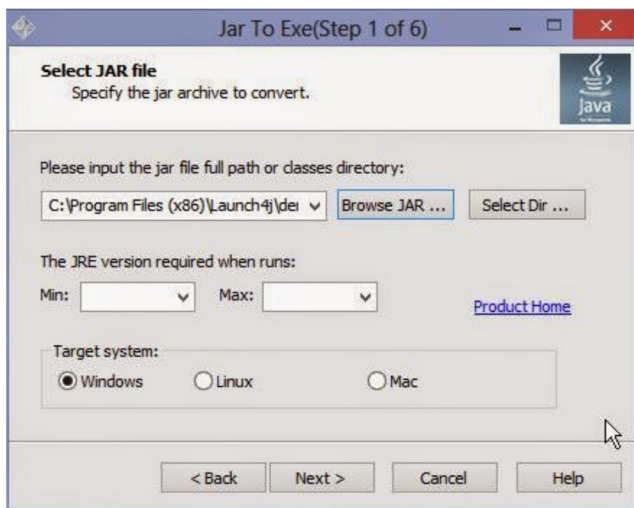


# Reverse Engineering Tips

Welcome to the Reverse Engineering Tips blog which is dedicated to unraveling the mysteries of reverse engineering. Together, we will learn how to work on many different targets with a vast assortment of tools.

Saturday, December 20, 2014

## Unpacking Jar2Exe 2.1: Extracting The Jar File At All 3 Protection Levels



Welcome to this extensive tutorial for unpacking Jar2Exe. Jar2Exe is a java executable wrapper which works by taking your original java archive, wrapping it into an executable, and executing it through a virtual environment using the jvm.dll provided with each java distribution. It also provides the ability to hide your archive and encrypt the class names making recovery difficult. The goal of this tutorial is to demonstrate how to recover a jar file at the 3 different protection levels provided by Jar2Exe.

In this tutorial, I will be using the file SimpleApp.jar which is included with launch4j. You can protect it with the same settings to reverse along with me.

### Tools Needed:

Resource Hacker

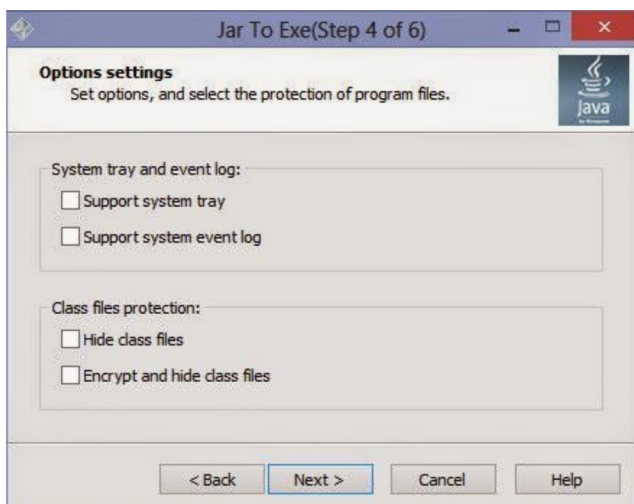
Winhex

Olydbg 1.10+ MemoryDump 0.9 and Olly Advanced or StrongOD Plugin(for advanced ctrl+g).

DJ Java Decompiler

7-Zip or Winrar

### Jar2Exe Level 1: No Hiding, No Encryption:



### Search This Blog

### Donations

If this blog is helpful to you,  
consider buying me a cup of coffee.

[Donate](#)

### About Me

Chester Fritz

[View my complete profile](#)

### Blog Archive

► 2015 (1)

▼ 2014 (3)

▼ December (3)

Keygenning With Delphi:  
Useful Delphi Functions  
an...

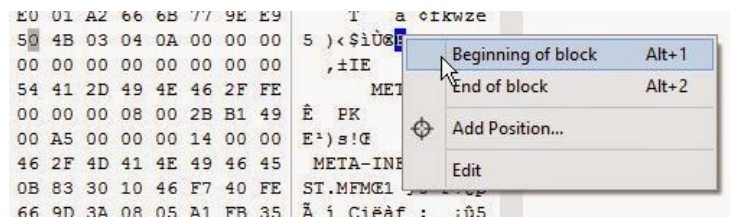
Unpacking Jar2Exe 2.1:  
Extracting The Jar File  
At ...

