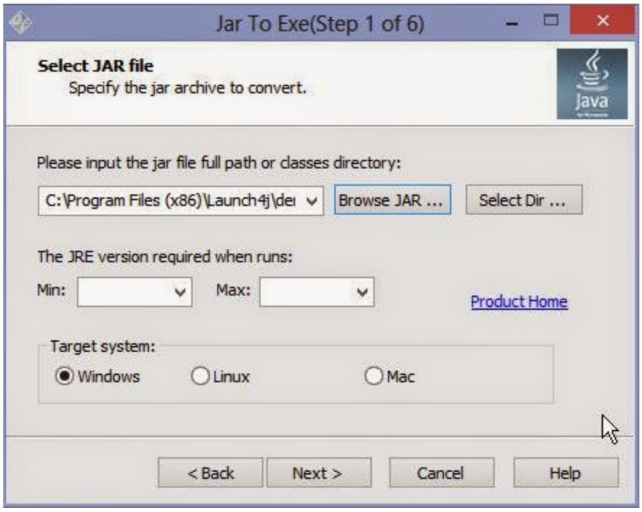
Reverse Engineering Tips

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

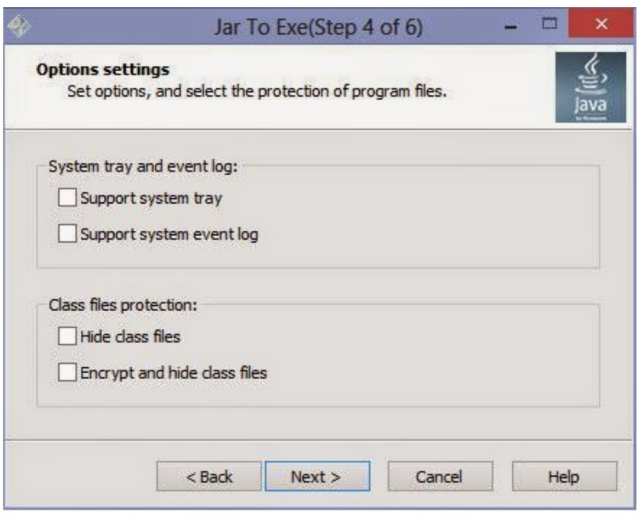


Welcome to this extensive tutorial for unpacking Jar2Exe. Jar2Exe is a java executable wrapper which works by taking your original java archive, wrapping it into an executable, and executing it through a virtual environment using the jvm.dll provided with each java distribution. It also provides the 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
Ollydbg 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

Search

Donations

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

Donate

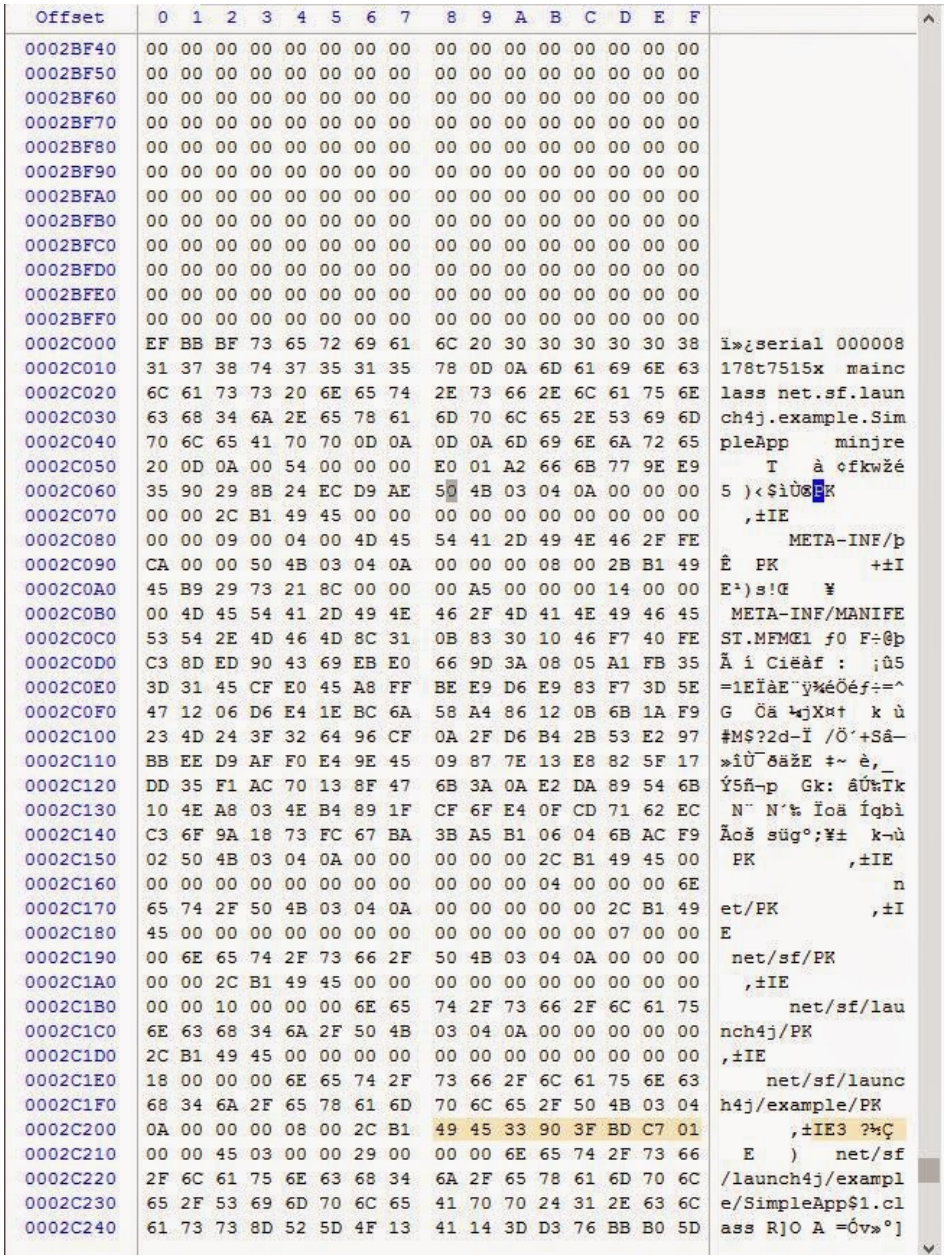
About Me

Chester Fritz

View my complete profile

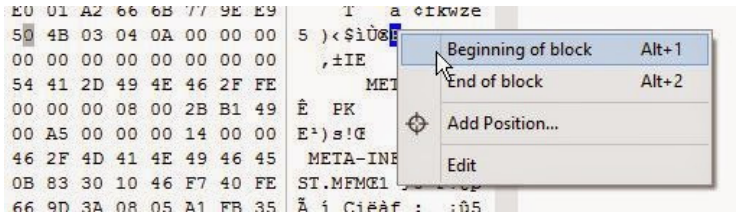
Blog Archive

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.

