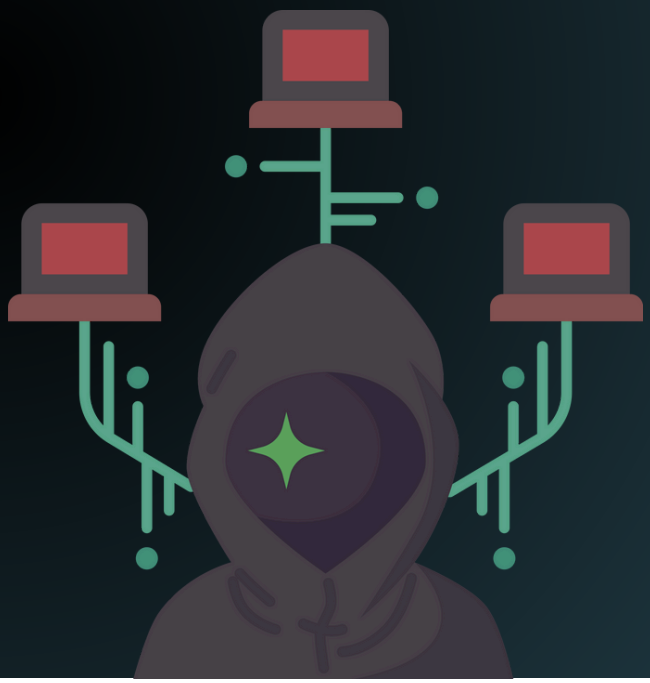


EMOTET

Malware Analysis Report

ANALYSIS DETAILS

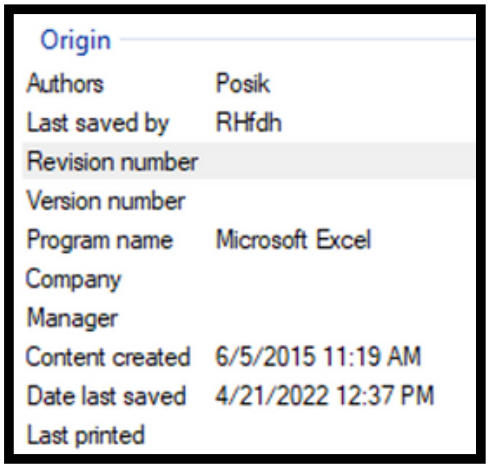
STAGE 1

FileName	Electronic Form.xls
Size	48640 bytes
MD5	669f206a1c8e7f83a4a1ce4d273cef4e
SHA-256	af436d0c3c763f7205d0217fe4f8c4e4c0940ac892edf4fa404aea24fd62fc07
File Type	Microsoft Excel 97-2003 Worksheet (.xls)

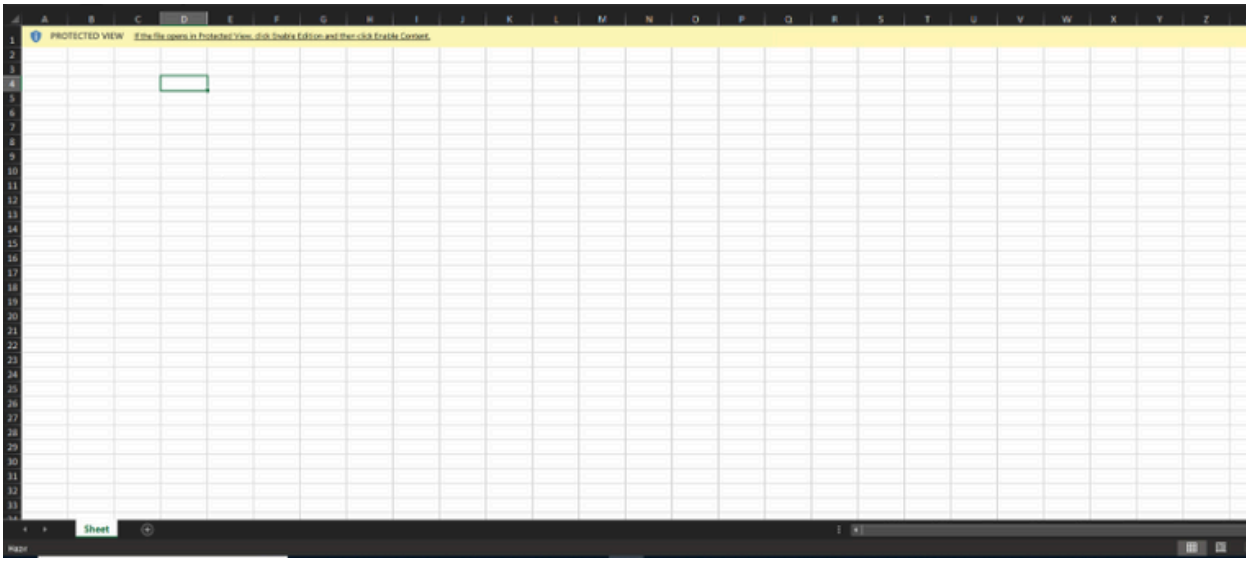
Metadata

In our case, Emotet tries to gain access through embedded malicious macro which is placed in a **".xls" file**. If it manages to connect to the server, then it downloads its Stage 2 payload. In **exif data** we have seen that author of the file is user "**Posik**" and "**RHfdh**" is the last account that modified the file.

When we open the file, we see an empty excel file with an image at first row imitating excel pop-ups to convince victim to enable macros in the file.

