Intro. To Reversing - W32Pinkslipbot

blog.opensecurityresearch.com/2011/12/intro-to-reversing-w32pinkslipbot.html

By Michael G. Spohn.

To follow along, download the bundle of source files (http://www.opensecurityresearch.com/files/PinkslipBotJScript.zip) or the individual files will be referenced when first talked about.

Background

A couple of months ago, a colleague sent me a file to analyze. He recovered it from a host compromised by a variant of the W32/Pinkslipbot malware family (aka Qakbot, Akbot, Qbot). The file was found in the \Windows\Temp folder and was scheduled to run every four hours by the Windows scheduler.

Analysis of the file revealed it was not a portable executable (PE) binary. It appeared to be some form of obfuscated script. Whatever it was, I knew it was some form of executable because the scheduler was able to execute it. After a couple of hours of investigative work, I was able to figure out what the script was and how it works.

Even though reverse engineering of malware is considered an art only performed by people with deep knowledge of assembly language, this is simply not true. Sure, analysis of complex malware binaries requires advanced skills. However, in this case, anyone with basic scripting skills and a little tenacity can reverse engineer this script. Let's get started.

Initial Analysis

The original script file appears to be randomly named (kxoe4.zbz), is 6,677 bytes long, and contains 212 lines of script code. (The file listing can be found in Appendix A.) When you look at the listing, you encounter quite a mess. As you examine it closer, you can see the footprints of some form of Java script. The tell-tale signs include the Java keywords 'function', 'var', 'switch', break', and 'case.' There also appears to be C style comments delimited by '/*' and '*/.' Most of the text in the file is obfuscated.

Since the file appears to contain obfuscated Java script, the approach I took to figure out what this script does consists of five steps:

1. Organize (beautify) the code to make it more readable.
2. Identify the script entry point.
3. Determine the de-obfuscation algorithm(s).
4. Decrypt the script contents.
5. Document what the script does.

Obfuscation Algorithm

The first step in our process is to organize the file contents to make it more readable. This is easily done in any text editor. (My favorite editor is Notepad++). There are three code listings you can refer to:

Fire up your editor and load the "pretty", or "commented" versions to make following along easier. Going forward, line numbers refer to contents of Listing_2.txt.

Now that the script is more readable, it is easy to see the script contains four global variables and nine functions. The function names and the arguments are obfuscated. Luckily, we only have nine of them to reverse. The functions are listed below in Table 1.

Table 1 – Script Functions

Script Functions
function svjqxkqL5vb(u45jKF2cnu)
function KiEuqVtyP1A(ERfyC3i)
function s41dk(qqQFVIpO03, yKfNtb)
function o2t84kpEE(AaUg5G)
function tNAaDaVD(S5V8HkvS, tknjLQDR, uQxzc, h162w)
function yU2D2PABJPv(rx1SRpmtJm, UiSt3EtmV)
function zA8QwjCFyA(HlzNb5)
function FiVvczeeGk(QpATGC)
function UK8wMwIhuG(ML2xDT8)

Next, we need to determine the script's entry point. An entry point is the first function that is called when an executable starts. Looking through the beautified script listing, I found an interesting entry in line 271.

```
var ML2xDT8 = new ActiveXObject(s41dk('1fnW+Zjziozk8eDuxg==', 0));
```

This is the only method in the script that is both global in scope and declares a variable to hold the return value, so it is most likely the entry point. The function creates a new ActiveXObject and passes its constructor the return value of function s41dk(). It passes function s41dk() an obfuscated string as the first parameter and a 0 as the second.

Those of you familiar with the Microsoft scripting environment will recognize the ActiveXObject constructor is part of Microsoft's extended Java scripting language JScript. It provides a mechanism for scripts to instantiate and use ActiveX components.

According to MSDN, the AcitveXObject() function prototype is:

```
function ActiveXObject(ProgID : String [, location : String])
```

The first argument takes the form "*serverName.typeName*" where *serverName* is the application that hosts the control; *typeName* is the name of the object to create. The second parameter is optional and contains the name of the network server where the object should be created.

This information tells us the script call to ActiveXObject() via the s41dk() function must be a string in the form *serverName.typeName*. Accordingly, the '1fnW+Zjziozk8eDuxg==' string passed to s41dk() must be in this format after de-obfuscation.

