MALWARE ANALYSIS 3

HANDS IN THE COOKIE JAR

Peter Ferrie
Microsoft, USA

Viruses for Java are relatively rare, and parasitic viruses for Java are even rarer. There are two ways in which a virus can infect a Java application. One is to attach to a class file, which might involve decompiling the class file to its individual components, inserting the virus code, and then reassembling the result. This technique is extremely error-prone but is known to be possible since it has been used successfully by the StrangeBrew [1] and BeanHive [2] viruses (and more recently on the very Java-like .NET platform [3]). The other method is to place the virus code inside a JAR file and include a reference to the virus class file, for example by changing the application's entry point. This is the technique that is used by the Java/Handjar virus.

SPLIT THE DIFFERENCE

The virus begins by retrieving the name of the directory from which the JAR file was run. The temporary files that are created during the infection phase will be placed in this directory.

The virus also retrieves the path of the executing class file, and splits the string according to the operating-system-specific separator ('\' for *Windows* and '/' for everything else). This is effectively the third line of code and it is here that we encounter the first bug. It is clear that the code was built and tested on a non-*Windows* system, because on *Windows*, the regular expression that is passed to the split function is malformed. This causes an exception to be raised and the virus code exits noisily.

JAR HEAD

If the path has been split successfully, then the virus isolates the last component of the path, which is the filename of the executing JAR file. The virus creates a list of .jar files that exist in the current directory. It is rare for a JAR file to be named '.JAR' or '.Jar', for example, instead of '.jar'. However, the comparison is case-sensitive and such files will be ignored if they exist. The virus attempts to parse the running JAR file to extract its contents. If the 'mytmpdir' directory does not exist, then the virus attempts to create it. This is the directory that will hold the files from an infected JAR file – the 'infected' directory.

The virus queries the list of entries in the JAR file. For each of them, it retrieves the name of the entry and then constructs a path that includes the name of the entry. The virus extracts the filename by isolating everything that follows the last '/' character, but then does nothing with it. The virus also extracts the name of the new subdirectory by isolating everything up to the last '/' character, if it exists. If the entry is a subdirectory, the virus appends the subdirectory name to the path. There is an interesting 'bug' in the code at this point – the variable that receives the path is declared once and assigned twice:

```
File dir = dir = ...
```

This is harmless, of course, because the assignment of the variable to itself does not change the result – it just demonstrates the carelessness of the virus author.

The virus creates the required subdirectories (there might be more than one) if they do not exist already. If the entry refers to a file, then the virus decompresses it in 16KB chunks and writes the result as a file in the appropriate subdirectory. After all of the entries have been processed, the virus constructs a list of the extracted files. This results in a problem during the infection phase, which is described below.

The virus examines each of the JAR files that it has found, excluding the one that is running already. If the 'tmpdir' directory does not exist, then the virus attempts to create it. This is the directory that will hold the contents from the other JAR files – the 'clean' directory. For each of the JAR files, the virus extracts the contents into the 'clean' directory. The virus checks for the presence of a file named 'kjlfaojdfaljgsdfaKdlkAUSfdld'. This is the infection marker. It is not known whether the string has any significance. For some reason, the virus checks for the presence of the file by iterating over the entire contents of the directory while attempting to match the name, instead of simply checking for the existence of the file.

BUGBLATTER BEAST

If the JAR file is not infected, the virus reads the 'manifest.mf' file and attempts to split the lines according to the operating-system-specific line separator. This is a serious bug because the line separator for the current operating system might not match the line separator that was used when the manifest file was created. If there is a mismatch whereby the operating-system-specific line separator is carriage return/line feed, but the manifest file line separator is only line feeds, then the code will cause an exception to be raised and the virus code will exit noisily. If there is a mismatch whereby the operating-system-specific line separator is line feed only, but the manifest file line separator is carriage return/line

feeds, then a different bug arises (which is described below). If the separation is 'successful', then the virus assumes that the second line will contain the 'Main-Class:' string, and isolates the text that follows the string. This is the application's entry point.

The virus constructs another string that has the same contents as the text following the 'Main-Class:' string, but excluding the last character. The likely reason for excluding the last character is that the virus author had a manifest file that used carriage return/line feed as the line separator. As we know already, the virus was built on a non-*Windows* system, which means that the operating-system-specific line separator was line feed only. When separating the lines by dropping only the line feed, the carriage-return character is retained, resulting in a string that is not usable for the purposes of the virus. The virus author 'solved' this problem by always dropping the last character. However, when the operating-system-specific line separator matches the manifest file line separator, the virus drops the last character of the entry point name. This results in yet another bug, which prevents infected files from running (see below).

LAUNCH IN T-MINUS...

The virus gains access to the installed Java compiler and uses it to compile the virus launcher class directly in memory. The code used to perform the in-memory compilation is taken from a well-known website which hosts the example code. The virus author has not changed any of the code at all, including the variable names, and the result of the compilation is assigned, but never checked.

The launcher class is intended to be the application's new entry point. It is supposed to display a message, run the host code, and then run the virus code. Unfortunately, because the virus includes a multi-line text string in the code to compile, there is a silent compilation failure. That is, the request to compile succeeds, but the output file is not created. It seems likely that the virus writer added the message after the testing was completed, and thus did not notice the problem, because the first generation of the virus inserts its other files into the target JAR file, which makes the target JAR file appear infected. However, the target file will not replicate further because it is missing its entry point class.

MAKE ME A SANDWICH

After compiling the code, the virus alters the manifest file to refer to the virus launcher class. There is *yet* another bug in this code – the virus performs a search-and-replace to change every reference to the original class name, instead of changing the specific one that follows the 'Main-Class:' string. As a result, if the original class name contains a substring of any of the strings that appear in a manifest file ('Manifest-Version: 1.0', 'Created-By:', '(Oracle Corporation)', 'Main-Class:'), then those strings will be corrupted and the target file will not run at all.

The virus places the infection marker in the directory, copies the virus launcher file, and then copies all of the class files from the virus directory to the host directory. Yes, *all* of them. It is not known why the virus does this, because what we have now is a directory containing the contents of the clean JAR file along with the contents of a previously infected JAR file. The next replication of this 'sandwich' will take that collection of files and combine it with the contents of the next JAR file to infect, and so on, resulting in an unbounded size increase. Furthermore, if any of the files from the virus directory have the same name as those in the infected directory, then the files from the virus directory will replace them, resulting in unpredictable behaviour when the application is run.

At this point, the virus copies two of the virus class files explicitly to the infected directory, the third one having been copied by the previous copy operation. The virus deletes the original clean JAR file, and creates a newly infected JAR in its place. The virus deletes the contents of the two directories recursively, along with the two temporary class files that were created during the infection process, and then examines the next file in the list. It exits after all of the files have been processed.

CONCLUSION

The author of this virus managed to take the cross-platform promise of Java and break it to such a point that the virus runs on only a subset of supported platforms, and makes a sandwich that no one would want. That takes skills that no one would envy.

REFERENCES

[1] StrangeBrew. http://www.f-secure.com/v-descs/sbrew.shtml.

[2] Java.BeanHive. http://www.symantec.com/security_response/writeup.jsp?docid=2000-121910-5507-99

[3] Ferrie, P. Let them eat brioche. Virus Bulletin, November 2004, p.6. http://www.virusbtn.com/pdf/magazine/2004/200411.pdf.