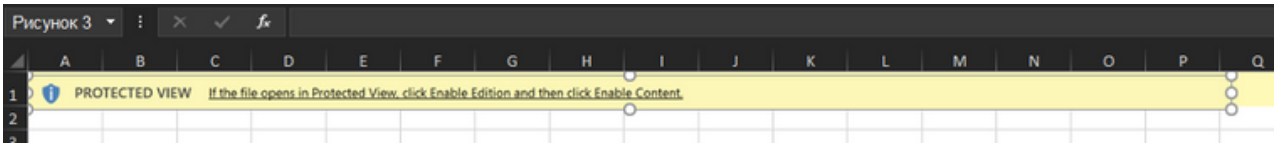


File Information



What user will be seeing when the file is opened

When we click the image we can see the image name written in Russian.



Russian name for the image. When translated into English, it means:"Figure 3"

Any.Run Sandbox Output.

The malware tries to connect to the domains shown below until it manages to connect to one of them. After looking at these domains, we noticed that there is a high probability that these websites do not belong to the attacker. In fact, these websites are hacked by the attacker to be used as a file server for malware.

Domains found in the file

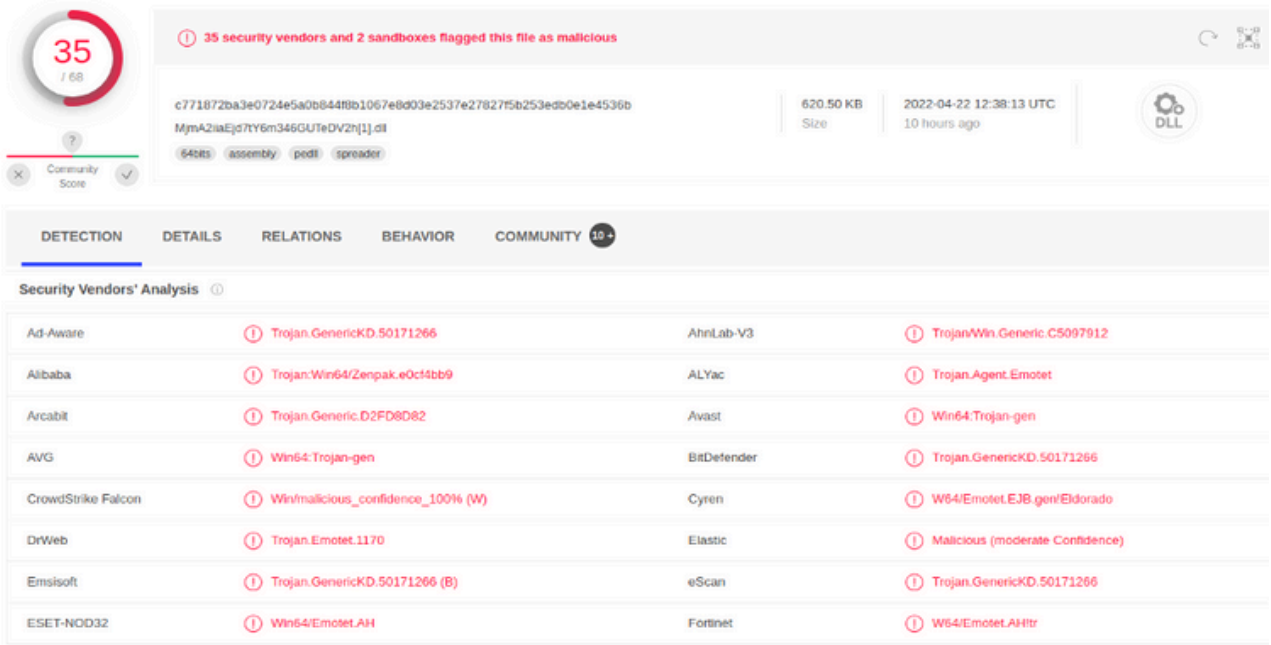
We noticed that the domains, to which the malicious payload tries to connect, are all from different countries. The reason behind this is probably that the attacker wanted to evade potential country or region based ip/domain blockings.

STAGE 2

FileName	Regex: "[a-zA-Z]{unknown Length}.[a-zA-Z]{3}"
Size	635392 bytes
MD5	dfceeb38c136043ab3cf99c0bebdd3bb
SHA-256	c771872ba3e0724e5a0b844f8b1067e8d03e2537e27827f5b253edb0e1e4536b
File Type	Dynamic Linked Library(DLL)

Metadata

After initial access, the malicious payload downloads a DLL, then executes it. The DLL is executed with the "**regsvr32.exe**" program. As seen above, the DLL file does not have a static name. When it is executed, it makes a copy of itself into the "**%APPDATA%\Local\[a-z]{unknown Length}\<filename>**" path. The name of the DLL is dynamically changed but it consists of letters, using both upper and lower case. Moreover, the file extension uses 3 randomized letters. The file path, where the DLL copied is dynamically assigned too. The name of the folder in APPDATA that is created by malware too, consists of lower and upper case letters with unknown length. We can see that the DLL is classified as "Emotet" by a number of software in the VirusTotal output below. After analyzing the code, we have detected anti-debugging, evasion and persistence capabilities in the DLL.



