali dati, ovvero:

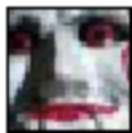
- 1) 0x41927A per ottenere la grandezza della Resource Directory
- 2) 0x419286 per ottenere il VA della Resource Directory

Otteniamo quindi 9D9C come grandezza e E000 come VA.

Procediamo patchando il binario:

```
000000C0 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 .....
000000D0 00 B0 01 00 00 02 00 00 00 00 00 02 00 00 00 00 .°.
000000E0 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00 .....
000000F0 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 .....
00000100 C4 A0 01 00 64 00 00 00 00 E0 00 00 9C 9D 00 00 Ä ..d...à..ce.
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Salviamo e notiamo come adesso l'eseguibile ha anche un'icona:



pepsi_dump_SCY.exe

Proviamo ad avviarlo e tutto funziona! Il packer è stato rimosso:

```
UnpackMe: Pepsi packer v2 - www.tuts4you.com

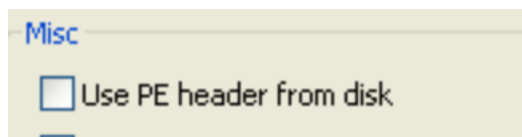
UnpackMe by Xylitol for tuts4you
Packed by Pepsi Packer v2
Author of the packer: Slick