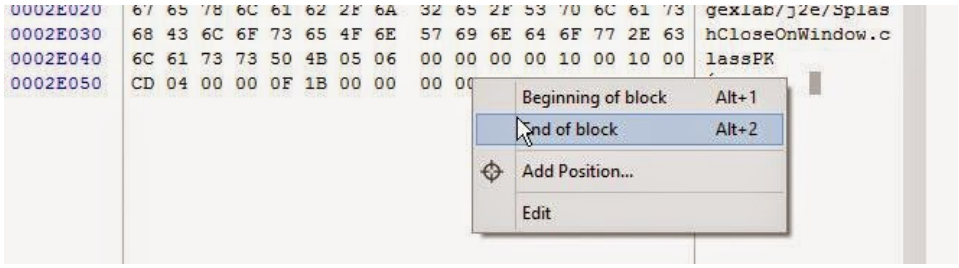


Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0002BF40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002BF50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002BF60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002BF70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0002BF80	00	00	00	00	00	00	00	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	ï»¿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 ¢fkwžé
0002C060	35	90	29	8B	24	EC	D9	AE	50	4B	03	04	0A	00	00	00	5)‹$ìÙ®PK
0002C070	00	00	2C	B1	49	45	00	00	00	00	00	00	00	00	00	00	,±IE
0002C080	00	00	09	00	04	00	4D	45	54	41	2D	49	4E	46	2F	FE	META-INF/þ
0002C090	CA	00	00	50	4B	03	04	0A	00	00	00	08	00	2B	B1	49	Ê PK +±I
0002C0A0	45	B9	29	73	21	8C	00	00	00	A5	00	00	00	14	00	00	E¹)s!Œ ¥
0002C0B0	00	4D	45	54	41	2D	49	4E	46	2F	4D	41	4E	49	46	45	META-INF/MANIFE
0002C0C0	53	54	2E	4D	46	4D	8C	31	0B	83	30	10	46	F7	40	FE	ST.MFMŒ1 ƒ0 F÷@þ
0002C0D0	C3	8D	ED	90	43	69	EB	E0	66	9D	3A	08	05	A1	FB	35	Ã í Cië à f : ¡û5
0002C0E0	3D	31	45	CF	E0	45	A8	FF	BE	E9	D6	E9	83	F7	3D	5E	=1EÏàE¨ÿ¾éÖéƒ÷=^
0002C0F0	47	12	06	D6	E4	1E	BC	6A	58	A4	86	12	0B	6B	1A	F9	G Öä ¼jX¤† k ù
0002C100	23	4D	24	3F	32	64	96	CF	0A	2F	D6	B4	2B	53	E2	97	#M$?2d–Ï /Ö´+Sâ—
0002C110	BB	EE	D9	AF	F0	E4	9E	45	09	87	7E	13	E8	82	5F	17	»îÙ¯ðäžE ‡~ è‚_
0002C120	DD	35	F1	AC	70	13	8F	47	6B	3A	0A	E2	DA	89	54	6B	Ý5ñ¬p Gk: âÚ‰Tk
0002C130	10	4E	A8	03	4E	B4	89	1F	CF	6F	E4	0F	CD	71	62	EC	N¨ N´‰ Ïoä Íqbì
0002C140	C3	6F	9A	18	73	FC	67	BA	3B	A5	B1	06	04	6B	AC	F9	Ãoš süg°;¥± 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	6E	n
0002C170	65	74	2F	50	4B	03	04	0A	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	07	00	00	E
0002C190	00	6E	65	74	2F	73	66	2F	50	4B	03	04	0A	00	00	00	net/sf/PK
0002C1A0	00	00	2C	B1	49	45	00	00	00	00	00	00	00	00	00	00	,±IE
0002C1B0	00	00	10	00	00	00	6E	65	74	2F	73	66	2F	6C	61	75	net/sf/lau
0002C1C0	6E	63	68	34	6A	2F	50	4B	03	04	0A	00	00	00	00	00	nch4j/PK
0002C1D0	2C	B1	49	45	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	net/sf/launc
0002C1F0	68	34	6A	2F	65	78	61	6D	70	6C	65	2F	50	4B	03	04	h4j/example/PK
0002C200	0A	00	00	00	08	00	2C	B1	49	45	33	90	3F	BD	C7	01	,±IE3 ?½Ç
0002C210	00	00	45	03	00	00	29	00	00	00	6E	65	74	2F	73	66	E) net/sf
0002C220	2F	6C	61	75	6E	63	68	34	6A	2F	65	78	61	6D	70	6C	/launch4j/exampl
0002C230	65	2F	53	69	6D	70	6C	65	41	70	70	24	31	2E	63	6C	e/SimpleApp$1.cl
0002C240	61	73	73	8D	52	5D	4F	13	41	14	3D	D3	76	BB	B0	5D	ass R]O A =Óv»°]

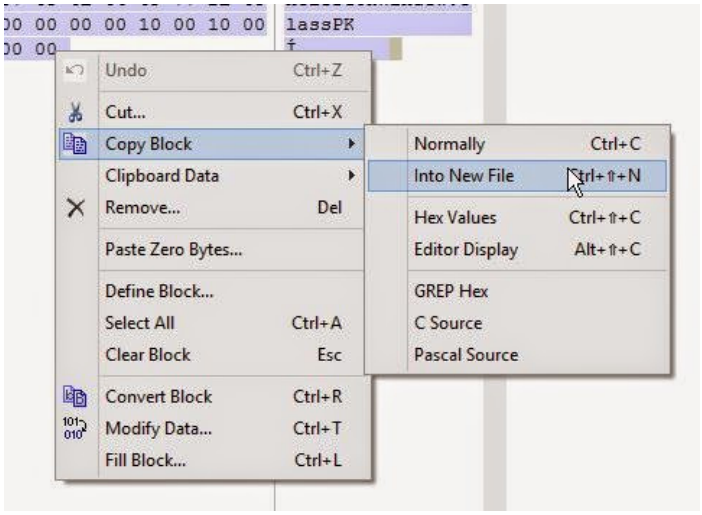
Once you are here, the first occurrence of 'PK' labels the start of our archive. We can label this as our beginning of block:



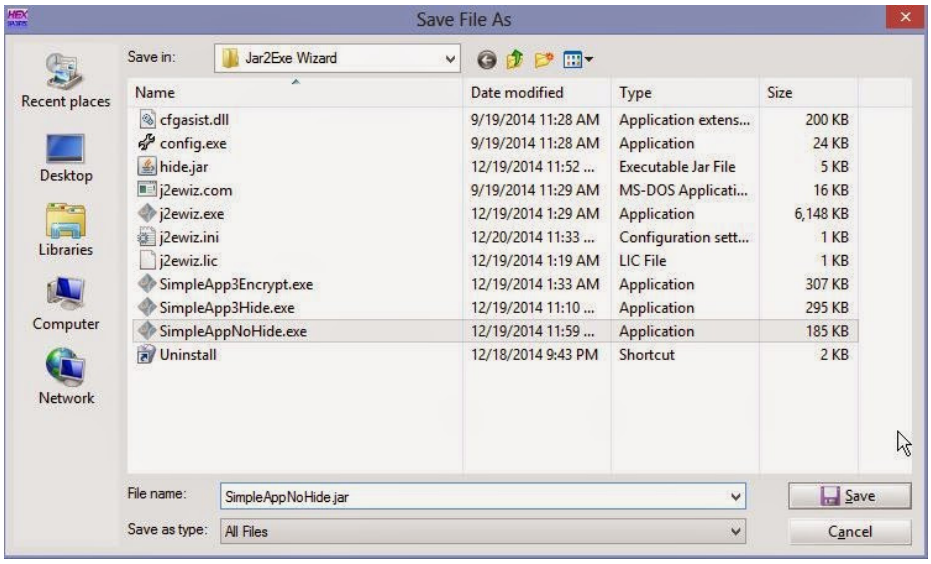
Afterwards, we can scroll to the very last byte in the file and label it as our End of Block:



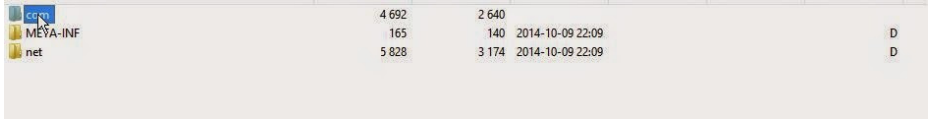
Once you have done so, right click the selected block and click Edit. In the new pane, go to Copy Block -> Into New File.



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

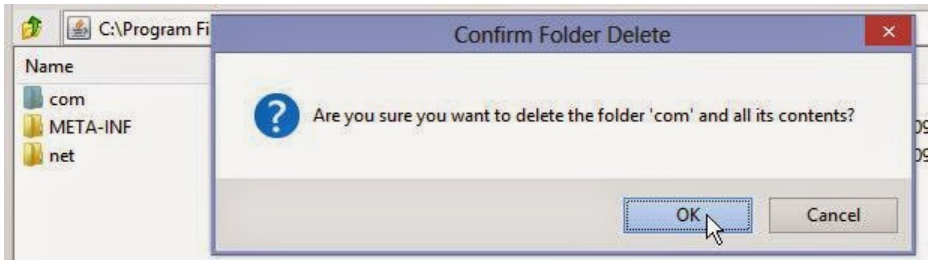


Once we are finished, the program will run correctly, but to be tidy, we need to delete the extra files that Jar2Exe added to our archive. Let's open the archive in 7-zip or winrar(whichever you fancy).

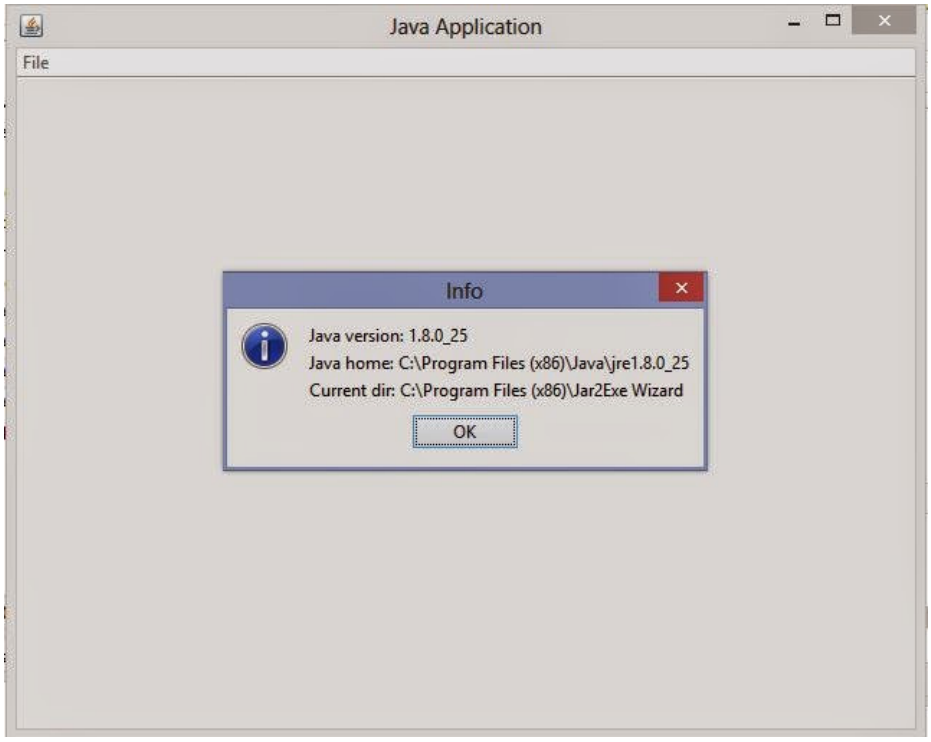


Once it is opened, you can see that Jar2Exe adds an additional directory called 'com'. While some other java applications may use this directory, Jar2Exe adds an additional subdirectory called regexlab.