VirusTotal findings of malicious DLL

In the second payload, a lot of **Chinese strings** can be seen, which could point us to the single or many attackers' motives. There is a suspicious string "AdEdit" too.

	unicode u"MS Shell Dlg" (DialogResource.Dialog Font Typeface)	u"MS Shell Dlg"
	p_unicode u"关于 AdEdit(&A)..."	u" AdEdit(&A)..."
	p_unicode u"另存为"	"SX[:N\n"
	p_unicode u"所有文件 (*.*)"	u"所有文件 (*.*)"
	p_unicode u"一个未命名的文件"	u"一个未命名的文件"
	p_unicode u"隐藏(&H)"	u"隐藏(&H)"
Rsrc_StringTable_f03_804	p_unicode u"无法获取错误信息。"	u"无法获取错误信息。"
	p_unicode u"试图执行系统不支持的操作。"	u"试图执行系统不支持的操作。"
	p_unicode u"无法获取所需资源。"	u"无法获取所需资源。"
	p_unicode u"内存不足。"	u"内存不足。"
	p_unicode u"发生未知错误"	u"发生未知错误"
	p_unicode u"遇到无效参数。"	u"遇到无效参数。"
Rsrc_StringTable_f11_804	p_unicode u"无效的文件名。"	u"无效的文件名。"
	p_unicode u"打开文档失败。"	u"打开文档失败。"
	p_unicode u"保存文档失败。"	u"保存文档失败。"
	p_unicode u"是否将改动保存到 %1？"	"OX[0R "
	p_unicode u"建立空文档失败。"	u"建立空文档失败。"
	p_unicode u"该文件太大，无法打开。"	u"该文件太大，无法打开。"
	p_unicode u"无法启动打印作业。"	u"无法启动打印作业。"
	p_unicode u"启动帮助失败。"	u"启动帮助失败。"
	p_unicode u"内部应用程序出错。"	u"内部应用程序出错。"
	p_unicode u"命令失败。"	u"命令失败。"
	p_unicode u"没有足够的内存执行操作。"	u"没有足够的内存执行操作。"
	p_unicode u"系统注册项已被移除并且相应的 INI 文件（如存在）也被删...	u"系统注册项已被移除并且相应的 INI 文件（如存在）也被删除。"
	p_unicode u"不是所有的系统注册项（或 INI 文件）都被移除。"	u"不是所有的系统注册项（或 INI 文件）都被移除。"
	p_unicode u"在系统中没有找到该程序所需的文件 %s。"	u"在系统中没有找到该程序所需的文件 %s。"
	p_unicode u"该程序连接到文件 %s 中丢失的输出 %s 。该计算机上可能...	u"该程序连接到文件 %s 中丢失的输出 %s 。该计算机上可能装有一个与 %s 不兼容的...
Rsrc_StringTable_f12_804	p_unicode u"请输入一个整数。"	u"请输入一个整数。"
	p_unicode u"请输入一个数。"	u"请输入一个数。"
	p_unicode u"请输入一个在 %1 和 %2 之间的整数。"	"N*N(W "
	p_unicode u"请输入一个在 %1 和 %2 之间的数字。"	"N*N(W "
	p_unicode u"请输入不多于 %1 个的字符。"	u"请输入不多于 %1 个的字符。"
	p_unicode u"请选择一个按钮。"	u"请选择一个按钮。"
	p_unicode u"请输入一个在 0 和 255 之间的整数。"	"N*N(W "
	p_unicode u"请输入一个正整数。"	u"请输入一个正整数。"
	p_unicode u"请输入一个日期和/或时间值。"	u"请输入一个日期和/或时间值。"
	p_unicode u"请输入一个货币值。"	u"请输入一个货币值。"
	p_unicode u"请输入一个 GUID。"	u"请输入一个 GUID。"
	p_unicode u"请输入一个时间值。"	u"请输入一个时间值。"
	p_unicode u"请输入一个日期值。"	u"请输入一个日期值。"
Rsrc_StringTable_f13_804	p_unicode u"意外的文件格式。"	u"意外的文件格式。"

Strings found on malicious DLL

There is a function that creates Windows registry keys. We can see the code below.

```
HKEY FUN_10012154(longlong param_1)

{
  LSTATUS LVar1;
  DWORD local_res10 [2];
  HKEY local_res18;
  HKEY local_res20;
  HKEY local_18 [2];

  local_18[0] = (HKEY)0x0;
  local_res18 = (HKEY)0x0;
  local_res20 = (HKEY)0x0;
  LVar1 = RegOpenKeyExA((HKEY)0xffffffff80000001,"software",0,0x2001f,&local_res18);
  if (LVar1 == 0) {
    LVar1 = RegCreateKeyExA(local_res18,*(LPCSTR *)(param_1 + 0xa8),0,(LPSTR)0x0,0,0x2001f,
                            (LPSECURITY_ATTRIBUTES)0x0,&local_res20,local_res10);
    if (LVar1 == 0) {
      RegCreateKeyExA(local_res20,*(LPCSTR *)(param_1 + 0xd0),0,(LPSTR)0x0,0,0x2001f,
                      (LPSECURITY_ATTRIBUTES)0x0,local_18,local_res10);
    }
  }
  if (local_res18 != (HKEY)0x0) {
    RegCloseKey(local_res18);
  }
  if (local_res20 != (HKEY)0x0) {
    RegCloseKey(local_res20);
  }
  return local_18[0];
}
```

Code that interacts with the Windows registry

Registry keys found in the file:

```
Software\Microsoft\Windows\CurrentVersion\Policies\Explorer
Software\Microsoft\Windows\CurrentVersion\Policies\Network
Software\Microsoft\Windows\CurrentVersion\Policies\Comdlg32
```