Simple rule:
Manual unpacking only !
```

Dumping: Metodo B - Automatico

Nel primo metodo abbiamo patchato manualmente l'header del dump per sistemare i valori della Resource Directory, adesso vediamo come configurare adeguatamente Scylla per avere un dump perfetto senza dover modificare nulla.

Torniamo all'OEP e apriamo Scylla, configuriamolo esattamente come abbiamo fatto in precedenza, ma prima di procedere al Dump, clicchiamo su Misc e quindi su Options. Dalla finestra che si aprirà togliamo il segno di spunta dalla voce "Use PE header from disk":



Proseguiamo effettuando regolarmente il Dump e clicchiamo su "Fix Dump" per ottenere un eseguibile unpackato funzionante al 100%!

Dumping: Metodo C - Estrazione File Unpackato Dalla Memoria Temporanea

Questo metodo è un po' più complicato di quelli discussi in precedenza, ma ho voluto aggiungerlo in quanto lo ritengo interessante.

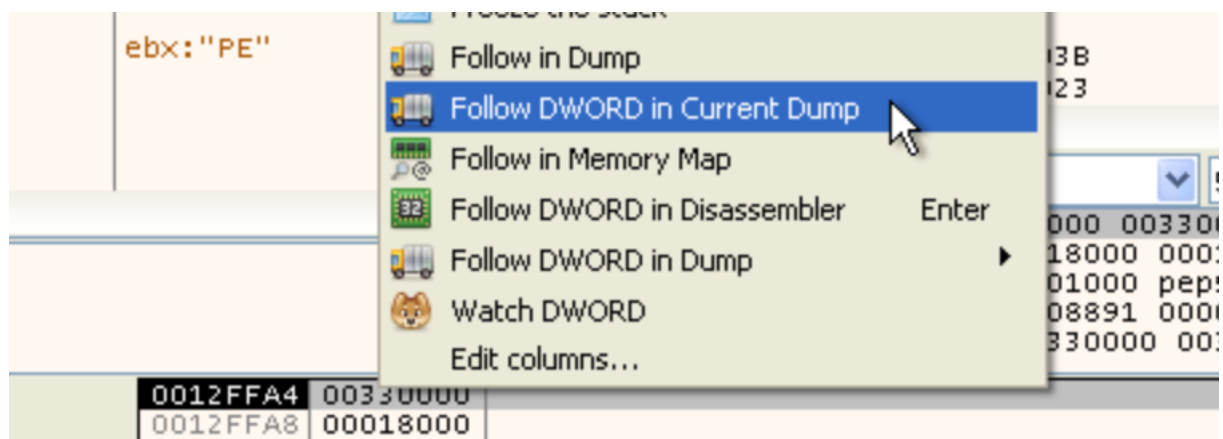
Durante l'analisi abbiamo scoperto che per quasi tutta l'esecuzione del packer è presente il file unpackato in un'area di memoria temporanea e che dopo essere stato copiato sul segmento .pepsi, viene cancellato.

Inoltre, abbiamo scoperto che una funzione aggiungerà una IAT valida proprio al file unpackato che si trova in questa area di memoria.

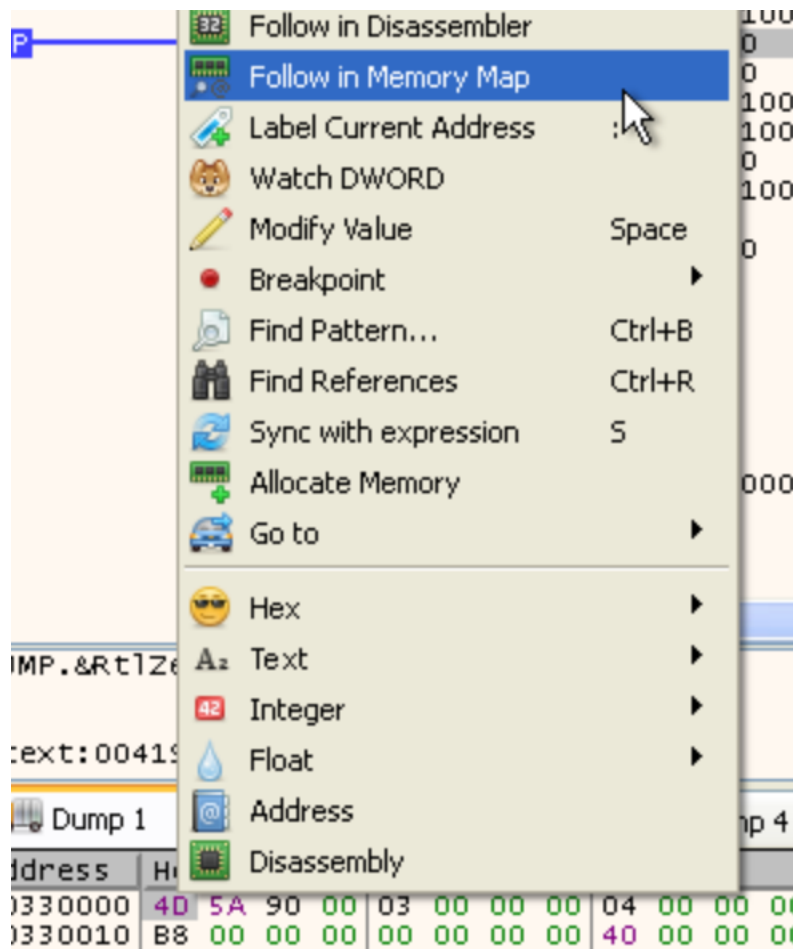
Possiamo fare il dump di quest'area di memoria, mettendo un breakpoint dal debugger subito prima che venga sovrascritta, ovvero dove avviene la call a RtlZeroMemory:

00419217	FF35 0C914100	push dword ptr ds:[<virtualsi
0041921D	FF35 18914100	push dword ptr ds:[<reserved_s
00419223	E8 42030000	call <JMP.&RtlZeroMemory>
00419228	68 00400000	push 4000
0041922D	FF35 0C914100	push dword ptr ds:[<virtualsi

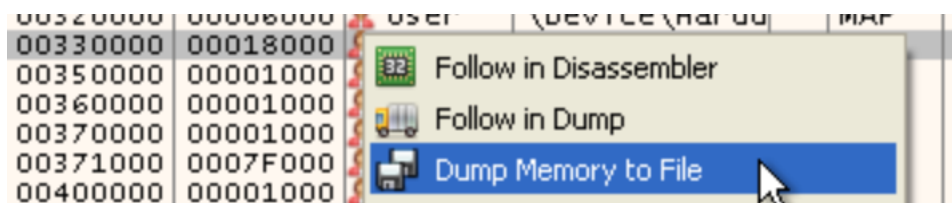
Ci troviamo fermi subito prima della cancellazione di questa area di memoria. Sapendo che l'indirizzo in questione è stato l'ultimo ad essere stato pushato, possiamo raggiungerlo nella Memory Map cliccando con il tasto destro sulla relativa finestra dello stack view e scegliendo "Follow DWORD in Current Dump":



E subito dopo dall'hex view clicchiamo su "Follow In Memory Map":



Ci ritroveremo sul tab Memory Map con la relativa sezione di memoria che ci interessa già selezionata. Possiamo procedere cliccando con il tasto destro e scegliendo “Dump Memory to File”:



Adesso abbiamo il file unpackato estratto dalla sezione di memoria temporanea, che ha anche una IAT valida, ma c'è un problema: l'eseguibile non può ancora funzionare perché è mappato in memoria per essere eseguito. Possiamo fare una verifica aprendolo con CFF:

pepsi_00330000.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Lin
000001B0	000001B8	000001BC	000001C0	000001C4	000001C8	000001CC	000001D0	00
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Wc
.text	000032A2	00001000	00003400	00000400	00000000	00000000	0000	00
.rdata	000004E6	00005000	00000600	00003800	00000000	00000000	0000	00
.data	00007384	00006000	00000200	00003E00	00000000	00000000	0000	00
.rsrc	00009D9C	0000E000	00009E00	00004000	00000000	00000000	0000	00

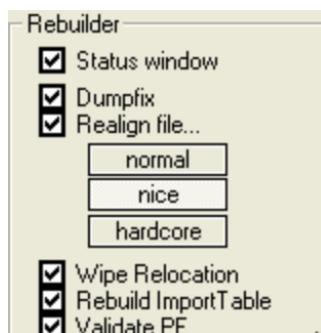
This section contains:
Code Entry Point: 000013A8

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000B80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000B90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000BA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000BB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000BC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000BD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000BE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000BF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000C00	55	8B	EC	83	C4	B0	81	7D	0C	10	01	00	00	75	3F	C7	U i A° } ..u?Ç