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:

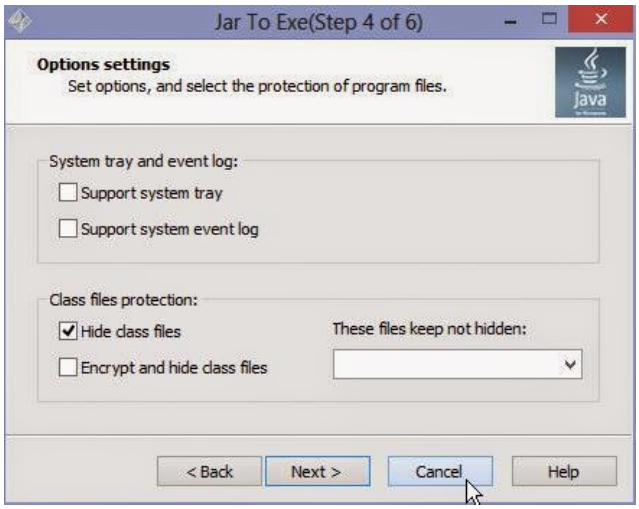


After confirming the deletion, we can close the archive and run the jar file.

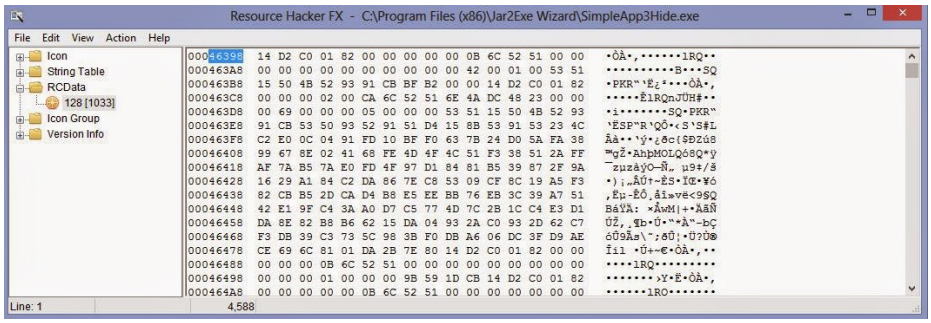


If you did everything correctly, the application should run without problems and our work is finished.

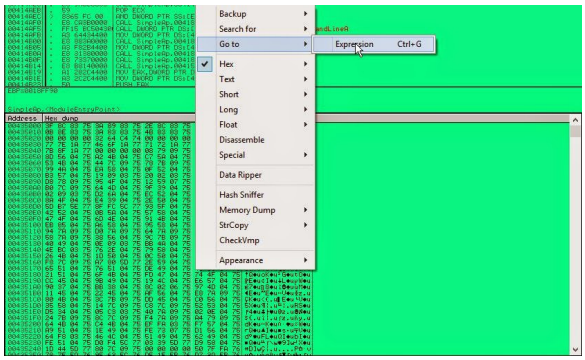
Jar2Exe Level 2: Hidden Archive:



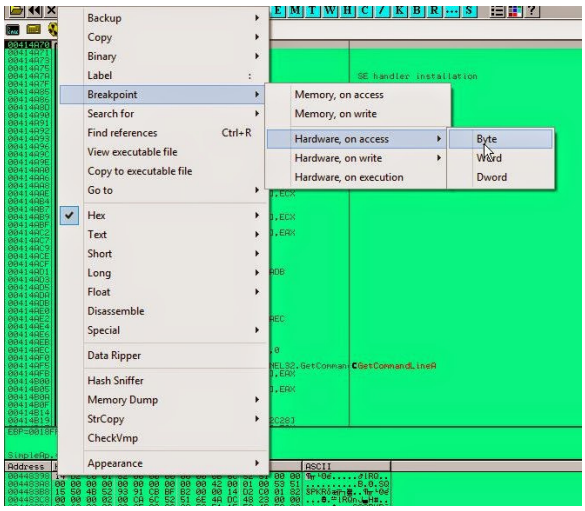
With the level 2 protection, Jar2Exe takes our java archive, encrypts it, and adds it as an RCData entry in the resource directory. To find the offset of the encrypted archive, we need to open our executable which I named 'SimpleAppHide.exe' in a resource editor. For this, I prefer resource hacker:



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:

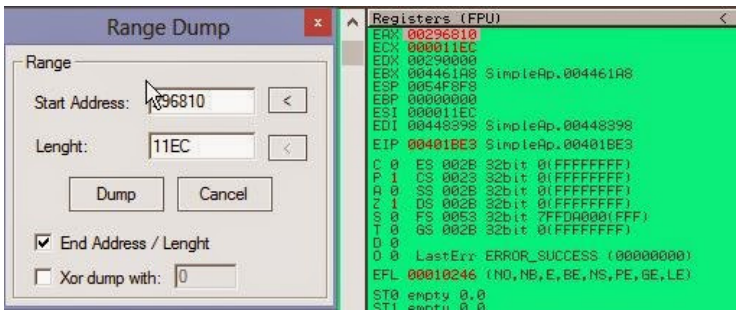


Once you arrive at the offset, right click the byte and add a Hardware Breakpoint On Access -> Byte:

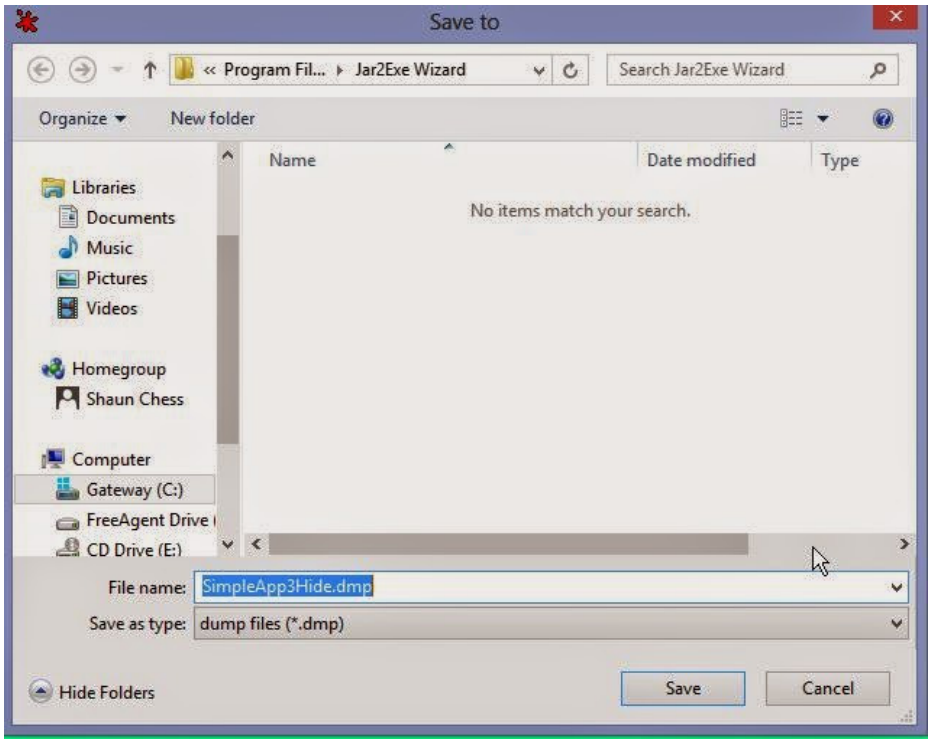


Now, let's run the program and wait for the break to occur. We should arrive here:

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 DWOR 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:

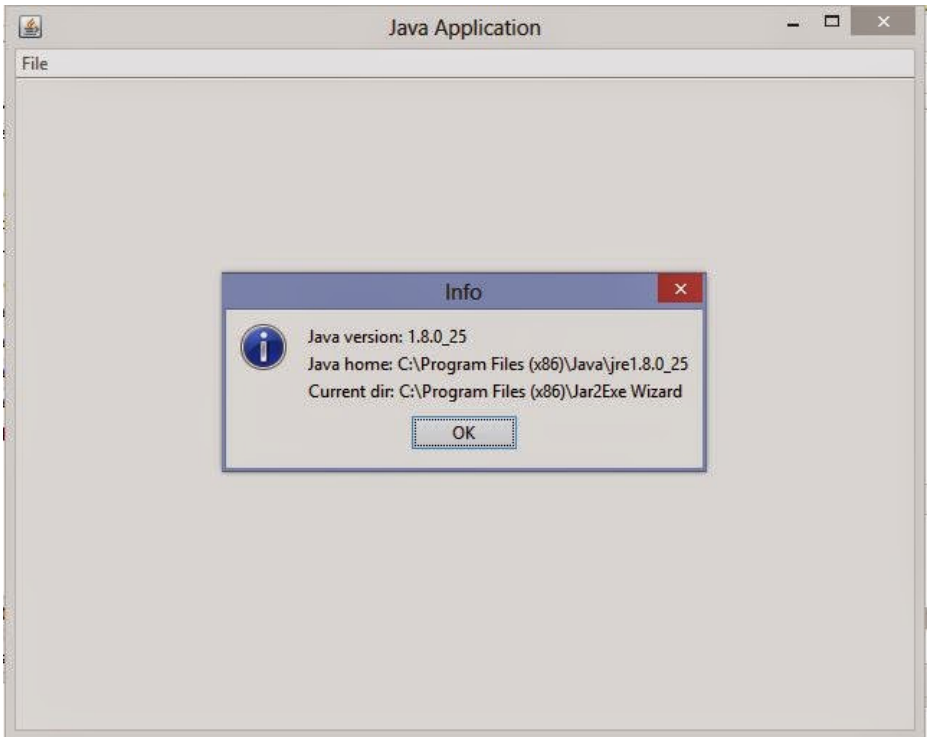


To begin, check the field which says End Address / Lenght. This will allow us to enter the length value. Set the Start Address to the value in EAX and the lenght(length) equal to the value in ECX. In my case, the address is 296810 and the size is 11EC. Once you have entered these values, we are ready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

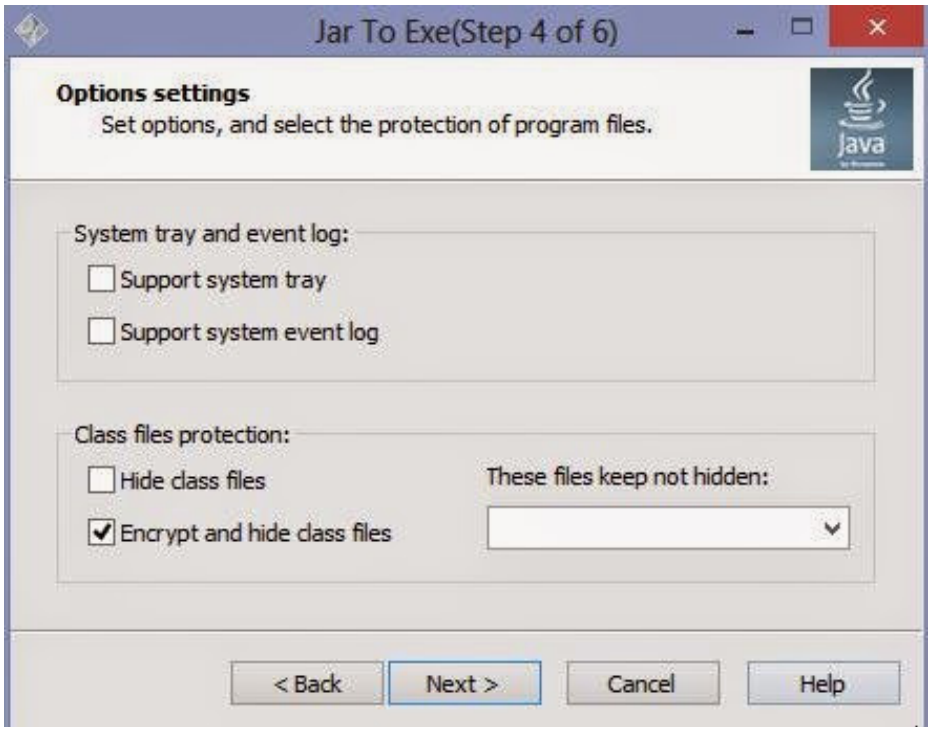


A limitation of this tool is that you can only save the file with a .dmp extension, but we can easily rename it once we are finished. Simply save it with the name you fancy. In this case I named it as SimpleApp3Hide.dmp for easy recognition.