Windows registry keys under the Software key

```
/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */

void FUN_10029a80(longlong param_1)


{
  HANDLE hProcess;
  DWORD64 in_stack_00000000;
  longlong local_res8 [4];
  PVOID local_48;
  ulonglong local_40;
  PRUNTIME_FUNCTION local_38;
  DWORD64 local_30;
  DWORD64 local_28;
  longlong local_20;
  undefined8 local_18;

  if ((param_1 == DAT_10068a20) && ((short)((ulonglong)param_1 >> 0x30) == 0)) {
    return;
  }
  local_res8[0] = param_1;
  RtlCaptureContext(&DAT_1006dfb0);
  local_30 = DAT_1006e0a8;
  local_38 = RtlLookupFunctionEntry(DAT_1006e0a8,&local_28,(PUNWIND_HISTORY_TABLE)0x0);
  if (local_38 == (PRUNTIME_FUNCTION)0x0) {
    _DAT_1006e048 = local_res8;
    DAT_1006e0a8 = in_stack_00000000;
  }
  else {
    RtlVirtualUnwind(0,local_28,local_30,local_38,(PCONTEXT)&DAT_1006dfb0,&local_48,&local_40,
                     (PKNONVOLATILE_CONTEXT_POINTERS)0x0);
  }
  _DAT_1006df20 = DAT_1006e0a8;
  _DAT_1006e030 = local_res8[0];
  _DAT_1006df10 = 0xc0000409;
  _DAT_1006df14 = 1;
  local_20 = DAT_10068a20;
  local_18 = DAT_10068a28;
  _DAT_1006dfa8 = IsDebuggerPresent();
  FUN_1003a570();
  SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)0x0);
  UnhandledExceptionFilter((_EXCEPTION_POINTERS *)&PTR_DAT_100543b8);
  if (_DAT_1006dfa8 == 0) {
    FUN_1003a570();
  }
  hProcess = GetCurrentProcess();
  TerminateProcess(hProcess,0xc0000409);
  return;
```

```
{
  BOOL BVar1;
  LONG LVar2;
  code *pcVar3;
  HANDLE hProcess;
  _EXCEPTION_POINTERS local_5a8;
  EXCEPTION_RECORD local_598;
  _CONTEXT local_4f8;

  pcVar3 = (code *)_decode_pointer(DAT_1006e498);
  if (pcVar3 == (code *)0x0) {
    FUN_1003a570();
    RtlCaptureContext(&local_4f8);
    FUN_1002ad40((ulonglong *)&local_598,0,0x98);
    local_598.ExceptionCode = 0xc000000d;
    local_5a8.ExceptionRecord = &local_598;
    local_5a8.ContextRecord = &local_4f8;
    BVar1 = IsDebuggerPresent();
    SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)0x0);
    LVar2 = UnhandledExceptionFilter(&local_5a8);
    if ((LVar2 == 0) && (BVar1 == 0)) {
      FUN_1003a570();
    }
    hProcess = GetCurrentProcess();
    TerminateProcess(hProcess,0xc000000d);
  }
  else {
    (*pcVar3)(param_1,param_2,param_3,param_4,param_5);
  }
  return;
}
```

We can see below that payload enumerates the system language. This is a common behavior that is used as a prevention against targeting allies. Or rather than blacklisting some languages, it can be used for whitelisting (target list) too.

```
local_2b8 = 0xfffffffffffffffe;
local_38 = DAT_10068a20 ^ (ulonglong)auStackY808;
local_2f8[0] = 0;
iVar5 = 0;
iVar13 = 0;
pHVar7 = GetModuleHandleA("kernel32.dll");
pFVar8 = GetProcAddress(pHVar7,"GetUserDefaultUILanguage");
if (pFVar8 == (FARPROC)0x0) {
  DVar4 = GetVersion();
  if ((int)DVar4 < 0) {
    local_2d8 = (HKEY)0x0;
    LVar3 = RegOpenKeyExA((HKEY)0xffffffff80000001,"Control Panel\\Desktop\\ResourceLocale",0
                          0x20019,&local_2d8);
    if (LVar3 == 0) {
      local_2f4 = 0x10;
      lpType = &local_2dc;
      LVar3 = RegQueryValueExA(local_2d8,(LPCSTR)0x0,(LPDWORD)0x0,lpType,local_278,&local_2f4
      iVar13 = iVar5;
      if (((LVar3 == 0) && (local_2dc == 1)) &&
         (uVar10 = FUN_1002be70((char *)local_278,&DAT_1004e24c,(longlong *)&local_2e0,lpType
         (int)uVar10 == 1)) {
        local_2f8[0] = (ushort)local_2e0;
        local_2d0[0] = ConvertDefaultLocale((uint)(local_2f8[0] & 0xfc00) | local_2e0 & 0x3ff
        local_2d0[1] = ConvertDefaultLocale(local_2e0 & 0x3ff);
        iVar13 = 2;
      }
      RegCloseKey(local_2d8);
    }
  }
  else {
    pHVar7 = GetModuleHandleA("ntdll.dll");
    if (pHVar7 != (HMODULE)0x0) {
      local_2f8[0] = 0;
      EnumResourceLanguagesA
                (pHVar7,&DAT_00000010,(LPCSTR)0x1,(ENUMRESLANGPROCA)&LAB_10002c14,
                 (LONG_PTR)local_2f8);
      if (local_2f8[0] != 0) {
        uVar6 = (uint)local_2f8[0];
        local_2d0[0] = ConvertDefaultLocale((uint)(local_2f8[0] & 0xfc00) | uVar6 & 0x3ff);
        local_2d0[1] = ConvertDefaultLocale(uVar6 & 0x3ff);
        iVar13 = 2;
      }
```

Payload tries to find language of the system

In some parts of the code we encountered a suspicious file named “hhctrl.ocx” which is an ActiveX Control file. Also, the ".ocx" files are used for exploiting the "Regsvr32" utility's functionality.