00000C10	05	1D	61	40	00	7C	00	00	00	C7	05	25	61	40	00	00	a@. ...Ç %a@..
00000C20	00	00	00	C7	05	21	61	40	00	02	00	00	00	FF	75	08	...Ç la@. ...ÿu
00000C30	E8	71	04	00	00	50	E8	0B	01	00	00	6A	00	6A	2E	6A	èq ..Pè ..j..j
00000C40	01	FF	75	08	E8	69	04	00	00	E9	F1	00	00	00	83	7D	vu èi ..éñ... }

Come possiamo vedere tutte le sections sono disallineate.

Per risolvere il problema, occorre riallineare tutte le sezioni in modo tale che il binario possa essere caricato correttamente dal loader degli eseguibili di Windows. Questa operazione viene eseguita automaticamente da Scylla e viene chiamata "unmapping del dump". Possiamo procedere manualmente oppure facendoci aiutare da Lord PE.

Apriamo Lord PE, clicchiamo su options e mettiamo il segno di spunta sulla voce "Dumpfix":




Adesso clicchiamo su “Rebuild PE” e scegliamo il nostro dump. Confermiamo cliccando su OK e avremo il nostro eseguibile riallineato e completamente funzionante!

Possiamo verificare che le varie sections adesso siano allineate correttamente usando CFF:

pepsi_00330000.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...
00000138	00000140	00000144	00000148	0000014C	00000150	00000154	00000158
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word
.text	00004000	00001000	000032A2	00000200	00000000	00000000	0000
.rdata	00001000	00005000	000004E6	00003600	00000000	00000000	0000
.data	00008000	00006000	000001BE	00003C00	00000000	00000000	0000
.rsrc	00009D9C	0000E000	00009D9C	00003E00	00000000	00000000	0000

This section contains:
Code Entry Point: 000013A8



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	55	8B	EC	83	C4	B0	81	7D	0C	10	01	00	00	75	3F	C7	U i Ä* } ..u?Ç
00000010	05	1D	61	40	00	7C	00	00	00	C7	05	25	61	40	00	00	a@. ...Ç !%a@..
00000020	00	00	00	C7	05	21	61	40	00	02	00	00	00	FF	75	08	...Ç !a@. ...ÿu
00000030	E8	71	04	00	00	50	E8	0B	01	00	00	6A	00	6A	2E	6A	èq ...Pè ...j.j.j

Tutto è allineato correttamente.

Conclusione

Abbiamo completato l'analisi di questo packer e abbiamo documentato tre metodi diversi per effettuare l'unpacking.

Ricapitolando, Pepsi compie le seguenti operazioni:

- 1) Viene allocata un'area di memoria per uso temporaneo e al suo interno viene estratto l'eseguibile unpackato (senza IAT al momento)
- 2) Con le informazioni contenute in questa area di memoria viene calcolato e salvato temporaneamente l'indirizzo dell'OEP. Viene anche creata la IAT e scritta sempre su quest'area di memoria.
- 3) Vengono estrarre le informazioni relative alla Resource Directory (VA e dimensione) dall'header dell'eseguibile unpackato e sovrascritte sull'header di Pepsi.
- 4) L'eseguibile unpackato (partendo da 0x1000) viene copiato sul segmento .pepsi
- 5) L'area di memoria dove risiede l'eseguibile unpackato viene pulita
- 6) Viene lanciato l'OEP e l'esecuzione del programma originale inizia

Considerando che questo è il primo tentativo di SlicK nel creare un packer, posso solo fargli i miei complimenti 😊

Ringraziamenti

Come sempre ci tengo a ringraziare gli autori degli strumenti usati in documento. Un grazie in particolare va a SlicK per aver creato Pepsi e a Xylitol per aver creato questo unpackme.

Un grazie va anche a te che hai letto questo peper! Spero che l'analisi sia stata di tuo gradimento :)

Se vi è piaciuto questo documento e volete leggerne altri simili, visitate il mio sito:
<https://www.lucadamico.dev/>

Luca