Unpacking Launch4j 3.5:  
Extracting The Jar File

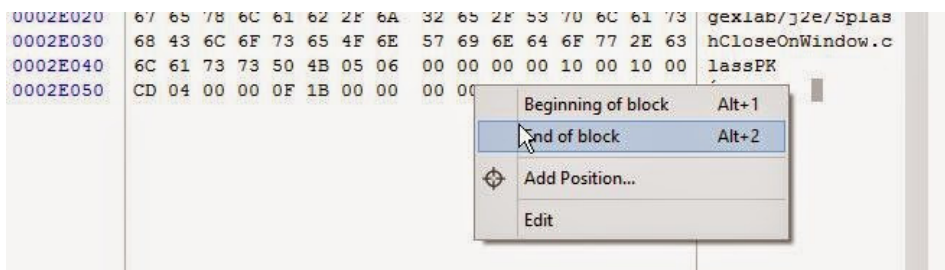
This is the default wrapping level which provides no protection to your java file. As you can see, the Hide class files and Encrypt and hide class files options are left unchecked. This level of protection simply takes your java archive/jar file, concatenates it to the end of the executable, and embeds its java files inside of it. This java archive can be recovered using a hex editor, just like we did with launch4j in the previous tutorial. To begin, we will open sample file called 'SimpleAppNoHide.exe' in winhex. To find the archive, scroll to the bottom of the file. Another option you can try other than scrolling is to search for the ascii string "serial "(with the space). This should take you directly to the archive, but I cannot guarantee that this approach will always be successful.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		
0002BF40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BF50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BF60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BF70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BF80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BF90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BFA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BFB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BFC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BFD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BFE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002BFF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
0002C000	EF	BB	BF	73	65	72	69	61	6C	20	30	30	30	30	30	38	i>serial 000008	
0002C010	31	37	38	74	37	35	31	35	78	0D	0A	6D	61	69	6E	63	178t7515x mainc	
0002C020	6C	61	73	73	20	6E	65	74	2E	73	66	2E	6C	61	75	6E	lass net.sf.laun	
0002C030	63	68	34	6A	2E	65	78	61	6D	70	6C	65	2E	53	69	6D	ch4j.example.Sim	
0002C040	70	6C	65	41	70	70	0D	0A	0D	0A	6D	69	6E	6A	72	65	pleApp minjre	
0002C050	20	0D	0A	00	54	00	00	00	E0	01	A2	66	6B	77	9E	E9	T à cfwkzé	
0002C060	35	90	29	8B	24	EC	D9	AE	50	4B	03	04	0A	00	00	00	5 )<\$iÜ\$K	
0002C070	00	00	2C	B1	49	45	00	00	00	00	00	00	00	00	00	00	,±IE	
0002C080	00	00	09	00	04	00	4D	45	54	41	2D	49	4E	46	2F	FE	META-INF/p	
0002C090	CA	00	00	50	4B	03	04	0A	00	00	00	08	00	2B	B1	49	Ê PK ±±I	
0002C0A0	45	B9	29	73	21	8C	00	00	00	A5	00	00	00	14	00	00	E³)s!G ¥	
0002C0B0	00	4D	45	54	41	2D	49	4E	46	2F	4D	41	4E	49	46	45	META-INF/MANIFE	
0002C0C0	53	54	2E	4D	46	4D	8C	31	0B	83	30	10	46	F7	40	FE	ST.MFMG1 f0 F÷@p	
0002C0D0	C3	8D	ED	90	43	69	EB	E0	66	9D	3A	08	05	A1	FB	35	Ã í Cieáf : ;û5	
0002C0E0	3D	31	45	CF	E0	45	A8	FF	BE	E9	D6	E9	83	F7	3D	5E	=1EÏaE`y%âÖéf÷=^	
0002C0F0	47	12	06	D6	E4	1E	BC	6A	58	A4	86	12	0B	6B	1A	F9	G Œa 4jXx+ k ù	
0002C100	23	4D	24	3F	32	64	96	CF	0A	2F	D6	B4	2B	53	E2	97	#M\$?2d-Ï /Ö'+Sâ-	
0002C110	BB	EE	D9	AF	F0	E4	9E	45	09	87	7E	13	E8	82	5F	17	»iÜ`âažE ±~ è, _	
0002C120	DD	35	F1	AC	70	13	8F	47	6B	3A	0A	E2	DA	89	54	6B	Ý5ñ-p Gk: áÜkTk	
0002C130	10	4E	A8	03	4E	B4	89	1F	CF	6F	E4	0F	CD	71	62	EC	N` N`k Ioâ íqbl	
0002C140	C3	6F	9A	18	73	FC	67	BA	3B	A5	B1	06	04	6B	AC	F9	Ãoš süg°;±i k-ù	
0002C150	02	50	4B	03	04	0A	00	00	00	00	00	2C	B1	49	45	00	PK ,±IE	
0002C160	00	00	00	00	00	00	00	00	00	00	00	04	00	00	00	00	n	
0002C170	65	74	2F	50	4B	03	04	0A	00	00	00	00	00	00	2C	B1	49	et/PK ,±I
0002C180	45	00	00	00	00	00	00	00	00	00	00	00	00	00	07	00	00	E
0002C190	00	6E	65	74	2F	73	66	2F	50	4B	03	04	0A	00	00	00	00	net/sf/PK
0002C1A0	00	00	2C	B1	49	45	00	00	00	00	00	00	00	00	00	00	00	,±IE
0002C1B0	00	00	10	00	00	00	00	6E	65	74	2F	73	66	2F	6C	61	75	net/sf/laun
0002C1C0	6E	63	68	34	6A	2F	50	4B	03	04	0A	00	00	00	00	00	00	nch4j/PK
0002C1D0	2C	B1	49	45	00	00	00	00	00	00	00	00	00	00	00	00	00	,±IE
0002C1E0	18	00	00	00	6E	65	74	2F	73	66	2F	6C	61	75	6E	63	63	net/sf/launc
0002C1F0	68	34	6A	2F	65	78	61	6D	70	6C	65	2F	50	4B	03	04	00	h4j/example/PK
0002C200	0A	00	00	00	08	00	2C	B1	49	45	33	90	3F	BD	C7	01	01	,±IE3 ?4Ç
0002C210	00	00	45	03	00	00	29	00	00	00	6E	65	74	2F	73	66	66	E ) net/sf
0002C220	2F	6C	61	75	6E	63	68	34	6A	2F	65	78	61	6D	70	6C	6C	/launch4j/exampl
0002C230	65	2F	53	69	6D	70	6C	65	41	70	70	24	31	2E	63	6C	6C	e/SimpleApp\$1.cl
0002C240	61	73	73	8D	52	5D	4F	13	41	14	3D	D3	76	BB	B0	5D	5D	ass R]O A =Öv»°]

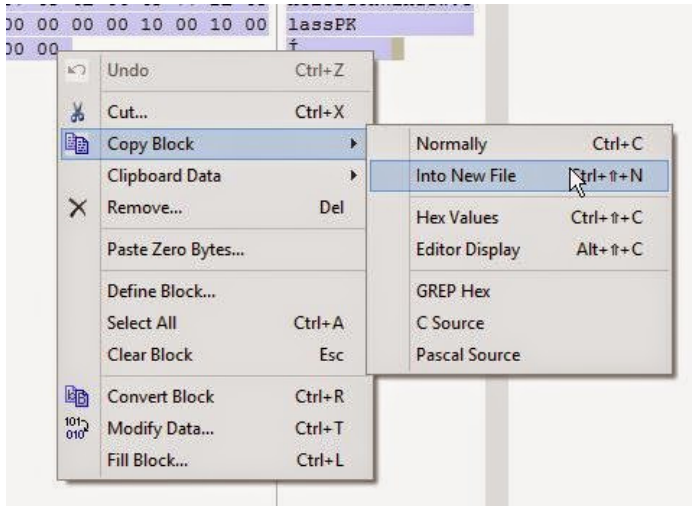
Once you are here, the first occurrence of 'PK' labels the start of our archive. We can label this as our beginning of block:



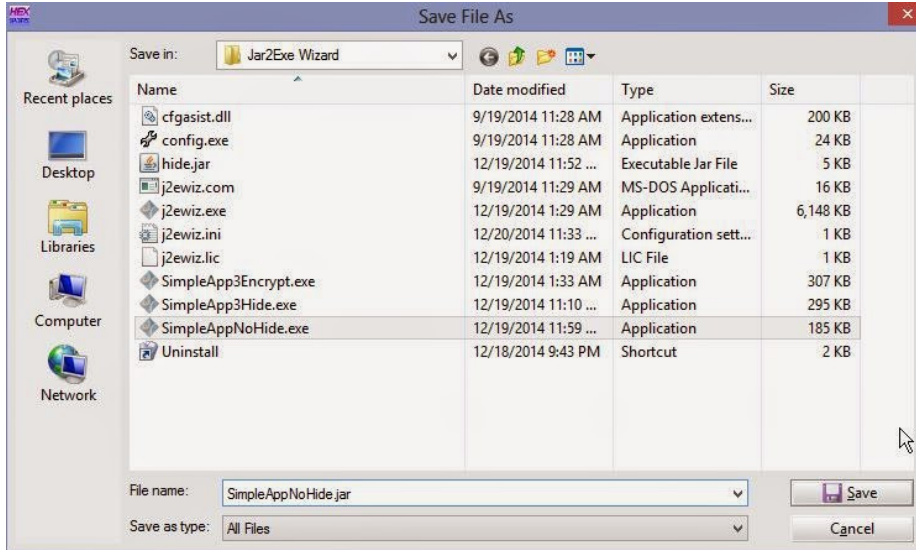
Afterwards, we can scroll to the very last byte in the file and label it as our End of Block:



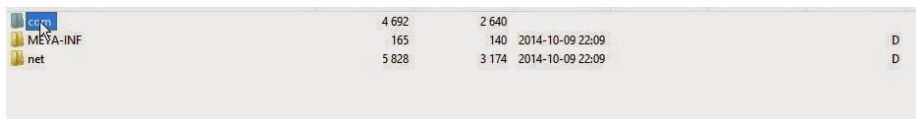
Once you have done so, right click the selected block and click Edit. In the new pane, go to Copy Block -> Into New File.



Afterwards, we can save it as a jar file. I used the name SimpleAppNoHide.jar:

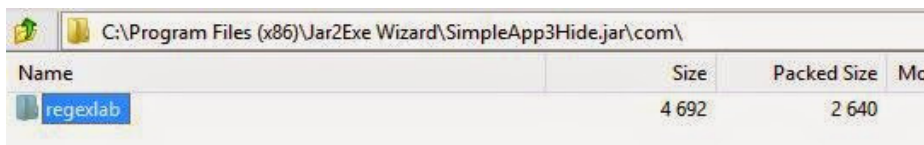


Once we are finished, the program will run correctly, but to be tidy, we need to delete the extra files that Jar2Exe added to our archive. Let's open the archive in 7-zip or winrar (whichever you fancy).