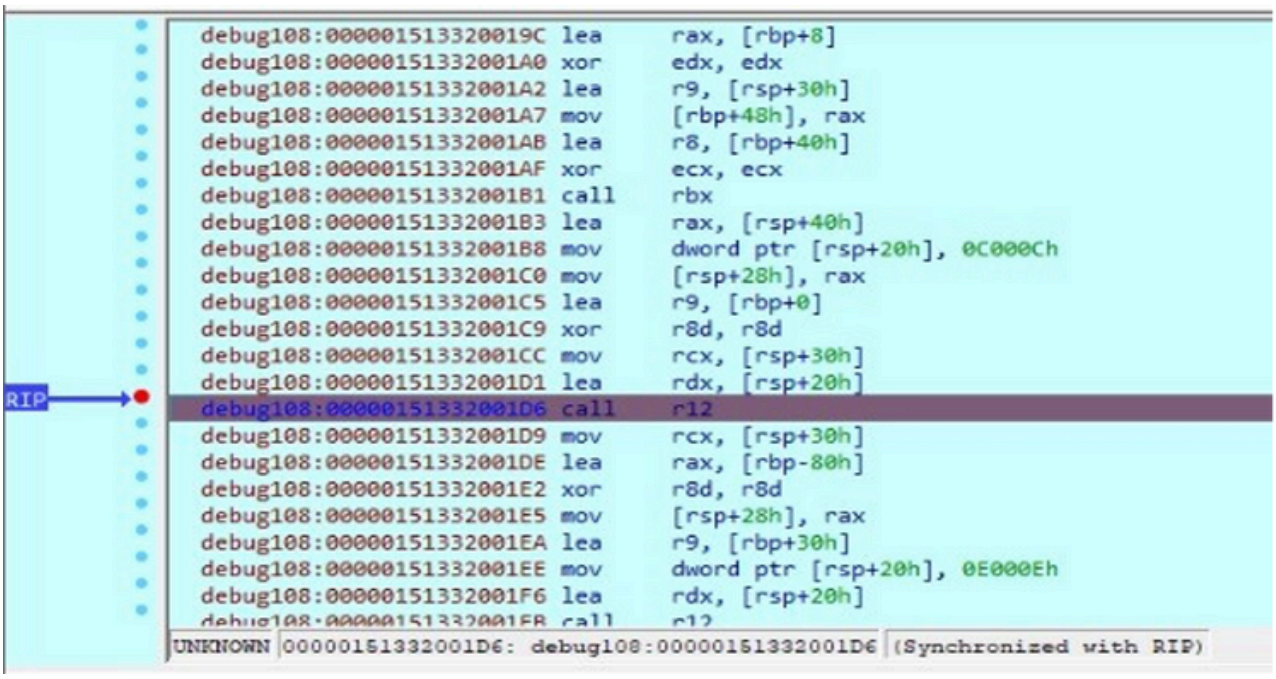
```
undefined8 FUN_1000e544(undefined8 param_1,undefined8 param_2,undefined4 param_3,undefined8

{
  code *pcVar1;
  CNoTrackObject *pCVar2;
  undefined8 uVar3;
  HMODULE hModule;
  FARPROC pFVar4;

  AfxLockGlobals(0xc);
  pCVar2 = CProcessLocalObject::GetData((CProcessLocalObject *)&DAT_1006d8a0,CreateObject);
  if (pCVar2 == (CNoTrackObject *)0x0) {
    FUN_10004f10();
    pcVar1 = (code *)swi(3);
    uVar3 = (*pcVar1)();
    return uVar3;
  }
  if (*(longlong *)(pCVar2 + 0x10) == 0) {
    hModule = FID_conflict:AfxCtxLoadLibraryA("hhctrl.ocx");
    *(HMODULE *)(pCVar2 + 8) = hModule;
    if (hModule != (HMODULE)0x0) {
      pFVar4 = GetProcAddress(hModule,"HtmlHelpA");
      *(FARPROC *)(pCVar2 + 0x10) = pFVar4;
      if (pFVar4 != (FARPROC)0x0) goto LAB_1000e5d3;
      FreeLibrary(*(HMODULE *)(pCVar2 + 8));
      *(undefined8 *)(pCVar2 + 8) = 0;
    }
    uVar3 = 0;
  }
  else {
LAB_1000e5d3:
    AfxUnlockGlobals(0xc);
    uVar3 = (**(code **)(pCVar2 + 0x10))(param_1,param_2,param_3,param_4);
  }
  return uVar3;
}
```

Suspicious ocx file and access to HtmlHelpA function

For anti-debugging and evasion, the payload uses "**LdrGetProcedureAddress**" function to get starting addresses of functions it will execute. "**LdrGetProcedureAddress**" function takes ordinal or name as a parameter, in this payload name parameter is given. The name parameters are decoded from the executable at runtime to make detection harder. We can see a part of the code below, where this method of calling is utilized, and function names found in the stack that are used in this call as parameters.



After setting the parameters, LdrGetProcedureAdress is called several times. We can see instructions that call LdrGetProcedureAdress from its adress that is loaded into r2 register. Call happens at "**call r12**" instruction.