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

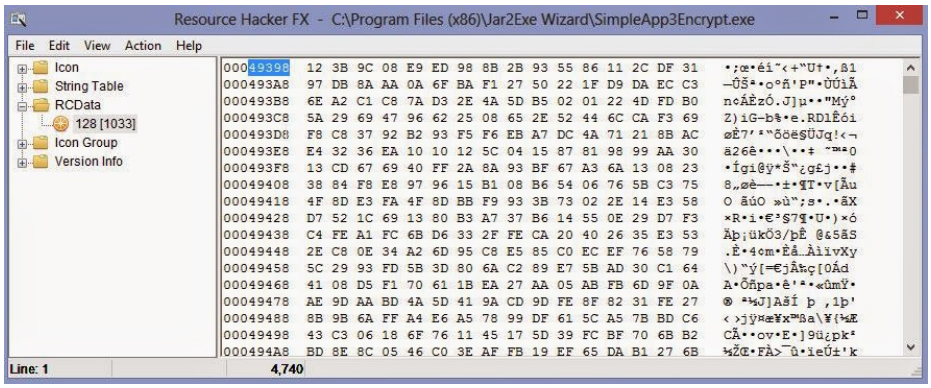


Jar2Exe Level 3: Hidden Archive + Encrypted Class Names:

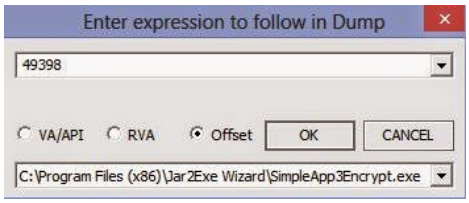


This is the strongest protection offered by Jar2Exe. First, it changes all of the class file names to gibberish. Second, it encrypts and hides the files just like the level 2 protection. While we can recover the archive with a similar method as in Level 2, we need to take an additional step to repair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

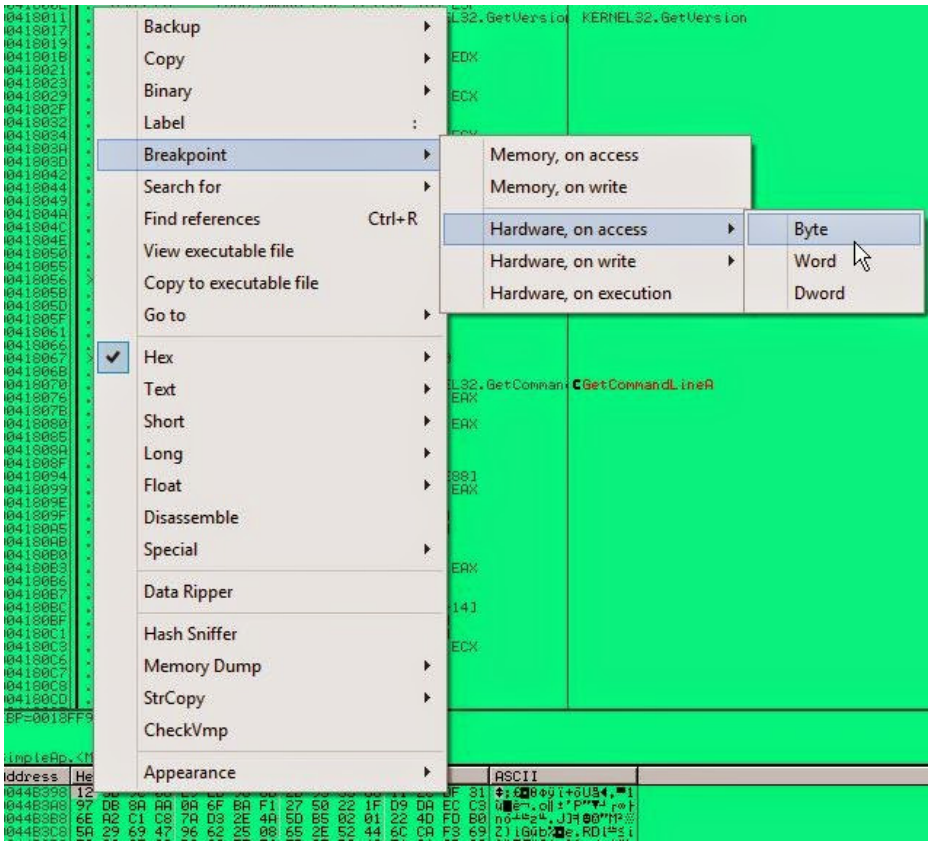
To begin, let's open the application "SimpleAppEncrypt.exe" in a resource editor just as before to acquire the offset of the encrypted RCData:



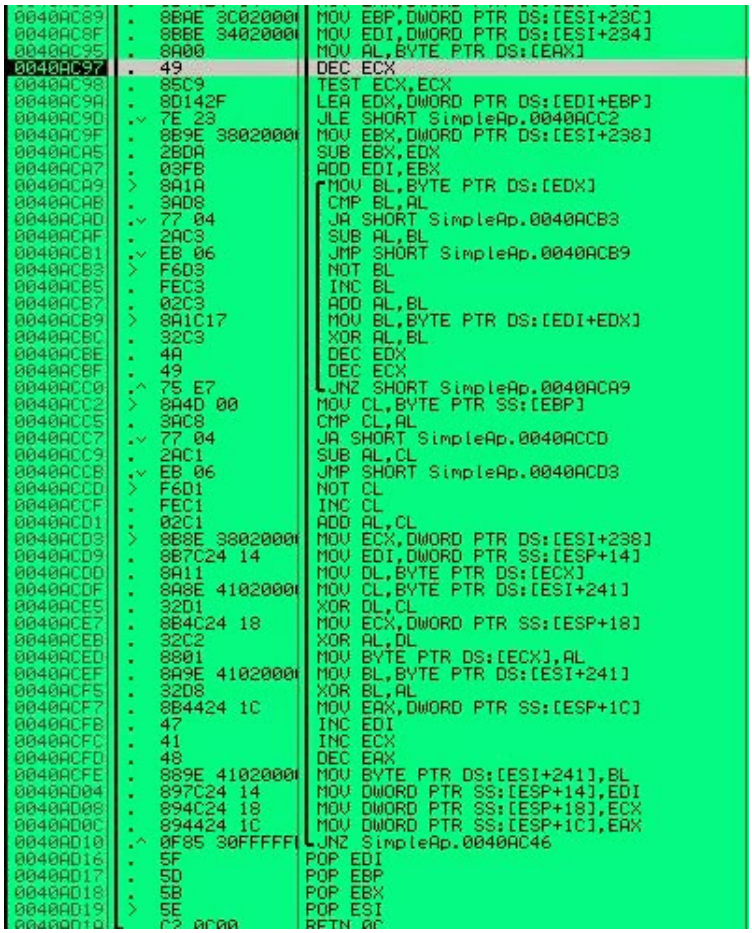
Our offset in this case is 49398h. Let's note that and open the application in Ollydbg and follow the offset.



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



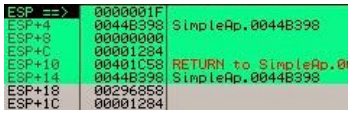
Run the program an wait for the breakpoint. We should arrive at the following code:



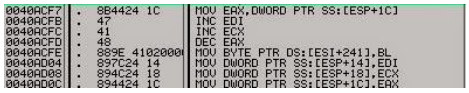
This is the new

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

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].