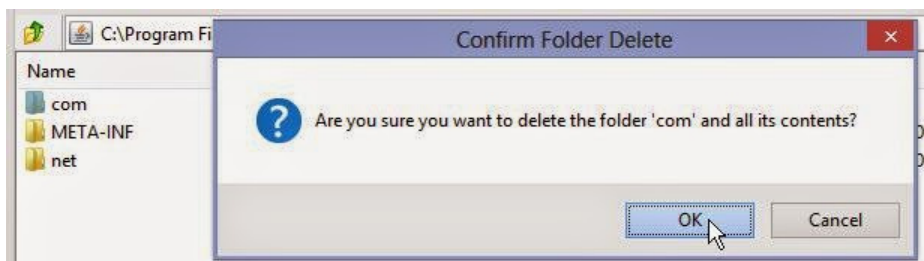


Once it is opened, you can see that Jar2Exe adds an additional directory called 'com'. While some other java applications may use this directory, Jar2Exe adds an additional subdirectory called regexlab.

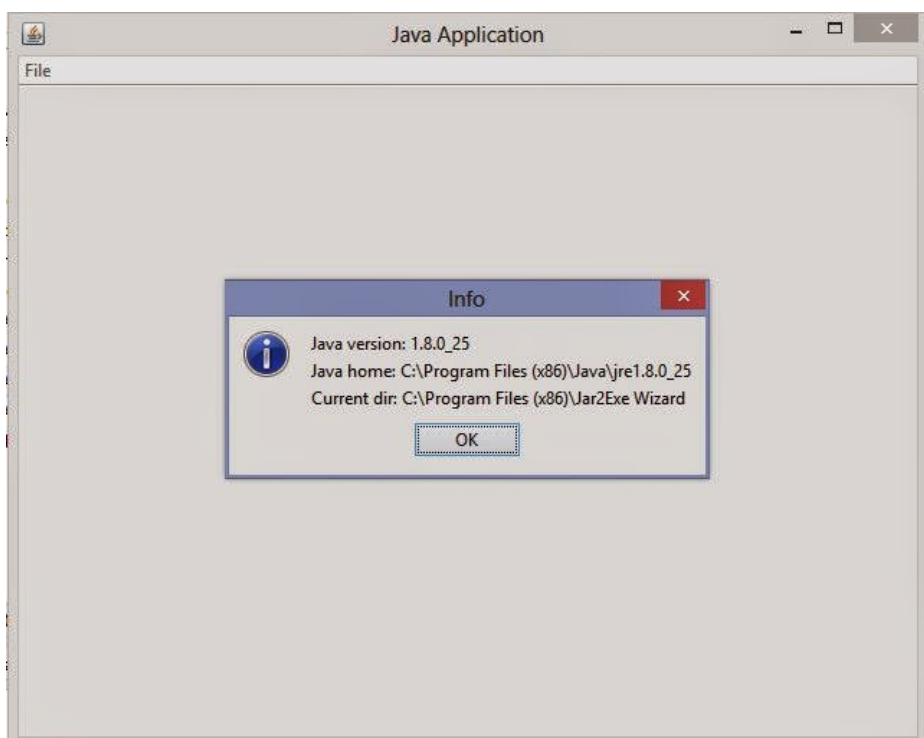




After entering the com directory, we see that regexlab is the only subdirectory it contains, meaning that the entire com folder is unused and can simply be deleted from the archive, let's go ahead and delete it:

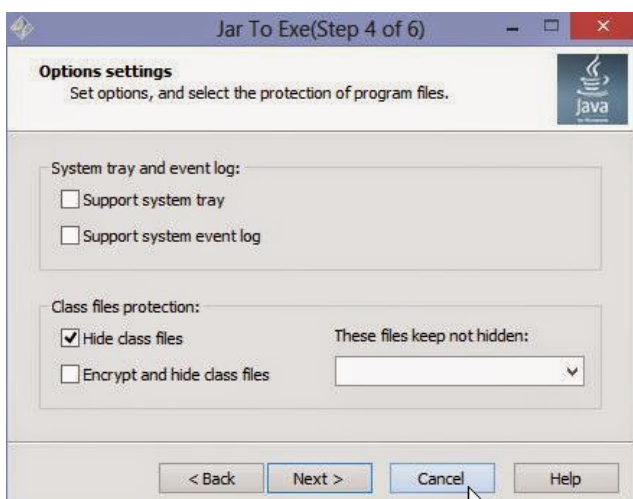


After confirming the deletion, we can close the archive and run the jar file.

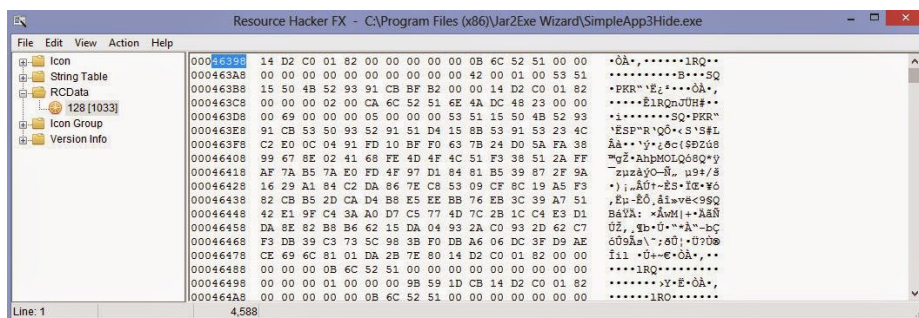


If you did everything correctly, the application should run without problems and our work is finished.

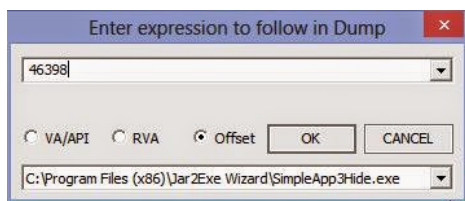
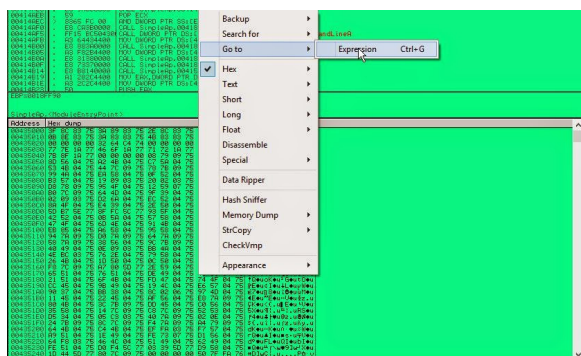
#### Jar2Exe Level 2: Hidden Archive:



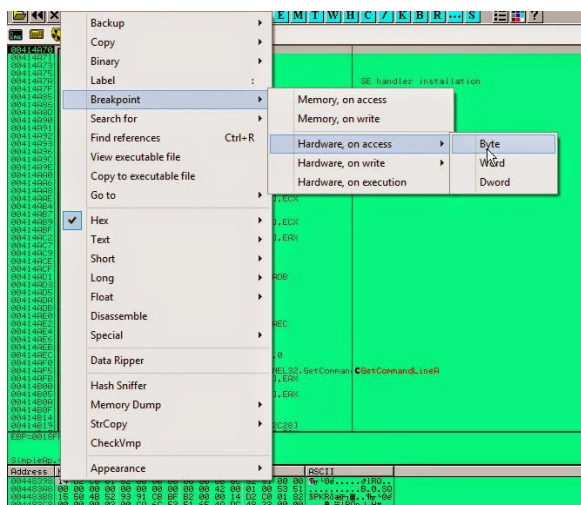
With the level 2 protection, Jar2Exe takes our java archive, encrypts it, and adds it as an RCDATA entry in the resource directory. To find the offset of the encrypted archive, we need to open our executable which I named 'SimpleAppHide.exe' in a resource editor. For this, I prefer resource hacker:



Above, we have located the encrypted data in the "RCDATA" section of the resource table. In this case, our offset is 46398h. Keep a note of this and open the application in Ollydbg. Once you have it opened, go to the hex dump section and go to the offset above:



Once you arrive at the offset, right click the byte and add a Hardware Breakpoint On Access -> Byte:



Now, let's run the program and wait for the break to occur. We should arrive here:

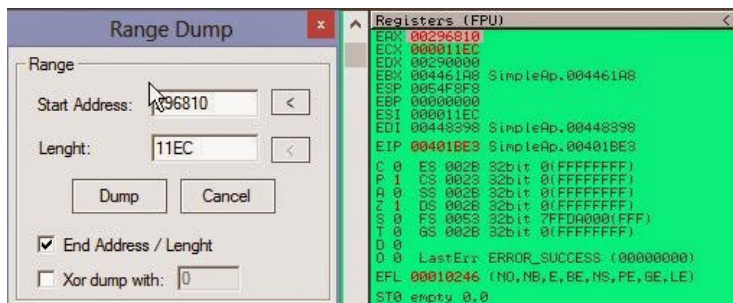
```

00401EC7  8B43          MOV EAX, DWORD PTR DS:[ECX+ED1]
00401EC9  8B43          MOV EAX, DWORD PTR DS:[ECX+ED1]
00401ECB  C0FA 06      SHR DL, 6
00401ECF  80EC 03      AND DL, 3
00401ED2  C0E0 02      SHL DL, 2
00401ED5  0000         OR DL, 0
00401ED7  8B4424 14     MOV EAX, DWORD PTR SS:[ESP+14]
00401ED9  8B1408       MOV BYTE PTR DS:[EAX+ECX], DL
00401EDE  41           INC ECX
00401EDF  3BCE        CMP ECX, ESI
00401BE1  7C E4       JL SHORT SimpleAp.00401BC7
00401BE3  996C24 18     CMP DWORD PTR SS:[ESP+18], EBP

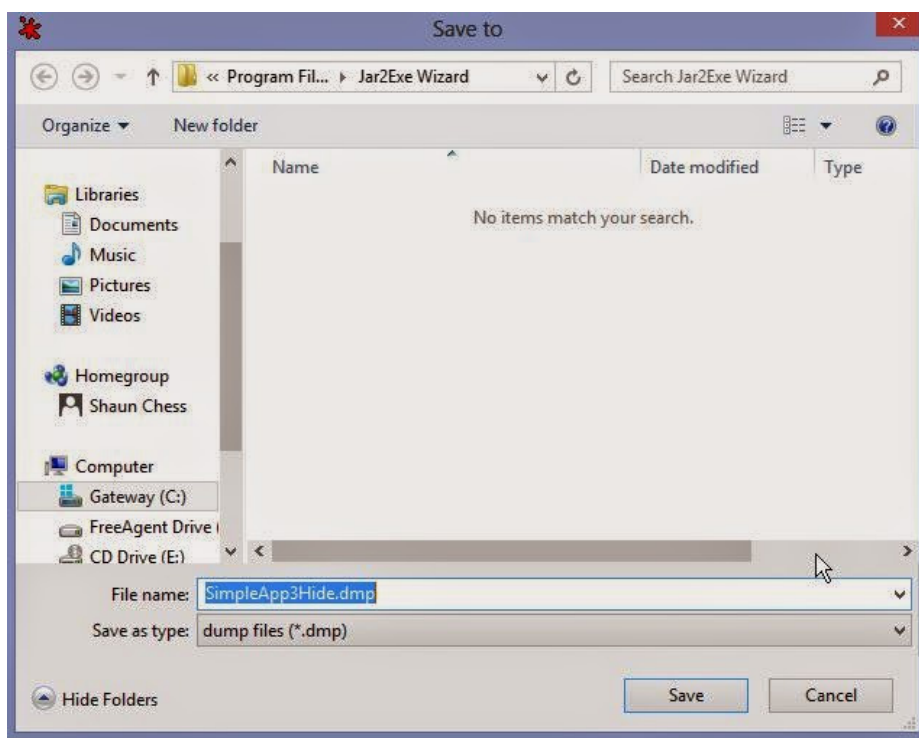
```

This loop will decrypt our archive byte for byte. Let's toggle a breakpoint on the instruction after the loop. In this example, the instruction is 00401BE3 >

CMP DWORD PTR SS:[ESP+18],EBP. Run the program now and let the decryption complete. Once we break here, the register EAX will contain the location where our decrypted jar file is located and ECX contains its size. Using this information, we are ready to dump the file from memory. To do this, I found a very simple plugin called Memory Dump. It allows you to simply specify a memory address and size to dump those bytes to disk. Let's open the plugin:



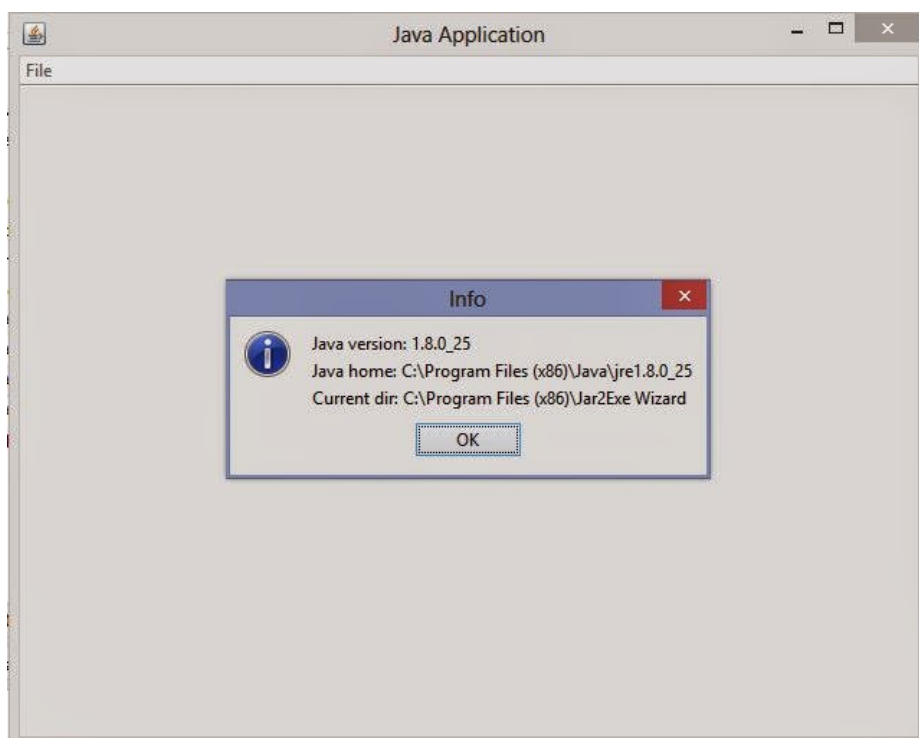
To begin, check the field which says End Address / Length. This will allow us to enter the length value. Set the Start Address to the value in EAX and the lenght(length) equal to the value in ECX. In my case, the address is 296810 and the size is 11EC. Once you have entered these values, we are ready to dump this to disk.