Registers, we can see name parameter on RAX and function address at R12 register.

```
Stack[00000CB4]:0000004FCECBDE1D db    1
Stack[00000CB4]:0000004FCECBDE1E db    0
Stack[00000CB4]:0000004FCECBDE1F db    0
Stack[00000CB4]:0000004FCECBDE20 aGetnativesyste db 'GetNativeSystemInfo3Q'
Stack[00000CB4]:0000004FCECBDE35 db    1
Stack[00000CB4]:0000004FCECBDE36 db    0
Stack[00000CB4]:0000004FCECBDE37 db    0
Stack[00000CB4]:0000004FCECBDE38 aRtladdfunction db 'RtlAddFunctionTable'
Stack[00000CB4]:0000004FCECBDE4B db    5
Stack[00000CB4]:0000004FCECBDE4C db    0
Stack[00000CB4]:0000004FCECBDE4D db    0
Stack[00000CB4]:0000004FCECBDE4E db    0
Stack[00000CB4]:0000004FCECBDE4F db    0
Stack[00000CB4]:0000004FCECBDE50 aFlushinstructi db 'FlushInstructionCache'
Stack[00000CB4]:0000004FCECBDE65 db    0
Stack[00000CB4]:0000004FCECBDE66 db    0
Stack[00000CB4]:0000004FCECBDE67 db    0
```

Strings on Stack to be used as a parameter when LdrGetProcedure Function called

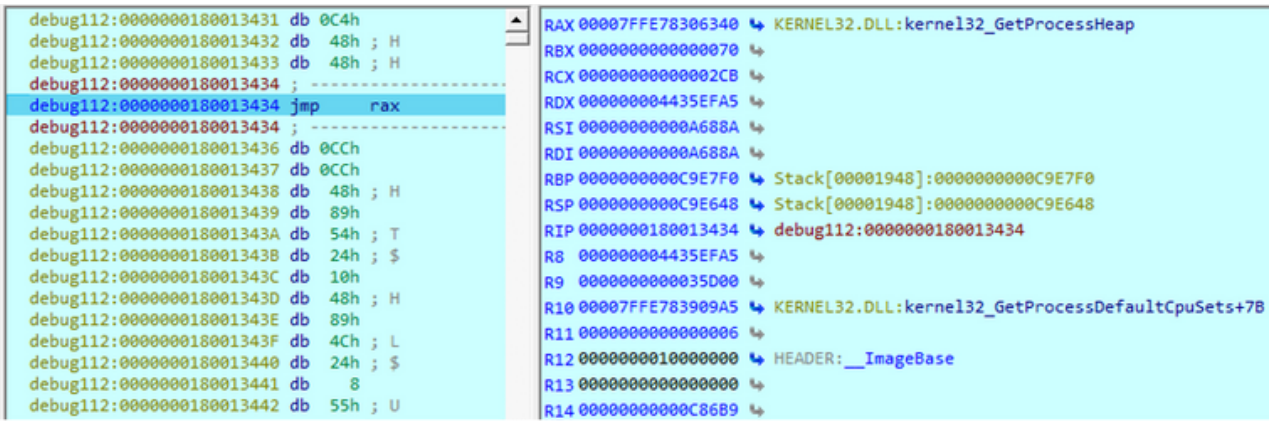
Our team has reverse engineered most of dllMain function. Summary of what dllMain does: first, it allocates a memory block for obfuscated code, then copies the obfuscated code into memory block. After that it deobfuscates the code and executes it as a function. In that function, DLL creates 2 more memory blocks: one for **another DLL in memory** and another for later uses of DLL in **dllRegisterServerFunction**. Both memory block go through similar processes with first memory block (deobfuscation). For more detailed information the created pseudocode below can be inspected:

```
int dllMain(){
 payload_1 = VirtualAlloc(NULL, 0x835, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
 // deobfuscatePayloadAndLoadToMemory(memory_block, dataPtr_1, dataPtr_2)
 deobfuscatePayloadAndLoadToMemory(payload_1, 0x107e310, 0x107d7d0);
 payload_1();
}

payload_1(){
 // loads stack with function names that will be used in dllMain
 loadDll = getLdrFunctionPointers("LdrLoadDll");
 getProcAddr = getLdrFunctionPointers("LdrGetProcedureAddress");
 loadDll("kernel32.dll");
 /* This is how actual program works, but for sake of simplicity we will mention
  these functions with their original names rather than using pointers for the
  rest of the code.
 virAlloc = getProcAddr("VirtualAlloc");
 virProtect = getProcAddr("VirtualProtect");
 flushCache = getProcAddr("FlusIntructionCache");
 getInfo = getProcAddr("GetNativeSystemInfo");
 sl = getProcAddr("sleep");
 funcTable = getProcAddr("RtlAddFunctionTable");
 loadLib = getProcAddr("LoadLibraryA");
 findR = getProcAddr("FindResourceW");
 loadR = getProcAddr("LoadResource");
 sizeofR = getProcAddr("SizeofResource");
 lockR = getProcAddr("LockResource");
 */
 res = FindResourceW(NULL, 0xD33E, 0xA); // hModule, lpName, lpType
 resHandle = LoadResource(NULL, res); // hModule, hResInfo
 resSize = SizeofResource(NULL, res); // returns 0x26C00
 LockResource(resHandle); // hResData
 payload_2 = VirtualAlloc(NULL,
                          resSize,
                          MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
 // payload_2 contains another dll file!
 // in this part payload_2 is deobfuscated
 infoHandle;
 GetNativeSystemInfo(infoHandle); // handle is allocated to infoHandle
 payload_3 = VirtualAlloc(0x180000000, 0x2a000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
 //payload_3 codes will be executed when dllRegisterServerFunction ran
 //deobfuscates some memory block then copies that memory to the new block
 VirtualProtect(payload_3 + 0x1000, 0x24C00, PAGE_EXECUTE_READ, PAGE_EXECUTE_READ);
 VirtualProtect(payload_3 + 0x26000, 0xC00, PAGE_READONLY, PAGE_READONLY);
 VirtualProtect(payload_3 + 0x27000, 0x400, PAGE_READWRITE, PAGE_READWRITE);
 VirtualProtect(payload_3 + 0x29000, 0xC00, PAGE_READONLY, PAGE_READONLY);
 FlushIntructionCache();
 RtlAddFunctionTable(payload_3+0x29000, 0xEB, payload_3);
 // FunctionTable, EntryCount, BaseAddress
 return
}
```