As we examine function s41dk(), on line 50 we see it first creates three variables: *ERfyC3i*, *D9fDoV4rR*, and *UoHfj*. Next, the script determines if some external variable *OfI8GieKmf* exists. If it does not, an array of eleven hex values is created and assigned to variable *TLqV2oczR*. Another variable, *OfI8GieKmf* is created and initialized to a value of 11, most likely to store the length of the array.

Next, the script calls function svjqxkqL5vb() passing the string '81728aamz' as a parameter. This is the very first function at the top of the script. A quick examination of its purpose reveals this function is a ruse that does nothing useful. It returns the length of a string that is never used by any caller. The function is called seven times in the script. This is an example of the length malware writers will go to delay and frustrate anyone who attempts to reverse their code.

Back in the s41dk() function we see it next calls function KiEuqVtyP1A(qqQFVIpO03). Remember, the variable *qqQFVIpO03* is the parameter that contains the obfuscated ActiveXObject information.

Examination of the function KiEuqVtyP1A() on line 12 quickly shows it contains de-obfuscation code. Now we are getting somewhere. It appears this function is an implementation of the Base64 decoding algorithm defined in RFC 4648. How do we know this? The telltale signs are the string containing the alphanumeric character set terminated by the '+' and '/' characters and the fact the processing loop terminates on the '=' character. This function simply accepts and input string and performs the Base64 decoding algorithm on it. It then returns the decoded string.

Returning to the s41dk() function, there is some more de-obfuscation code. Once the decoded Base64 string is returned from the call to KiEuqVtyP1A(), each character in the string is routed through an algorithm (line 65) described by the below pseudo code:

```
hexArr = Array[0x82, 0xaa, 0xb5, 0x8b, 0xf1, 0x83, 0xfe, 0xa2,
0xb7, 0x99, 0x85]
strB64decoded = decoded string from call to KiEuqVtyP1A()
asciiString = fully decoded string
for(int x=0; x< length of strB64decoded; x++)
{
    achar = strB64decoded[x]
    charASC = achar XOR'd with hexArr [x modulus 11]
    append charASC to asciiString
}
return asciiString
```

In other words, each character in the decoded Base64 string is XOR'd with a hex value in the array *hexArr*. The hex value to use is calculated by dividing the zero based index position of the character in the *strB64decoded* string by 11 and using the remainder as the index into *hexArr*.

This type of encoding algorithm is very common in malware. Malware authors prefer to use Base64 encoding since it is a safe and effective way to transmit ASCII and UNICODE data safely across the Internet. An additional encoding algorithm similar to the one described above is often used to encrypt the string before it is Base64 encoded.

Reversing

Now that we know the obfuscation uses a two-step XOR and Base64 algorithm, we need to emulate it so we can decrypt the strings. This is a straightforward task easily accomplished using scripting languages. If you are not a programmer, you probably have at least some experience with a scripting language.

My scripting language of choice is Python. I wrote a Python script (psbot_decode.py) that contains functions that encrypts and decrypts Unicode strings using the same algorithm as the malware script. It can also process a file of encrypted strings to save time. A listing of the script can be found in Appendix C.

Even if you are not familiar with Python, if you spend a few minutes looking at the code you should be able to understand how it works. If Python is not your thing, then I encourage you to port the functions to your favorite scripting language.

I created a text file that contains all of the obfuscated strings in the malware script. To do this, I searched the malware script for all calls to function s41dk() since this is the function that de-obfuscates strings in the script. I ran the file through my Python script. The results are shown below in Table 2.