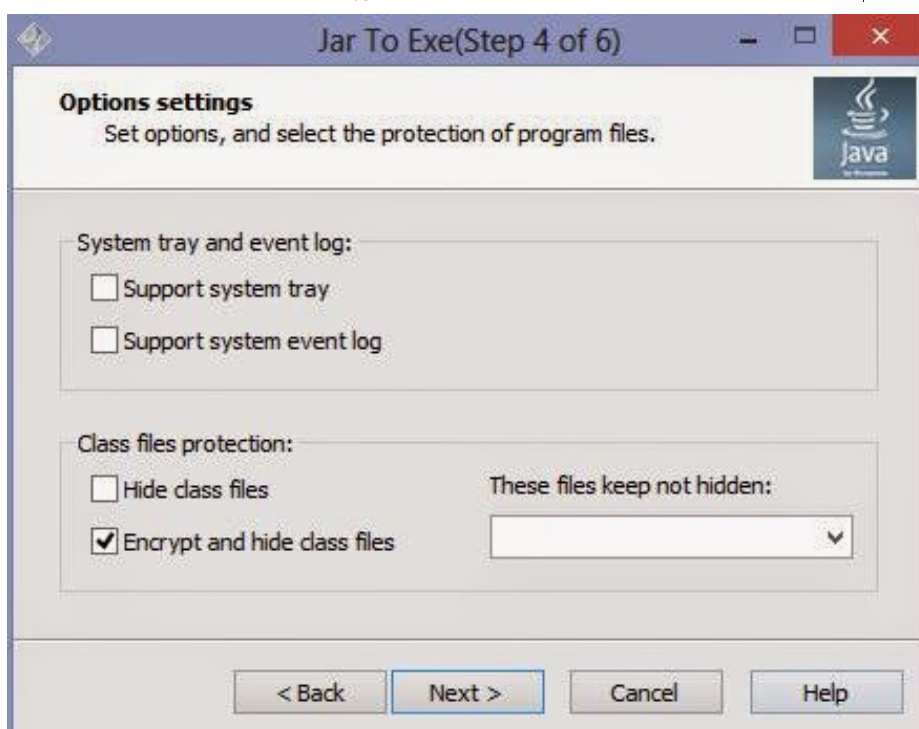
A limitation of this tool is that you can only save the file with a .dmp extension, but we can easily rename it once we are finished. Simply save it with the name you fancy. In this case I named it as SimpleApp3Hide.dmp for easy recognition.

SimpleApp3Hide.jar

After a quick renaming, the file is ready to run.



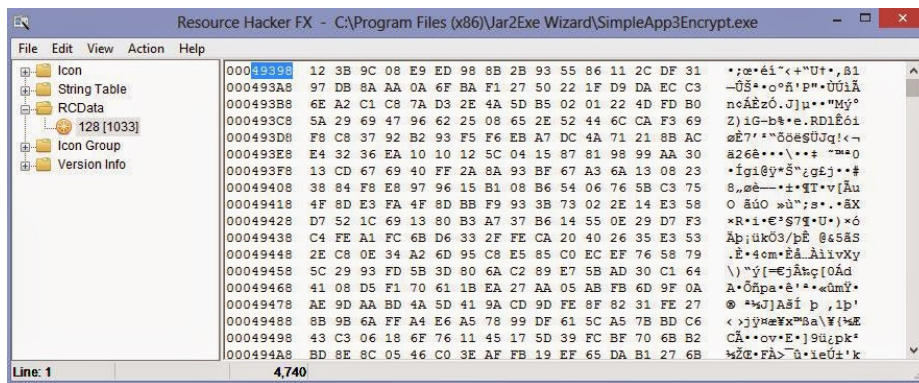
#### Jar2Exe Level 3: Hidden Archive + Encrypted Class Names:



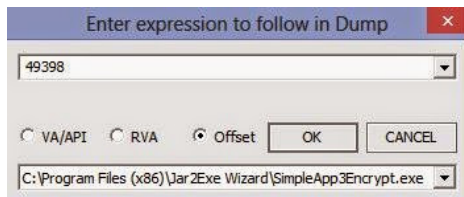
This is the strongest protection offered by Jar2Exe. First, it changes all of the class file names to gibberish. Second, it encrypts and hides the files just like the level 2 protection. While we can recover the archive with a similar method as in Level 2, we need to take an additional step to repair the class names.

To begin, let's open the application "SimpleAppEncrypt.exe" in a resource editor just as before to acquire the offset of the encrypted RCData:

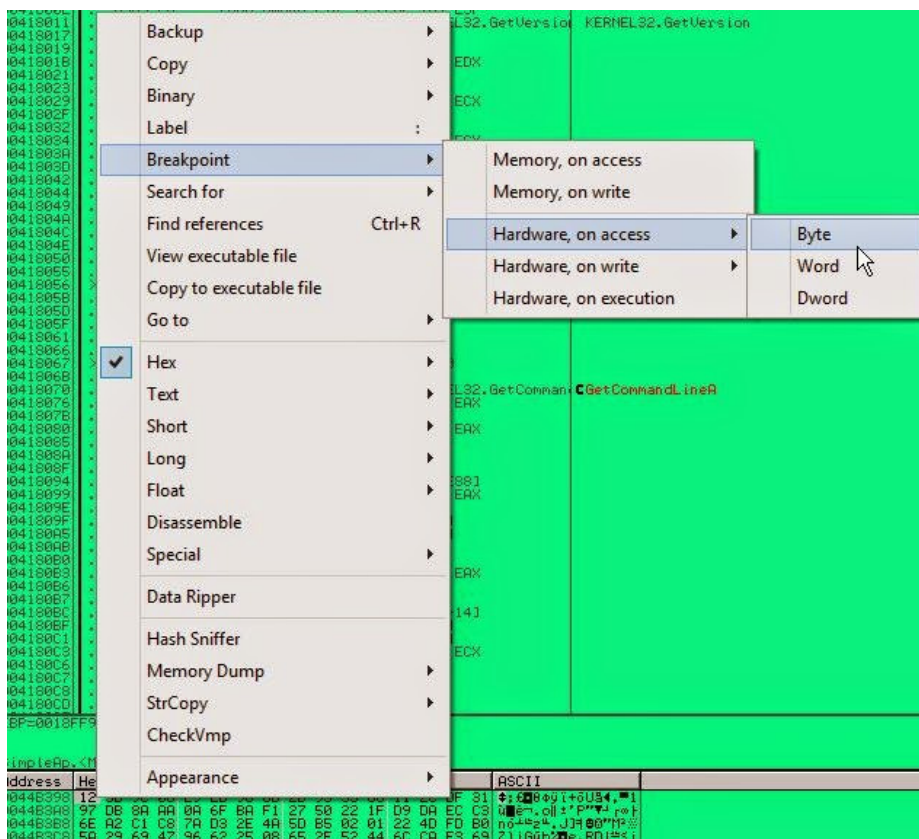




Our offset in this case is 49398h. Let's note that and open the application in Ollydbg and follow the offset.



Once we arrive at the offset, place a hardware breakpoint on access-> byte just like before.