Pseudocode of dllMain function

After dllMain, DllRegisterServer function is executed. DllRegisterServer function calls the internal DLL. After analyzing the internal DLL, we noticed that DLL makes calls to external DLLs through layers of obfuscation. Instead of calling functions directly, it resolves the addresses of functions, then calls them through RAX register. In our analysis, each time we external function executed, DLL executes "**jmp rax**" instruction to jump to that function rather than calling the adress of that function in external DLL. This could evade sandboxes that only traces function calls.



Example: We can see "jmp rax" instruction is executed, when RAX holds "GetProcessHeap" functions address

After we found out "jmp rax" instruction is executed every time an external DLL execution happened, we used this information to gather names of executed external functions.

- BCryptCloseAlgoRithmProvider
- BCryptDeriveKey
- BCryptDestroyHash
- BCryptDestroyKey
- BCryptDestroySecret
- BCryptExportKey
- BCryptFinalizeKeyPair
- BCryptFinishHash
- BCryptGenerateKeyPair
- BCryptGenRandom
- BCryptGetProperty
- BCryptHashData
- BCryptImportKey
- BCryptImportKeyPair
- BCryptOpenAlgorithmProvider
- BCryptSecretAgreement
- BCryptVerifySignature
- CloseHandle
- CommandLineToArgvW
- CryptBinaryToStringW
- FindFirstFileW
- GetCommandLineW
- RegCloseKey
- RtlAllocateHeap
- RtlCopyMemory
- RtlGetVersion
- SHGetFolderPathW
- GetComputerNameA
- GetCurrentProcessId
- GetFileInformationByHandleEx
- GetModuleFileNameW
- GetModuleHandleA
- GetProcessHeap
- GetSystemTimeAsFileTime
- GetWindowsDirectoryW
- HeapFree
- HttpSendRequestW
- InternetCloseHandle
- InternetOpenWS
- InternetQueryOptionW
- InternetReadFile
- InternetSetOptionW
- LoadLibraryW
- LocalFree
- lstrcpyW
- CreateEventW
- OpenSCManagerW
- PathFindFileNameW
- ProcessIdToSessionId
- RegSetValueExW
- WaitForSingleObject
- snwprintf
- snprintf
- memset

Malware achieves persistence by adding registry key under "**HKCU\Microsoft\Windows \CurrentVersion \ Run**". Malicious functionality of DLL is loaded in "payload_3" where malicious payload achieves persistence and discovers filesystem, users, operating system information, etc.

```
debug017:0000000001189716 db  6Fh ; o
debug017:0000000001189717 db    0
debug017:0000000001189718 db  6Ch ; l
debug017:0000000001189719 db    0
debug017:000000001189719A db  73h ; s
debug017:000000001189719B db    0
debug017:000000001189719C db  5Ch ; \
debug017:000000001189719D db    0
debug017:000000001189719E db  72h ; r
debug017:000000001189719F db    0
debug017:0000000001189720 db  61h ; a
debug017:0000000001189721 db    0
debug017:0000000001189722 db  64h ; d
debug017:0000000001189723 db    0
debug017:0000000001189724 db  61h ; a
debug017:0000000001189725 db    0
debug017:0000000001189726 db  72h ; r
debug017:0000000001189727 db    0
debug017:0000000001189728 db  65h ; e
debug017:0000000001189729 db    0
debug017:000000001189272A db  32h ; 2
debug017:000000001189272B db    0
debug017:000000001189272C db  5Ch ; \
debug017:000000001189272D db    0
debug017:000000001189272E db  62h ; b
debug017:000000001189272F db    0
debug017:0000000001189730 db  69h ; i
debug017:0000000001189731 db    0
debug017:0000000001189732 db  6Eh ; n
debug017:0000000001189733 db    0
debug017:0000000001189734 db  3Bh ; ;
```

Malicious payload checks several folders' contents. What we see here is one of radare2's folders.