Table 2 – De-obfuscated Strings

Obfuscated String	De-Obfuscated String
w/n2wrg=	ASCII
0Z/js7noiPGZ9vXnxJ2ptsa qgJu58NPSz+jdo5jD2+rgq5G/2MTVxur c79as2dDIlauQ19v1rLmg	S5V8HkvS.open("GET", uQxzc, false);S5V8HkvS.send(null);
spuHuMW2yJWPoMTA6fHOt8S26/ 3Syc/k+tug0a324s/S2v7v6pPgmsfR/u3rwdnmn+yO08Xq8ffcwvOI+Q==	0123456789ABCDEFGHI JKLMNOPQRSTUVWXTZabcdefghiklmno pqrstuvwxyz
w+76z7OtrdbF/OTv	ADODB.Stream
6t7B+8us0Q==	http://
0tja6JTwjQ==	Process
1u/42w==	TEMP
rM/N7g==	.exe
z/ntxr2x0PHS6/Pn2O3Gvcuq9uc=	MSXML2.ServerXMLHTTP
z8PW+Z7wkcTDt93P5v3fpdM=	Microsoft.XMLHTTP
z+aH87XXxozl7Ouq49jbl9fSgoewvog=	ML2xDT8.Run(ImPfT, 0);
p/nM+IXmk/DY9vGn9uHOvNOi3A==	%SystemRoot%\TEMP\~
rN7Y+w==	.tmp
1fnW+Zjziozk8eDuxg==	WScript.Shell
99qFu8Ktnc3at/DjkdH+wbLQy9mi4fe ah6WY7cXB3u388tja5p6 tl8zR9r7x2tr/g+KKx5nw6+TF	up003.com.ua;du01.in; du02.in;citypromo.info;spotrate.info
4ITQ85Q=	

We are making good progress. Now we need to replace the obfuscated strings in the script to understand what the functions do. A fully commented listing of the beautified malware script with de-obfuscated strings can be found in bundle.

Documenting

The final step in our analysis is the documentation of the script functions. I replaced all calls to the s41dk() with the de-obfuscated strings the function would return. Below is a table containing the description of each function.

Table 3 – Function Descriptions

Function Name	Function Description
svjqxkqL5vb(u45jKF2cnu)	Ruse function. Called 7 times. Return value not used.

41dk(qqQFVIpO03, yKfNtb)	Decodes Base64 encoded string. Uses standard Base64 decode algorithm.
KiEuqVtyP1A(ERfyC3i)	Secondary obfuscation algorithm. Uses XOR, bit-shifting.
o2t84kpEE(AaUg5G)	Determines if a file is a PE. ("MZ" in first two bytes.)
tNAaDaVD(S5V8HkvS, tknjLQDR, uQxzc, h162w)	Downloads a file from the Internet and saves it with a random file name in %SYSTEM%\Temp folder.
U2D2PABJPv(rx1SRpmtJm, UiSt3EtmV)	Creates a randomly named filename.
zA8QwjCFyA(HlzNb5)	Downloads a file from the Internet and executes it.
FiVvczeeGk(QpATGC)	Determines if the passed in filename exists.
UK8wMwIhuG(ML2xDT8)	Determines if there is a file in the %SYSTEM%\Temp folder with the same name as this script plus a .tmp extension.

To complete the documentation of the script, let's create a list of the scripts actions.

- Script entry-point creates a WScript.Shell ActiveX object.
- Look for a file named "~" + script name + ".tmp" in %SYSTEM%\Temp. If file exists then exit.
- If above file does not exist, loop through a list of 5 hard-coded domain names and try to download a file. Give the file a random file name and store it in %SYSTEM%\Temp and execute it.
- If the Internet file download fails, execute a file named "~" + random name + "b.exe" in %SYSTEM%\Temp
- Exit script.

Based on the above action list, this script appears to act as an updater for the Pinkslipbot binaries on a compromised system. This would explain why the script is executed every four hours as a scheduled job.

Summary

In this exercise, we converted an unreadable mess of a JScript file into a format that unlocked its actions. We reverse-engineered the script obfuscation algorithm and wrote a simple Python script to emulate it. Using the Python script, we de-obfuscated the strings in the malware script and documented what each function does. Finally, we listed the actions of the script.

In this effort, it is interesting to note why the malware authors went to all this trouble for such a simple script. First, the use of JScript makes sense. It will run on any Windows platform and does not require compilation. Since it is not a portable executable (PE) and is obfuscated, it can slip by most security countermeasures.

Not only is obfuscation used to defeat the security infrastructure, it can also defeat security analysts. At first glance, the file contents appear unreadable. Without taking a close look, the file may be ignored because it appears to be encrypted or a binary file.

Curious and tenacious investigators know better. Through this analysis process, I hope I have convinced you that you have the skills to reverse-engineer malware if you are willing to spend the time.