Run the program and wait for the breakpoint. We should arrive at the following code:



```

0040AC89 | . 8BAE 3C020000 | MOV EBP,DWORD PTR DS:[ESI+23C]
0040AC8F | . 8BBE 34020000 | MOV EDI,DWORD PTR DS:[ESI+234]
0040AC95 | . 8A00 | MOV AL,BYTE PTR DS:[EAX]
0040AC97 | . 49 | DEC ECX
0040AC98 | . 85C9 | TEST ECX,ECX
0040AC9A | . 8D142F | LEA EDX,DWORD PTR DS:[EDI+EBP]
0040AC9D | . 7E 23 | JLE SHORT SimpleAp.0040ACC2
0040AC9F | . 8B9E 38020000 | MOV EBX,DWORD PTR DS:[ESI+238]
0040ACA5 | . 2BD8 | SUB EBX,EDX
0040ACA7 | . 03FB | ADD EDI,EBX
0040ACA9 | . 8A1A | MOV BL,BYTE PTR DS:[EDX]
0040ACAB | . 8AD8 | CMP BL,AL
0040ACAD | . 77 04 | JA SHORT SimpleAp.0040ACB3
0040ACAF | . 2AC3 | SUB AL,BL
0040ACB1 | . EB 06 | JMP SHORT SimpleAp.0040ACB9
0040ACB3 | . F6D3 | NOT BL
0040ACB5 | . FEC3 | INC BL
0040ACB7 | . 02C3 | ADD AL,BL
0040ACB9 | . 8A1C17 | MOV BL,BYTE PTR DS:[EDI+EDX]
0040ACBC | . 32C3 | XOR AL,BL
0040ACBE | . 4A | DEC EDX
0040ACBF | . 49 | DEC ECX
0040ACC0 | . 75 E7 | JNZ SHORT SimpleAp.0040AC99
0040ACC2 | . 8A4D 00 | MOV CL,BYTE PTR SS:[EBP]
0040ACC5 | . 3AC8 | CMP CL,AL
0040ACC7 | . 77 04 | JA SHORT SimpleAp.0040ACCD
0040ACC9 | . 2AC1 | SUB AL,CL
0040ACCB | . EB 06 | JMP SHORT SimpleAp.0040ACD3
0040ACCD | . F6D1 | NOT CL
0040ACCF | . FEC1 | INC CL
0040ACD1 | . 02C1 | ADD AL,CL
0040ACD3 | . 8B8E 38020000 | MOV ECX,DWORD PTR DS:[ESI+238]
0040ACD9 | . 8B7C24 14 | MOV EDI,DWORD PTR SS:[ESP+14]
0040ACDD | . 8A11 | MOV DL,BYTE PTR DS:[ECX]
0040ACDF | . 8A8E 41020000 | MOV CL,BYTE PTR DS:[ESI+241]
0040ACE5 | . 32D1 | XOR DL,CL
0040ACE7 | . 8B4C24 18 | MOV ECX,DWORD PTR SS:[ESP+18]
0040ACEB | . 32C2 | XOR AL,DL
0040ACED | . 8B01 | MOV BYTE PTR DS:[ECX],AL
0040ACF7 | . 8A9E 41020000 | MOV BL,BYTE PTR DS:[ESI+241]
0040ACF5 | . 32D8 | XOR BL,AL
0040ACF7 | . 8B4424 1C | MOV EAX,DWORD PTR SS:[ESP+1C]
0040ACFB | . 47 | INC EDI
0040ACFC | . 41 | INC ECX
0040ACFD | . 48 | DEC EAX
0040ACFE | . 8B9E 41020000 | MOV BYTE PTR DS:[ESI+241],BL
0040AD04 | . 897C24 14 | MOV DWORD PTR SS:[ESP+14],EDI
0040AD08 | . 894C24 18 | MOV DWORD PTR SS:[ESP+18],ECX
0040AD0C | . 894424 1C | MOV DWORD PTR SS:[ESP+1C],EAX
0040AD10 | . 0F85 30FFFFFF | JNZ SimpleAp.0040AC46
0040AD16 | . 5F | POP EDI
0040AD17 | . 5D | POP EBP
0040AD18 | . 5B | POP EBX
0040AD19 | . 5E | POP ESI
0040AD1A | . C2 0C00 | RETN 0C

```

This is  
the  
new

decryption routine. The instruction 0040ACED-> MOV BYTE PTR DS:[ECX],AL will move each decrypted byte to their new memory location.

```

0040ACE7 | . 8B4C24 18 | MOV ECX,DWORD PTR SS:[ESP+18]
0040ACEB | . 32C2 | XOR AL,DL
0040ACED | . 8B01 | MOV BYTE PTR DS:[ECX],AL

```

According to the code, the address of the memory location where our file is decrypted to can be found at the stack pointer [ESP+18].

```

ESP => 0000001F
ESP+4 0044B398 SimpleAp.0044B398
ESP+8 00000000
ESP+C 00001284
ESP+10 00401C58 RETURN to SimpleAp.0
ESP+14 0044B398 SimpleAp.0044B398
ESP+18 00296858
ESP+1C 00001284

```

The value at ESP+1C contains the size of the memory region. This can be verified by checking the decrement value of the loop.

```

0040ACF7 | . 8B4424 1C | MOV EAX,DWORD PTR SS:[ESP+1C]
0040ACFB | . 47 | INC EDI
0040ACFC | . 41 | INC ECX
0040ACFD | . 48 | DEC EAX
0040ACFE | . 8B9E 41020000 | MOV BYTE PTR DS:[ESI+241],BL
0040AD04 | . 897C24 14 | MOV DWORD PTR SS:[ESP+14],EDI
0040AD08 | . 894C24 18 | MOV DWORD PTR SS:[ESP+18],ECX
0040AD0C | . 894424 1C | MOV DWORD PTR SS:[ESP+1C],EAX

```

This code verifies that ESP+1C is the value decremented. ESP+1c is moved to EAX, decremented, and then moved back.

At our current location where the hardware breakpoint landed us, the values at ESP +18 and ESP+1C are the ones we need for our dump. Please note them and target a breakpoint on the instruction outside of the loop.

```

0040AD04 | . 897C24 14 | MOV DWORD PTR SS:[ESP+14],EDI
0040AD08 | . 894C24 18 | MOV DWORD PTR SS:[ESP+18],ECX
0040AD0C | . 894424 1C | MOV DWORD PTR SS:[ESP+1C],EAX
0040AD10 | . 0F85 30FFFFFF | JNZ SimpleAp.0040AC46
0040AD16 | . 5F | POP EDI
0040AD17 | . 5D | POP EBP
0040AD18 | . 5B | POP EBX
0040AD19 | . 5E | POP ESI
0040AD1A | . C2 0C00 | RETN 0C
0040AD1D | . 90 | NOP

```

Once you break here, we are ready to make a dump of the file. But first, I would like to show you the encrypted strings in our archive to give you a glimpse of how jar2exe encrypts them:

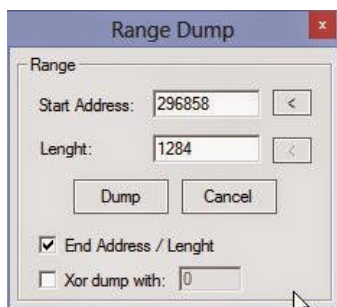
```

00296858 50 4B 03 04 0A 00 00 00 00 00 2C B1 49 45 00 00 PK.....IE..
00296860 00 00 00 00 00 00 00 00 00 00 20 00 04 00 35 33 .....53
00296878 64 32 37 37 32 61 65 66 32 63 62 34 35 65 34 35 d2772aef2cb45e45
00296888 65 64 63 32 32 35 31 39 34 32 31 83 37 62 FE CA edc2251942137b
00296898 00 00 50 4B 03 04 0A 00 00 00 00 00 28 B1 49 45 ..PK.....IE
002968A8 89 29 73 21 8C 00 00 00 A5 00 00 00 20 00 00 00 il)st!...N...
002968B8 61 64 34 65 31 36 30 35 61 63 38 33 65 35 36 34 ad4e1605ac83e564
002968C8 31 34 37 33 39 38 63 39 30 37 65 31 37 37 38 65 147398c907e1778e
002968D8 4D 8C 31 0B 83 30 10 46 F7 4B FE C3 8D ED 90 43 M11a30fF0aTi0eC
002968E8 69 EB E0 66 9D 3A 08 05 A1 FB 35 3D 31 45 CF E0 i3xf=:iJ5=1E=
002968F8 45 A8 FF BE E9 06 E9 83 F7 3D 5E 47 12 06 D6 E4 E2=8p03=-^G+m2
00296908 1E BC 6A 58 A4 86 12 0B 6B 1A F9 23 4D 24 3F 32 ^jXk3+ok+.#M$?2
00296918 64 96 CF 0A 2F D6 B4 2B 53 E2 97 BB EE D9 AF F0 d0=.m!+Sf0je>=
00296928 E4 9E 45 09 87 7E 13 E8 82 5F 17 DD 35 F1 AC 70 2AE.c"!3e S+kp
00296938 13 8F 47 6B 3A 0A E2 DA 89 54 6B 10 4E A8 03 4E !!AGk:.r reTkN
00296948 B4 89 1F CF 6F E4 0F CD 71 62 EC C3 6F 9A 18 73 -e=c2=qbwtoft
00296958 FC 67 BA 38 A5 B1 06 04 6B AC F9 02 50 4B 03 04 "gll;N+ok%OPK
00296968 0A 00 00 00 00 00 2C B1 49 45 00 00 00 00 00 00 .....IE.....
00296978 00 00 00 00 00 00 20 00 00 00 62 62 64 35 65 34 .....bbd5e4
00296988 66 33 64 62 34 66 30 37 32 34 38 63 32 39 65 f3db4f07248c2b9e
00296998 31 30 34 62 38 65 34 38 38 39 50 4B 03 04 0A 00 104b8e4889PK...
002969A8 00 00 00 2C B1 49 45 00 00 00 00 00 00 00 00 .....IE.....
002969B8 00 00 00 20 00 00 39 64 61 62 35 39 35 35 .....9dab5955
002969C8 63 36 66 33 34 63 66 63 34 38 30 32 62 38 32 31 c6f34cfc4802b821
002969D8 32 64 39 66 61 31 38 36 50 4B 03 04 0A 00 00 00 2d9fa186PK.....
002969E8 00 00 2C B1 49 45 00 00 00 00 00 00 00 00 00 00 ...IE.....
002969F8 00 00 20 00 00 00 36 61 61 34 32 62 37 61 63 66 .....6aa42b7acf
00296A08 35 39 64 63 38 66 61 37 34 32 38 31 33 63 39 31 59dc8fa742813c91
00296A18 61 38 66 37 38 65 50 4B 03 04 0A 00 00 00 00 00 a8f78ePK.....
00296A28 2C B1 49 45 00 00 00 00 00 00 00 00 00 00 00 00 ...IE.....
00296A38 20 00 00 64 32 39 31 66 30 32 63 38 32 33 35 ....d291f02c8235

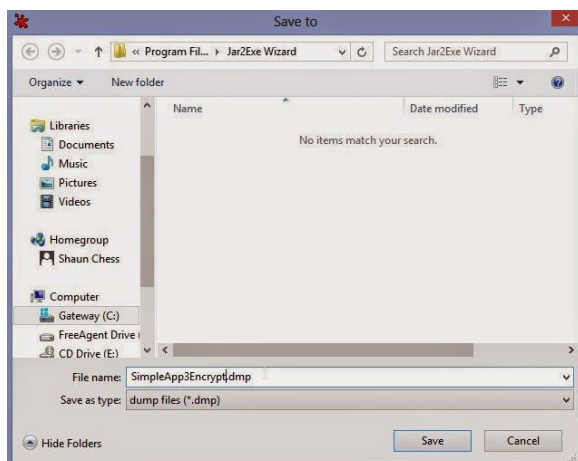
```

As you can see here, the names of our classes and manifest files have been completely obfuscated. As a result, we need to recover these names manually. The way that jar2exe gets the original names for the class is to read them directly from the class file itself. I will show you a simple approach to doing this once we dump the archive.

Now since I am finished digressing, let's dump the file just like before.



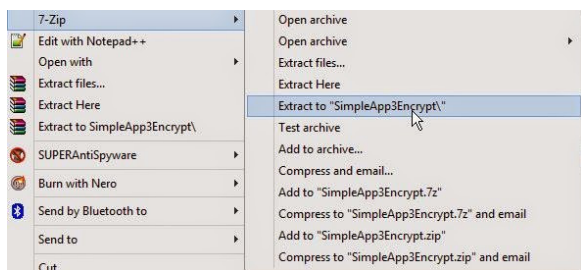
In my case, the location of the decrypted file is at 296858 and its length is 1284. We are ready to dump:



Now that we have saved the file, let's rename change the extension to .jar and prepare to extract the contents.

SimpleApp3Encrypt.jar

Please note that due to the encryption employed on the names, the file will not run until we correct the classes. Let's extract the contents to a new folder so we can begin repairing.

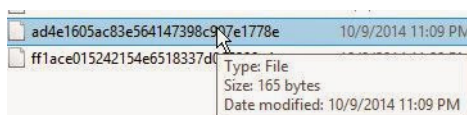


Once we have extracted the files, let's open the new directory and see the

files.

6aa42b7acf59dc8fa742813c91a8f78e	10/9/2014 11:09 PM	File folder	
9dab5955c6f34cfc4802b8212d9fa186	10/9/2014 11:09 PM	File folder	
53d2772aef2cb45e45edc2251942137b	10/9/2014 11:09 PM	File folder	
bbd5e4f3db4f07248c2b9e104b8e4889	10/9/2014 11:09 PM	File folder	
d291f02c823593b23fe9033e4e398c61	10/9/2014 11:09 PM	File folder	
88c93d66586f4715cc1491a426638f3d	10/9/2014 11:09 PM	File	4 KB
781760a97de81d54d73c88f0944e9966	10/9/2014 11:09 PM	File	1 KB
aa03f53f365cfab187a365fa8f2b0c3c	10/9/2014 11:09 PM	File	1 KB
ad4e1605ac83e564147398c907e1778e	10/9/2014 11:09 PM	File	1 KB
ff1ace015242154e6518337d065288c4	10/9/2014 11:09 PM	File	1 KB

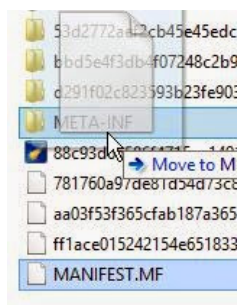
As we can see here, the file names are complete gibberish. The randomly named directories will typically not contain a file and can be deleted. The rest of the files will be the original class and manifest files. To begin the recovery, I like to first identify which file is the manifest. We can usually assume that the manifest will be the smallest file contained in the archive. The manifest.mf file tells the java runtime which file contains the main() class. Without it, the runtime would not know how to execute the code. Let's find the smallest file and open it in a text editor.



```

1 Manifest-Version: 1.0
2 Ant-Version: Apache Ant 1.6.5
3 Created-By: 16.3-b01 (Sun Microsystems Inc.)
4 Main-Class: net.sf.launch4j.example.SimpleApp
5 Class-Path: .
  