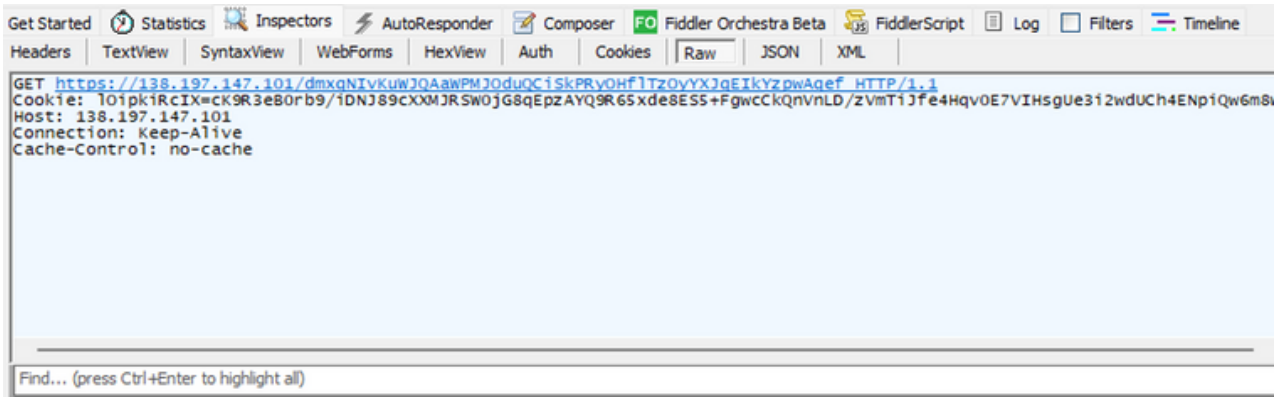
```
000069ea  0000001a  A  6USERDOMAIN=WINDEV2110EVAL
00006a1a  00000029  A  7USERDOMAIN_ROAMINGPROFILE=WINDEV2110EVAL
00006a59  0000000e  A  2USERNAME=User
00006a7d  0000001a  A  6USERPROFILE=C:\Users\User
00006aad  00000022  A  >VM_COMMON_DIR=C:\ProgramData\FEVM
00006ae5  00000012  A  >windir=C:\Windows
00006b0d  00000058  A  8_NT_SYMBOL_PATH=symsrv*symsrv.dll*C:\symbols*http://msdl.microsoft.com/download/symbols
000070a8  0000001f  A  1ALLUSERSPROFILE=C:\ProgramData
```

We can see the data that malicious file gathered in here

We noticed that DLL connects to 2 (two) different IP adresses using HTTPS protocol. We think these adresses are used as **Command & Control** centers because malware already achieved persistence at this moment.

9	200	HTTP	Tunnel to	138.201.142.73:8080	0		regsvr32:4200
10	200	HTTP	Tunnel to	138.197.147.101:443	0		regsvr32:4200
11	200	HTTPS	138.197.147.101	/dmxqNIvKuWJQAaWPMJOduQCiSkPRyOHftTz...	177	text/html; charset...	regsvr32:4200

Fiddler Proxy output. We can see that DLL is sending some data in 11th request



The cookie that is send to server is obfuscated. Server responds this request but the response is obfuscated too.

Indicator of Compromise

Type	Description	Indicator
URL	Stage 2 DLL	https[:]//dancefox24[.]de/templates/owT/
URL	Stage 2 DLL	https[:]//creemo[.]pl/wp-admin/0uDUHJ4KVAw/
URL	Stage 2 DLL	http[:]//focusmedica[.]in/fmlib/TYiQdcEj9FW0/
FILE	Stage 2 DLL	hhctl.ocx
URL	Stage 1 xls	https[:]//138[.]201.142[.]73/LlzjFOkXejhkkwKznWmaweETfyarhDC
URL	Stage 1 xls	https[:]//138[.]201.142[.]73/nzQqtfpKvlXrdedLOhwauLgz
URL	Stage 2 DLL	https[:]//dp-flex.co[.]jp/cgi-bin/Bt3Ycq5Tix/
URL	Stage 2 DLL	http[:]//dharmacomunicacao[.]com[.]br/OLD/PjBkVBhUH
URL	Stage 2 DLL	https[:]//www[.]fantasyclub[.]com[.]br/imgs/rggmVTfvT/
URL	Stage 2 DLL	http[:]//eleselektromekanik[.]com/69Iq5Pwbd0/s/
IP	Stage 2 DLL Possible C2	138[.]201.142.73:8080 (https)
IP	Stage 2 DLL Possible C2	138[.]197.147.101:443 (https)

Mitre ATT&CK

ID	Tactic	Technique
T1071.001	COMMAND AND CONTROL	Application Layer Protocol: Web Protocols
T1001.003	COMMAND AND CONTROL	Data Obfuscation: Protocol Impersonation
T1008	COMMAND AND CONTROL	Fallback Channels
T1105	COMMAND AND CONTROL	Ingress Tool Transfer
T1572	COMMAND AND CONTROL	Protocol Tunneling
T1005	COLLECTION	Data from Local System
T1087	DISCOVERY	Account Discovery
T1083	DISCOVERY	File and Directory Discovery
T1057	DISCOVERY	Process Discovery
T1012	DISCOVERY	Query Registry
T1082	DISCOVERY	System Information Discovery
T1140	DISCOVERY	Deobfuscate/Decode Files or Information
T1564.007	DEFENSE EVASION	Hide Artifacts: VBA Stomping
T1070.004	DEFENSE EVASION	Indicator Removal on Host: File Deletion
T1112	DEFENSE EVASION	Modify Registry
T1055.001	DEFENSE EVASION	Process Injection: Dynamic-link Library Injection
T1620T	DEFENSE EVASION	Reflective Code Loading
T1218.010	DEFENSE EVASION	System Binary Proxy Execution: Regsvr32
T1547.001	PERSISTENCE	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder
T1059.005	EXECUTION	Command and Scripting Interpreter: Visual Basic
T1204.002	EXECUTION	User Execution: Malicious File
T1566	INITIAL ACCESS	Phishing
T1608.001	RESOURCE DEVELOPMENT	Stage Capabilities: Upload Malware
T1592.002	RECONNAISSANCE	Gather Victim Host Information: Software