Decoding Obfuscated Strings in Adwind

From the latter half of 2015 to 2016, there have been an increasing number of cyber attacks worldwide using Adwind, a Remote Access Tool [1]. JPCERT/CC also received incident reports about emails with this malware in its attachment.

Adwind is malware written in Java language, and it operates in Windows and other OS as well. It has a variety of functions: to download and execute arbitrary files, send infected machine information to C&C servers and extend functions using plug-ins.

One of the characteristics of Adwind is its frequent updates. In an extreme case, an update was released merely in a two-week interval. When investigating Adwind-related incidents, it is important to correctly examine the functions of the Adwind version in use.

The challenge is, however, the strings stored within Adwind, which count up to about 500, are artfully obfuscated, and they need to be decoded in order to analyse the malware’s function. JPCERT/CC created a tool “adwind_string_decoder.py”, which efficiently decodes such obfuscated strings. This blog article describes how this tool works.

Although Adwind has multiple generations [1], this blog article and the tool created will examine the new Adwind versions which have been used in recent attacks since the latter half of 2015.

Obfuscated strings

Most of the strings that Adwind has are obfuscated as in Figure 1. The number of such strings differs depending on the Adwind’s version, but there are about 500. These strings look totally different in each Adwind version.

```
module.Server.settings.put(
    org.jsocket.b.IiiIIiiIIi.IIIiIiJSocket(",<)(1(4*"),
    (new StringBuilder()).insert(0, java.lang.System.getProperty(
        org.jsocket.a.iIIiiiiIII.IIIiIiJSocket("¥017T8@f0:J8"))).append(org.jso
module.Server.settings.put(
    org.jsocket.a.iIIiiiiIII.IIIiIiJSocket("¥tf¥020"),
    org.jsocket.a.IiiIIiiIIi.IIIiIiiiIiiIiiIi());
module.Server.settings.put(
    org.jsocket.b.IiiIIiiIIi.IIIiIiJSocket("¥177¥f¥r&9)(, 6!"),
    java.lang.System.getProperty(
        org.jsocket.a.iIIiiiiIII.IIIiIiJSocket("T%D)¥0242H)¥tnm¥037¥t+W:R2H3"))
```

Figure 1: Decompiled codes containing obfuscated strings

```
org.jsocket.a.iiIiiiiIII.IIIiIiJSocket.put(
    org.jsocket.a.iiIiIiiIIi.IIIiIiJSocket("OSJGRGWE"),
    (new StringBuilder()).insert(0, java.lang.System.getProperty(
        org.jsocket.a.iiIiIiiIIi.IIIiIiJSocket("Rie};rgwe"))).append(org.jsocke
org.jsocket.a.iiIiiiiIII.IIIiIiJSocket.put(
    org.jsocket.a.iiIiIiiIIi.IIIiIiJSocket("T[M"),
    org.jsocket.a.iiiiIIiiiI.IIIiIiJSocket());
org.jsocket.a.iiIiiiiIII.IIIiIiJSocket.put(
    org.jsocket.a.iiIiIiiIIi.IIIiIiJSocket("¥020obEVJGOOUN"),
    java.lang.System.getProperty(
        org.jsocket.a.iiIiIiiIIi.IIIiIiJSocket("ixyt)out43PB4vjgooun")));
```

Figure 2: Decompiled codes in another Adwind version

Figure 1 and Figure 2 describe codes which correspond to the same process. Both figures have line feeds inserted and are indented so that the decompiled codes can be read easier. Figure 2 is the Adwind version which was seen in mid-August 2015 and Figure 1 in late November 2015. As previously mentioned, Adwind gets updated frequently, and furthermore, the codes are obfuscated using different keys for each version.

The red-marked sections in Figure 1 and 2 indicate part of the process for collecting information to be sent to C&C servers, and contain a process for obtaining infected usernames. However, when calling the process for collecting information, the object (which is the username) is specified in the obfuscated strings. This makes it impossible to tell from the codes that the malware intends to collect infected usernames, unless they are decoded. Furthermore, it is difficult to decode the strings since the keys used for obfuscation are scattered and different for each Adwind version (as described later).

