JEShell: An OceanLotus (APT32) Backdoor

norfolkinfosec.com/jeshell-an-oceanlotus-apt32-backdoor/

norfolk March 24, 2019

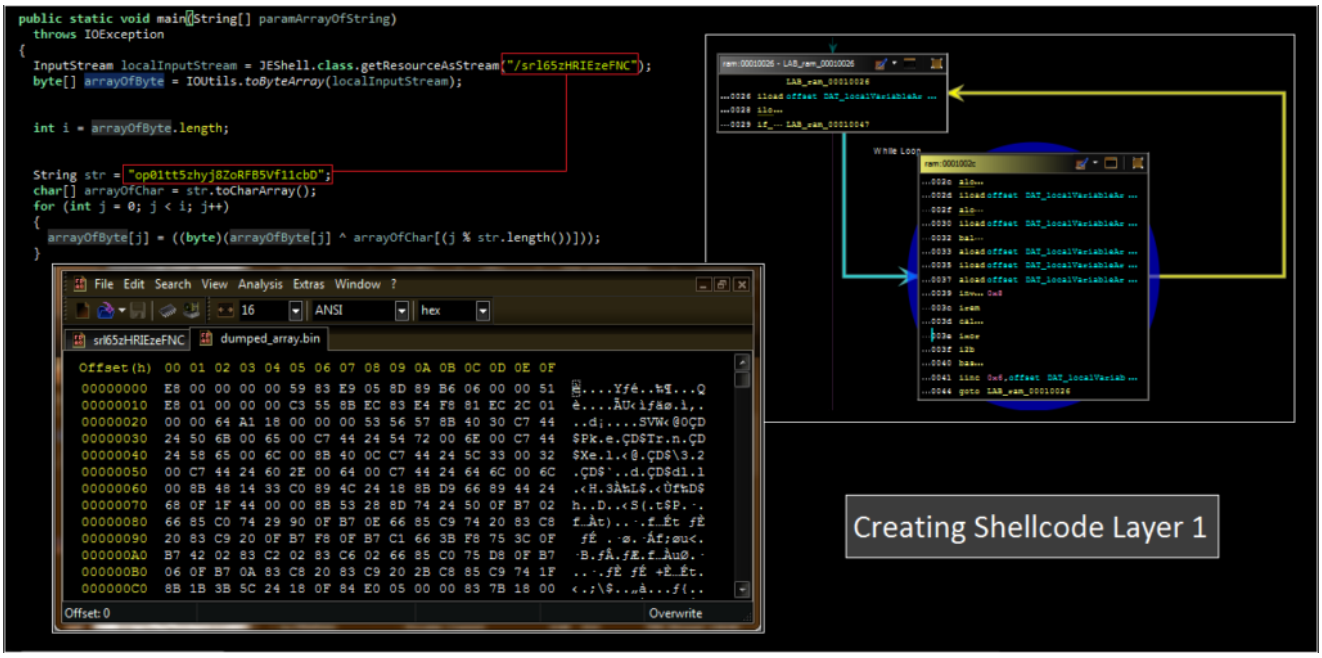
Recently, various industry and media sources have publicly reported that OceanLotus, a suspected Vietnam state-sponsored adversary, has conducted multiple targeted intrusions against auto manufacturers. This post examines a second-stage tool, JEShell, used during one such intrusion.

JEShell contains code-level overlaps with the OceanLotus KerrDown malware first publicly described in a Medium post and a Palo Alto Unit42 post. At a high level, JEShell is functionally similar to the KerrDown malware: both families decode and run layers of shellcode with the intention of downloading or directly installing a Cobalt Strike Beacon implant. Unlike KerrDown (a Windows DLL), JEShell is written in Java. JEShell is delivered alongside (rather than instead of) KerrDown and other implants and in some cases shares the same C2, likely as a measure of redundancy for the attacker.

This post examines one of two identified JEShell hashes:
MD5: dfc78da5202a70066eba124660fd5085
SHA1: 8cad6621901b5512f4ecab7a22f8fcc205d3762b
SHA256: ea854e2e17615c54edbd6ee2babb874d957f094f3945992f5ac27b78b023051c
C2: update.msoffice-templates[.]info

The other known file, not examined in this post, is:
MD5: 74731674920c51668c36cc3c16f30553
SHA1: 668572ba2aff5374a3536075b01854678c392c04
SHA256: 040c1fcec79cd19a6aaedf9cabf3cc21cc6c30e6af4048087995d71fc4571cee
C2: stream.playnetflix[.]com

JEShell contains an encrypted resources with a randomized named and an XOR key (different between samples) used to decode it. The XOR is performed in a rolling fashion using a “mod” function: the first byte of the encrypted resource is XORed by the byte representation of the first character of the key, the second byte by the second character, and so on. When the end of the key is reached, it returns to the first letter. By doing this, the authors ensure that the decoded resource is not revealed or detected by a brute force mechanism.