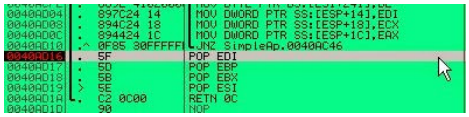


The value at ESP+1C contains the size of the memory region. This can be verified by checking the decrement value of the loop.

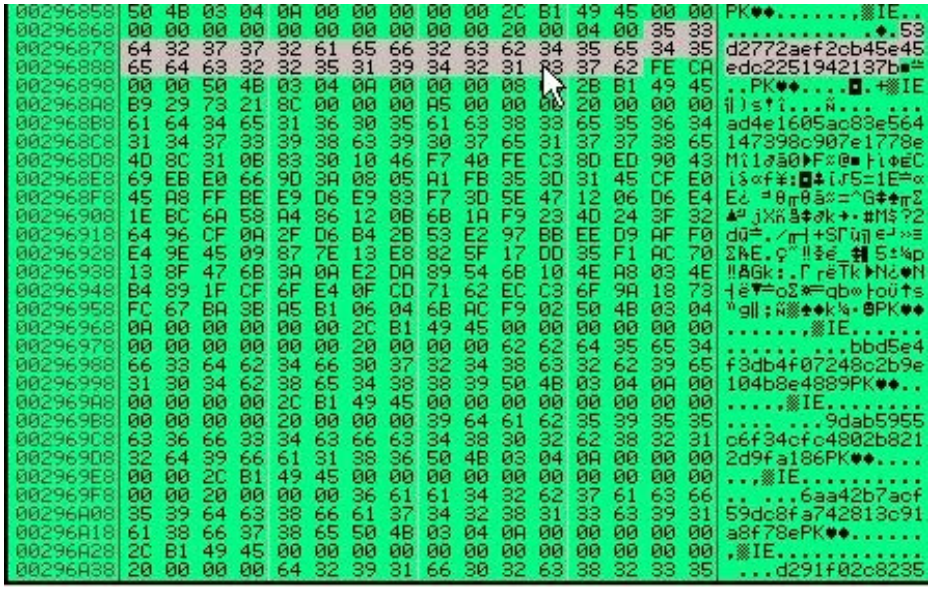


This code verifies that ESP+1C is the value decremented. ESP+1c is moved to EAX, decremented, and then moved back.

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.

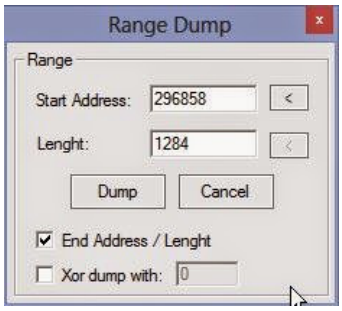


Once you break here, we are ready to make a dump of the file. But first, I would like to show you the encrypted strings in our archive to give you a glimpse of how jar2exe encrypts them:

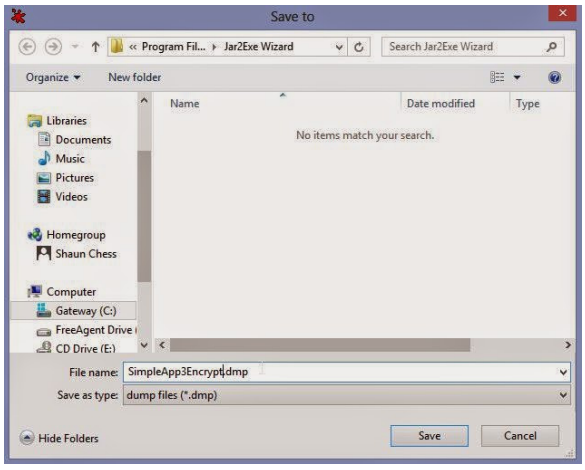


As you can see here, the names of our classes and manifest files have been completely obfuscated. As a result, we need to recover these names manually. The way that jar2exe gets the original names for the class is to read them directly from the class file itself. I will show you a simple approach to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

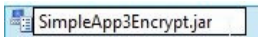
Now since I am finished digressing, let's dump the file just like before.



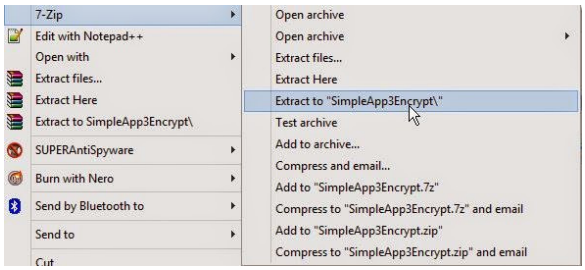
In my case, the location of the decrypted file is at 296858 and its length is 1284. We are ready to dump:



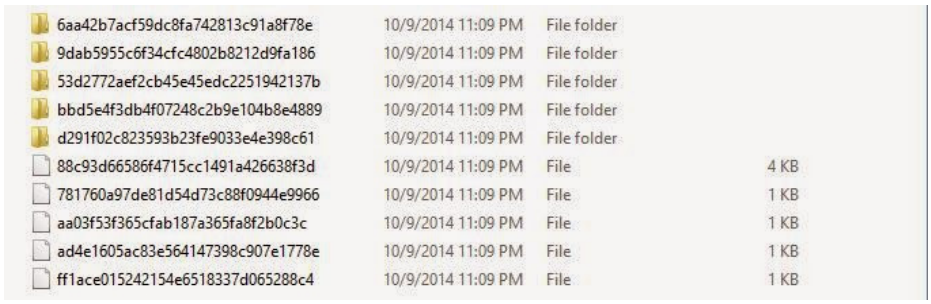
Now that we have saved the file, let's rename change the extension to .jar and prepare to extract the contents.



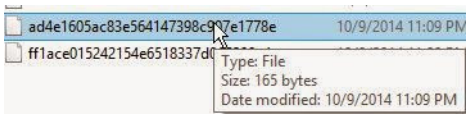
Please note that due to the encryption employed on the names, the file will not run until we correct the classes. Let's extract the contents to a new folder so we can begin repairing.

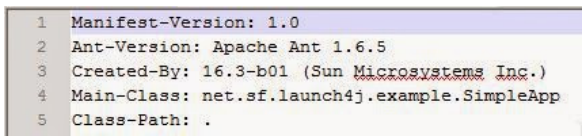


Once we have extracted the files, let's open the new directory and see the files.