In incident analysis, damage caused by the infection and the next attack sequence can sometimes be predicted by specifying the information sent to C&C servers. From the analysis perspectives, it is important to closely examine what kind of information the malware is targeting. For this purpose, static code analysis has to be effectively conducted, and about 500 obfuscated strings for various Adwind versions need to be quickly decoded.

Analysing the string-decoding process

Generally in stream cipher, in order to encrypt data *m* (with a random length), usually a pseudo-random number sequence *k* (with the same length as *m*) is created using its encryption key, and an encrypted string is generated by *m* XOR *k*. By combining the XOR for*m* XOR *k* and the pseudo-random number sequence *k* (which is described as (*m* XOR *k*) XOR *k*) using XOR operation, the string is decrypted to derive the plaintext *m*.

Obfuscation in Adwind is conducted in a method which is similar to the abovementioned stream cypher. However, Adwind creates *k* in a different method, not from an encryption key. In this article, *k* for Adwind is referred to as an “obfuscating key”.

Codes in Adwind contain some functions which take an obfuscated string as an argument, and returns its decoded string. These functions are referred to as F_i hereafter. One thing to note here is that F_i returns different results even for the same input, if the caller method is different. This means that, in order to do static analysis and obtain the obfuscating key corresponding to a certain string, it is necessary to understand which F_i processes the strings, as well as in which method the string exists to call for F_i. The following describes what kind of obfuscating key F_i generates to process decoding.

F_i takes its caller’s method name and class name as the basis, and generates an obfuscating key by giving them a transform process derived from the following factors. This process is repeated until it gains a certain length required for a key.

Factor 1: Which comes first when concatenating the method name and class name

Factor 2: The value used in the operation for transforming the basis string to a completely different string

All F_i consist mostly of the same codes, however, only those corresponding to the above factors have different codes in each F_i. Furthermore, Factor 2 does not exist in the code as an immediate constant, and is derived through obfuscated codes including bit-operations.

Adwind contains at least 5 varieties of F_i and about 60 methods including obfuscated strings, which means that it has a combination of about 100 obfuscating keys. Although F_i consists of relatively simple codes, this number makes it fairly difficult to remove obfuscation.

Additionally, the two factors mentioned above vary in each Adwind version. Therefore, even if we create a decoding tool for a certain version of Adwind, it cannot be applied to other versions.

On the other hand, F_i has the following characteristics in common:

- Has one argument of a string object

- Is a static function that returns a decoded string as a string object

- Contains a certain API call to obtain the caller's information

- Has limited varieties of instructions within the function

Based on the features, JPCERT/CC created a tool to automatically decode obfuscated strings using a method which does not rely on the Adwind version as much as possible.

adwind_string_decoder.py

This tool is available on GitHub. Feel free to download for your use.

> JPCERTCC/aa-tools - adwind_string_decoder.py
> https://github.com/JPCERTCC/aa-tools

In order to use adwind_string_decoder.py, a disassembler, javap, is required which is included in JDK (Java Development Kit). Users are required to set a path to javap, or configure so that the environment variable JAVA_HOME is pointed to the JDK folder.

adwind_string_decoder.py basically processes in the following sequence:

1. Open the selected jar file and call a disassembler
2. Scan all the disassembled codes, and extract functions which seem to be decoding functions from the arguments and types
3. Judge if it really is a decoding process from the kinds of instructions and sequences that appear in the function
4. If it is a decoding process, derive Factor 1 and 2 to generate obfuscating keys
5. Scan all the codes again and extract parts which call for the decoding process
6. Derive each method name and class name, and use them as the basis for obfuscating keys
7. Generate obfuscating keys and decode the strings

Before using adwind_string_decoder.py – Unpacking Adwind

Typically, Adwind is packed, and its main jar file is hidden in the artifact's jar file. Since adwind_string_decoder.py does not have the function to unpack Adwind, users are required to run Adwind in an analysis environment beforehand, and extract the jar image that appears in its memory. The jar image tends to disappear easily from the memory, however, it could be easier to extract it if you set a breakpoint in the API which reads the jar file, by using a Java debugger (e.g. jdb).