JEShell resource (top, boxed in red) and key (middle, boxed in red) decoded into shellcode (bottom left)

The resource is decoded into a byte array and loaded into memory through one of two workflows. On a 32-bit system, the resource is injected into the memory of the currently running process (Java.exe). On a 64-bit system a process is randomly selected and created from a hardcoded list and the array is injected into that process. The process list used on 64-bit systems in this sample is:

“C:\\Windows\\SysWOW64\\ARP.exe”, “C:\\Windows\\SysWOW64\\at.exe”, “C:\\Windows\\SysWOW64\\auditpol.exe”, “C:\\Windows\\SysWOW64\\bitsadmin.exe”, “C:\\Windows\\SysWOW64\\bootcfg.exe”, “C:\\Windows\\SysWOW64\\ByteCodeGenerator.exe”, “C:\\Windows\\SysWOW64\\cacls.exe”, “C:\\Windows\\SysWOW64\\chcp.com”, “C:\\Windows\\SysWOW64\\CheckNetIsolation.exe”, “C:\\Windows\\SysWOW64\\chkdsk.exe”, “C:\\Windows\\SysWOW64\\choice.exe”, “C:\\Windows\\SysWOW64\\cmdkey.exe”, “C:\\Windows\\SysWOW64\\comp.exe”, “C:\\Windows\\SysWOW64\\diskcomp.com”, “C:\\Windows\\SysWOW64\\Dism.exe”, “C:\\Windows\\SysWOW64\\esentutl.exe”, “C:\\Windows\\SysWOW64\\expand.exe”, “C:\\Windows\\SysWOW64\\fc.exe”, “C:\\Windows\\SysWOW64\\find.exe”, “C:\\Windows\\SysWOW64\\gpresult.exe”

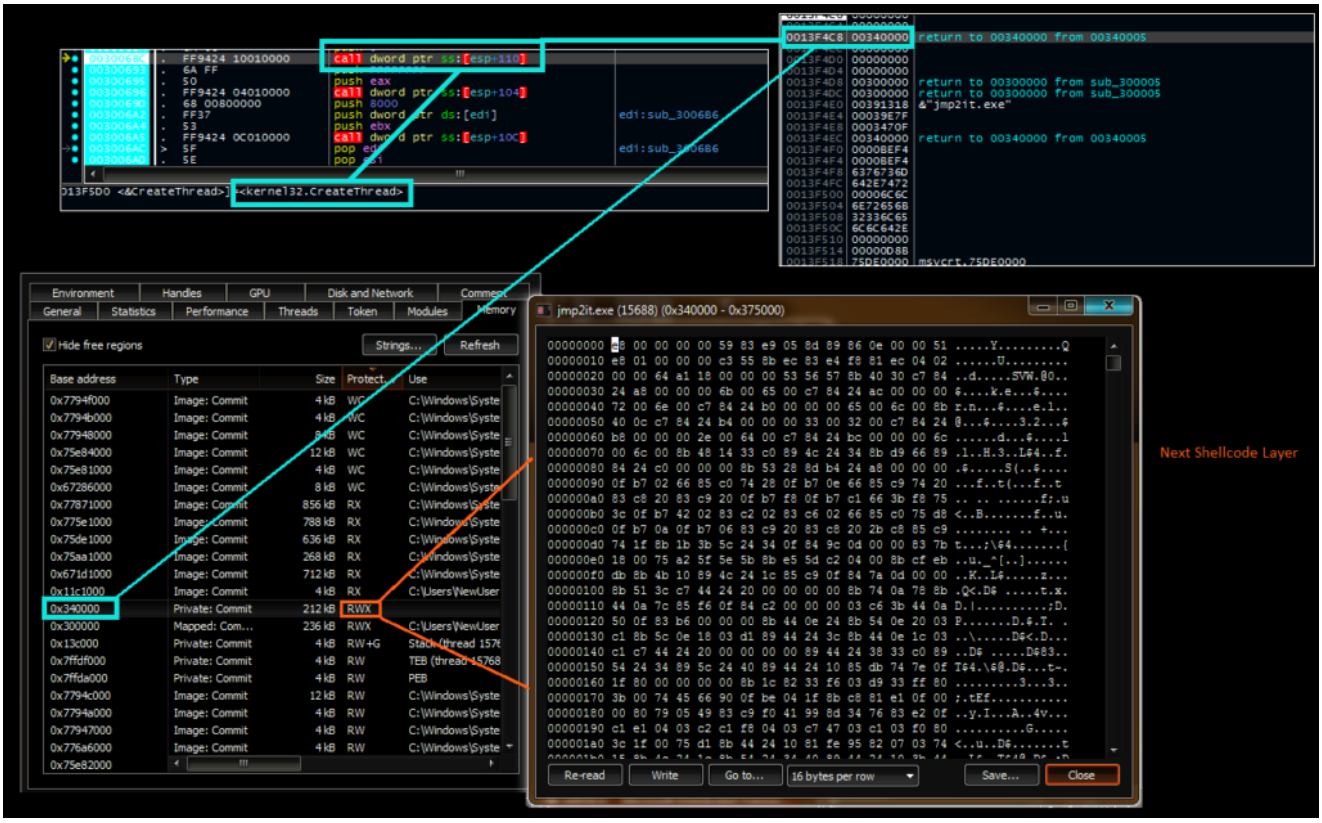
```
if (System.getProperty("sun.arch.data.model").equals("32"))
{
  localObject = iKernel32.VirtualAlloc(Pointer.createConstant(0), i, 4096, 64);
   kernel32.WriteProcessMemory(kernel32.GetCurrentProcess(), (Pointer)localObject, localMemory, i, localIntByReference);
  WinNT.HANDLE localHANDLE1 = iKernel32.CreateThread(null, 0, (Pointer)localObject, 0, 0, null);
  kernel32.WaitForSingleObject(localHANDLE1, -1);
}
else
{
  localObject = new String[] { "C:\\Windows\\SysWOW64\\ARP.exe", "C:\\Windows\\SysWOW64\\at.exe", "C:\\Windows\\SysWOW64\\auditpol.exe", "C:\\Windows'
  int m = localObject.length;
  int n = 0;
  Random localRandom = new Random();
  int i1 = n + localRandom.nextInt(m);

  WinBase.PROCESS_INFORMATION localPROCESS_INFORMATION = new WinBase.PROCESS_INFORMATION();
  WinBase.STARTUPINFO localSTARTUPINFO = new WinBase.STARTUPINFO();
  cb = new WinDef.DWORD(localPROCESS_INFORMATION.size());

  if (kernel32.CreateProcess(localObject[i1], null, null, null, false, new WinDef.DWORD(4L), null, null, localSTARTUPINFO, localPROCESS_INFORMATION))
  {
    Pointer localPointer = iKernel32.VirtualAllocEx(hProcess, Pointer.createConstant(0), i, 4096, 64);
    kernel32.WriteProcessMemory(hProcess, localPointer, localMemory, i, localIntByReference);
    WinNT.HANDLE localHANDLE2 = iKernel32.CreateRemoteThread(hProcess, null, 0, localPointer, 0, 0, null);
    kernel32.WaitForSingleObject(localHANDLE2, -1);
  }
}
```