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.





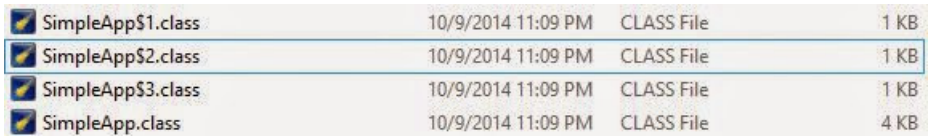
As you can see, we have identified our manifest file. This file should be renamed to 'MANIFEST.MF' and placed in a subdirectory called 'META-INF'. Remember that these names are case sensitive and must be typed in all uppercase.

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.

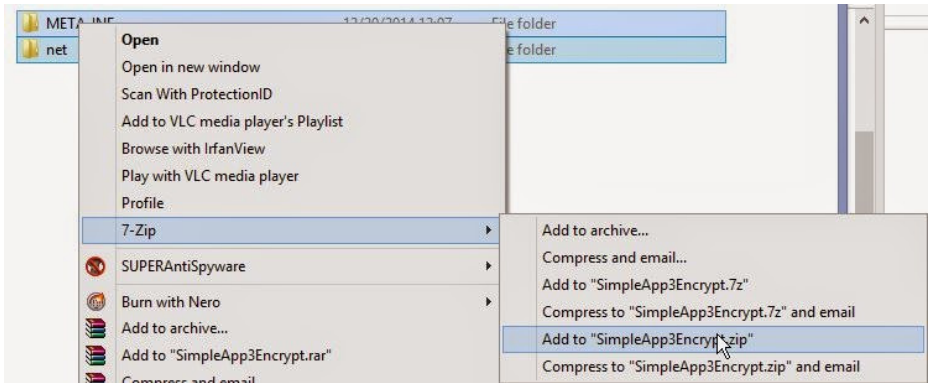
```
package net.sf.launch4j.example;

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.

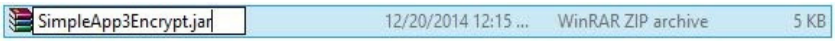
Now that we have completed this, we just need to go back and follow the same steps for the rest of the files. Please remember that every name is case sensitive. Once you have recovered each of them and placed them into the correct directory(there is only one directory), it should look like this:



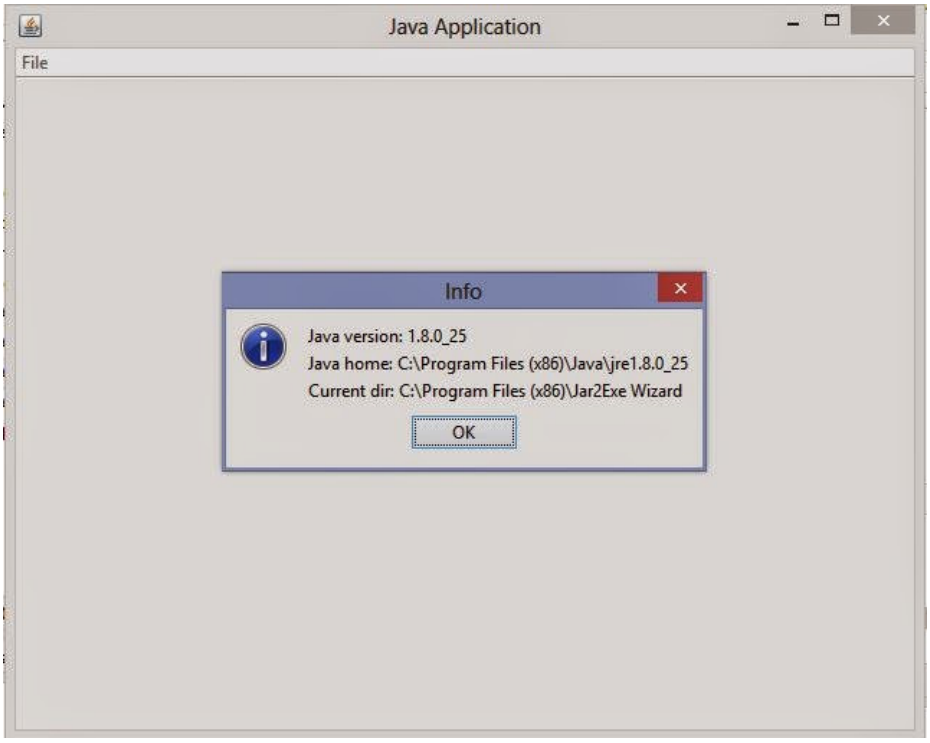
Now that we are done, it is time to recreate the jar file. A jar file is simply a zip archive under a different extension. All we need to do is go back to the directory that contains our two folders named 'net' and 'META-INF' and add them to a zip archive.



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



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 :)

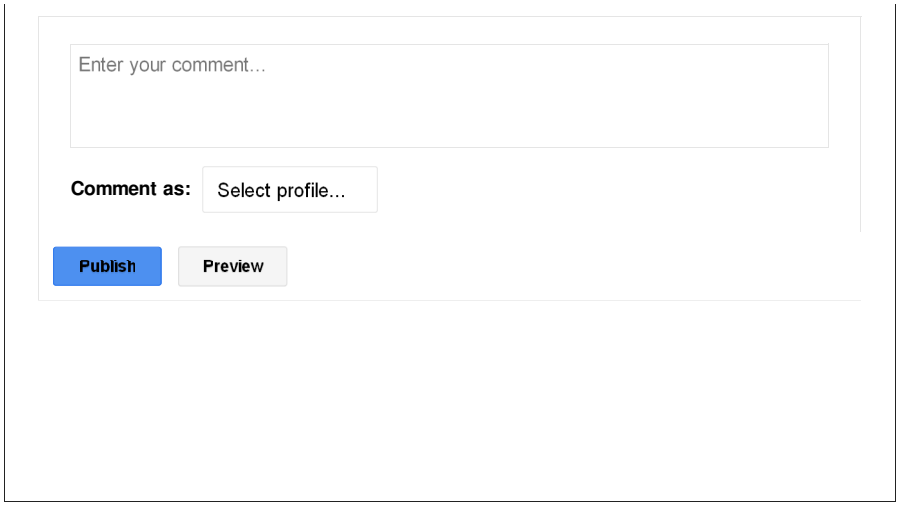
Reply

Nieo - Hacker January 11, 2015 at 11:19 AM

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

Br,
Team-URET

Reply



Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

sponsors