Executing adwind_string_decoder.py

To decode obfuscated strings, select the unpacked jar file and output file, and execute as follows:

```
python adwind_string_decoder.py sample.jar output.jasm
```

Then it outputs disassembled codes which contain decoded strings as comments, as in Figure 3.

```
72: ldc_w           #432 // String .<)(1(4*
75: invokestatic    #365 // STRING USERNAME // Method org/jsocket/b/IiiIIiiIIi.III
78: new             #22  // class java/lang/StringBuilder
81: dup
82: invokespecial   #23  // Method java/lang/StringBuilder."<init>":()V
85: iconst_0
86: ldc_w           #434 // String  T8@fO;J8
89: invokestatic    #363 // STRING user.name // Method org/jsocket/a/iIIiiiiIII.II
92: invokestatic    #24  // Method java/lang/System.getProperty:(Ljava/lang/String
95: invokevirtual   #368 // Method java/lang/StringBuilder.insert:(ILjava/lang/Str
```

Figure 3: Disassembled codes with some decoded strings inserted

Also, if you execute without any output files, the output of the disassembled codes will be omitted, and you can output decoded strings only to the standard output, as in Figure 4.

```
python adwind_string_decoder.py sample.jar
```

```
COMMAND
IDLE
User-Agent
Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.
Host

RAM
JRE_VERSION
java.runtime.version
SERVER_VERSION
v2.2
LAST_MODIFIED
```

Figure 4: Output of decoded strings only

It is also possible to scan the java codes (outputs of the decompiler), and replace the function call and argument with the decoded string. This option only supports codes in Fully Qualified Name (FQN) format. For example, you can obtain the output in Figure 6 from codes as in Figure 5. Since adwind_string_decoder.py does not have a decompiling function, you need to output a file with the decompiler and store it in a folder beforehand. After selecting that folder and a new folder for outputting the decoded file, execute as follows:

```
python adwind_string_decoder.py sample.jar source_folder output_folder
```

```
module.Server.settings.put(
    org.jsocket.b.IiiIIiiIIi.IIIiIiJSocket(",<)(1(4*"),
    (new StringBuilder()).insert(0, java.lang.System.getProperty(
        org.jsocket.a.iIIiiiiIII.IIIiIiJSocket("¥017T8@f0:J8"))).append(org.jso
module.Server.settings.put(
    org.jsocket.a.iIIiiiiIII.IIIiIiJSocket("¥tf¥020"),
    org.jsocket.a.IiiIIiiIIi.IIIiIiiiIiiIiiIi());
module.Server.settings.put(
    org.jsocket.b.IiiIIiiIIi.IIIiIiJSocket("¥177¥f¥r&9)(, 6!"),
    java.lang.System.getProperty(
        org.jsocket.a.iIIiiiiIII.IIIiIiJSocket("T%D)¥0242H)¥tnm¥037¥t+W:R2H3"))
```

Figure 5: Decompiled codes before decoding

```
module.Server.settings.put(
    "USERNAME",
    (new StringBuilder()).insert(0, java.lang.System.getProperty(
        "user.name")).append("_").append(org.jsocket.a.IiiIIiiIIi.IIIiIiJSocket
module.Server.settings.put(
    "RAM",
    org.jsocket.a.IiiIIiiIIi.IIIiIiiiIiiIiiIi());
module.Server.settings.put(
    "JRE_VERSION",
    java.lang.System.getProperty(
        "java.runtime.version"));
```

Figure 6: Decompiled codes after decoding

Using these decoded strings, it is easy to understand what kind of information the malware intends to collect and send to C&C servers. It is also possible to find out which OS can be infected by the malware, and how the Adwind functions may differ in each OS.

Summary

Since early February 2016, attacks using these new Adwind versions have been less and less seen. This may be good news, however, it seems that there are still new samples found at the end of February 2016. We hope that the tool we introduced here would be of your help in case if you see any new versions of Adwind.

- Kenichi Imamatsu

(Translated by Yukako Uchida)

Reference:

[1] Adwind: FAQ - Securelist
https://securelist.com/blog/research/73660/adwind-faq/

Appendix

SHA-256 Hash value of the sample

- 033db051fc98b61dab4a290a5d802abe72930338c4a0dd4705c74eacd84578d3
- f8f99b405c932adb0f8eb147233bfef1cf3547988be4d27efd1d6b05a8817d46