```

As you can see, we have identified our manifest file. This file should be renamed to 'MANIFEST.MF' and placed in a subdirectory called 'META-INF'. Remember that these names are case sensitive and must be typed in all uppercase.



To continue with the rest of the files, let's add a '.class' extension to each of them and open them in your favorite java decompiler. You can read these names with a hex editor if you fancy, but a decompiler makes it a little cleaner. Let's begin with the largest file which is 4kb. I will open it in DJ Java Decompiler.



```
package net.sf.launch4j.example;  
  
import java.awt.Dimension;  
import java.awt.Toolkit;  
import java.io.PrintStream;  
import javax.swing.*;  
  
public class SimpleApp extends JFrame
```


As we can see here, we have identified the name of this class file as SimpleApp which according to the manifest, contains our main() procedure. The name of

the package this file resides in is 'net.sf.launch4j.example'. The dots between these words represent a subdirectory. That means that this class belongs in a folder hierarchy of net/sf/launch4j/example. Let's rename this to class file to 'SimpleApp.class', create the directory hierarchy listed above, and place this file in it.



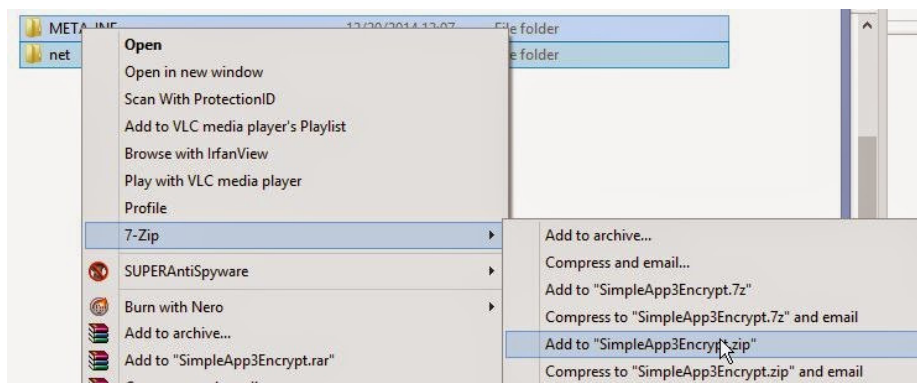
Name	Date modified	Type	Size
SimpleApp.class	10/9/2014 11:09 PM	CLASS File	4 KB

Now that we have completed this, we just need to go back and follow the same steps for the rest of the files. Please remember that every name is case sensitive. Once you have recovered each of them and placed them into the correct directory (there is only one directory), it should look like this:

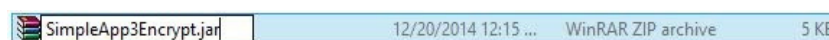


SimpleApp\$1.class	10/9/2014 11:09 PM	CLASS File	1 KB
SimpleApp\$2.class	10/9/2014 11:09 PM	CLASS File	1 KB
SimpleApp\$3.class	10/9/2014 11:09 PM	CLASS File	1 KB
SimpleApp.class	10/9/2014 11:09 PM	CLASS File	4 KB

Now that we are done, it is time to recreate the jar file. A jar file is simply a zip archive under a different extension. All we need to do is go back to the directory that contains our two folders named 'net' and 'META-INF' and add them to a zip archive.

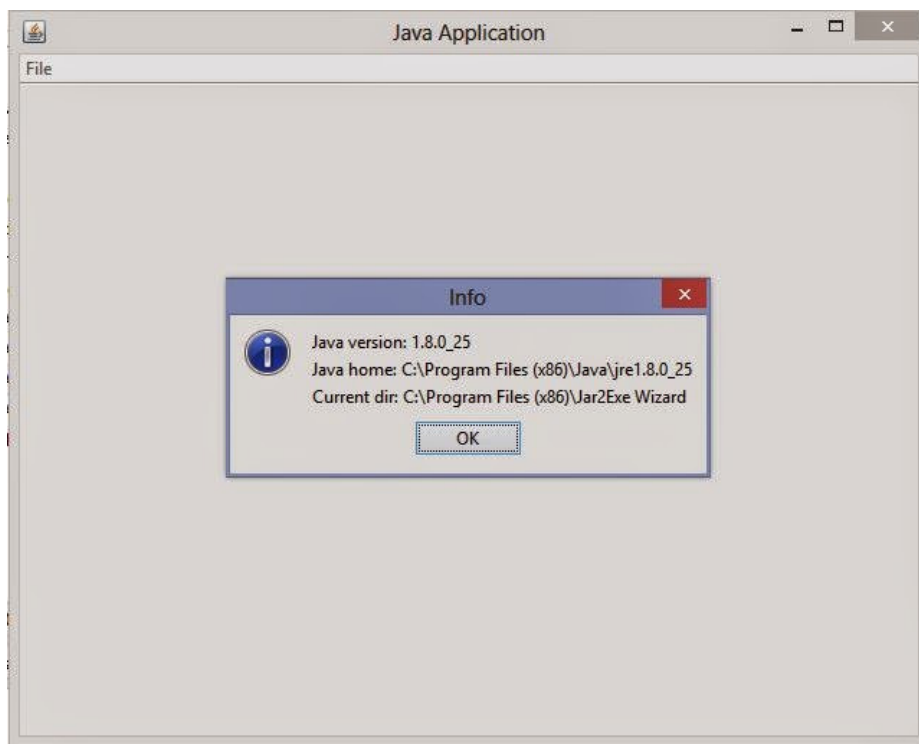


Now, we simply change the extension of the zip archive to jar and it is ready to run.



SimpleApp3Encrypt.jar	12/20/2014 12:15 ...	WinRAR ZIP archive	5 KB
-----------------------	----------------------	--------------------	------

Now let's see the finished result:



I hope that you have learned a few things from this tutorial and have walked away with some new knowledge. If something in this tutorial is not clear, please feel free to post a comment and I will try to explain it further. Until next time, happy reversing.

Posted by Chester Fritz at 1:15 PM

Labels: [crypto](#), [jar 2 exe](#), [jar to exe](#), [Jar2Exe](#), [Jar2Exe 2.1](#), [java](#), [re](#), [reverse engineering](#), [unpack](#), [unpack jar2exe](#), [unpacking](#), [unpacking Jar2Exe](#)

### 3 comments:



**hookahice** January 4, 2015 at 5:19 PM

IS there any way for you to make a tutorial about extracting JAR from exe used by Avian (<http://oss.readytalk.com/avian/>). There is an app called "Best PHP Obfuscator" that uses this and I would appreciate help with extracting the JAR to look at some decompiled code.

Thanks mate!

Reply



**Ret Nop** January 7, 2015 at 3:43 PM

nice tut guy! hope it work for me :)

Reply



**Nleo - Hacker** January 11, 2015 at 11:19 AM

Truly Aswm..  
Keep Sharing.... :)

Br,  
Team-URET

Reply

Enter your comment...

**Comment as:**

**Publish**

**Preview**

[Newer Post](#)

[Home](#)

[Older Post](#)

[Subscribe to: Post Comments \(Atom\)](#)

---

#### sponsors

---

Awesome Inc. template. Powered by Blogger.