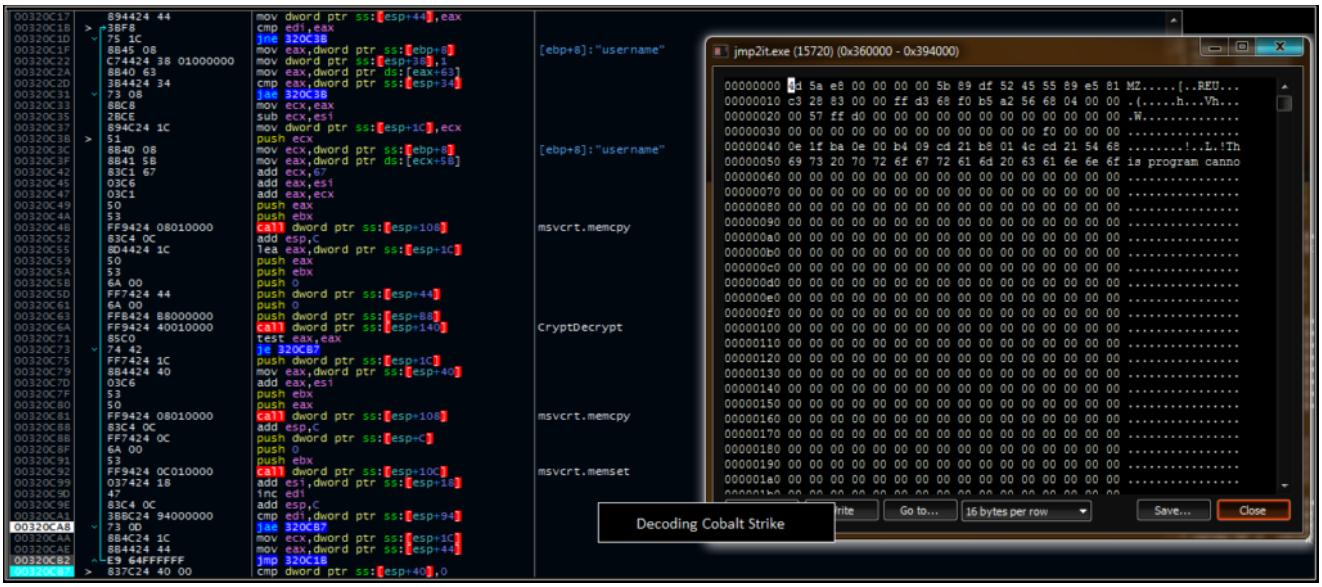
32-bit check and process injection routines for both 32 and 64-bit systems.

The injected shellcode contains multiple layers. The first layer dynamically resolves APIs and decodes a second layer (a previous post contains suggestions on analyzing shellcode), writing this to a section of virtual memory and executing it with the Kernel32.CreateThread Windows API call:



The first-stage shellcode (top left) calling Kernel32.ResumeThread on the second layer of shellcode written to an executable section of memory.

The second layer of shellcode behaves similarly: it decrypts (via CryptDecrypt) a copy of the Cobalt Strike Beacon implant into memory that is configured with a modified version of a malleable C2 profile.



Second layer of shellcode decoding a Cobalt Strike Beacon implant into memory