What the Pack(er)?

cyber.wtf/2022/03/23/what-the-packer/

March 23, 2022

Lately, I broke one of the taboos of malware analysis: looking into the packer stub of a couple of malware samples. Fortunately, I must say. Because I discovered something I was really surprised by. But first, a little detour.

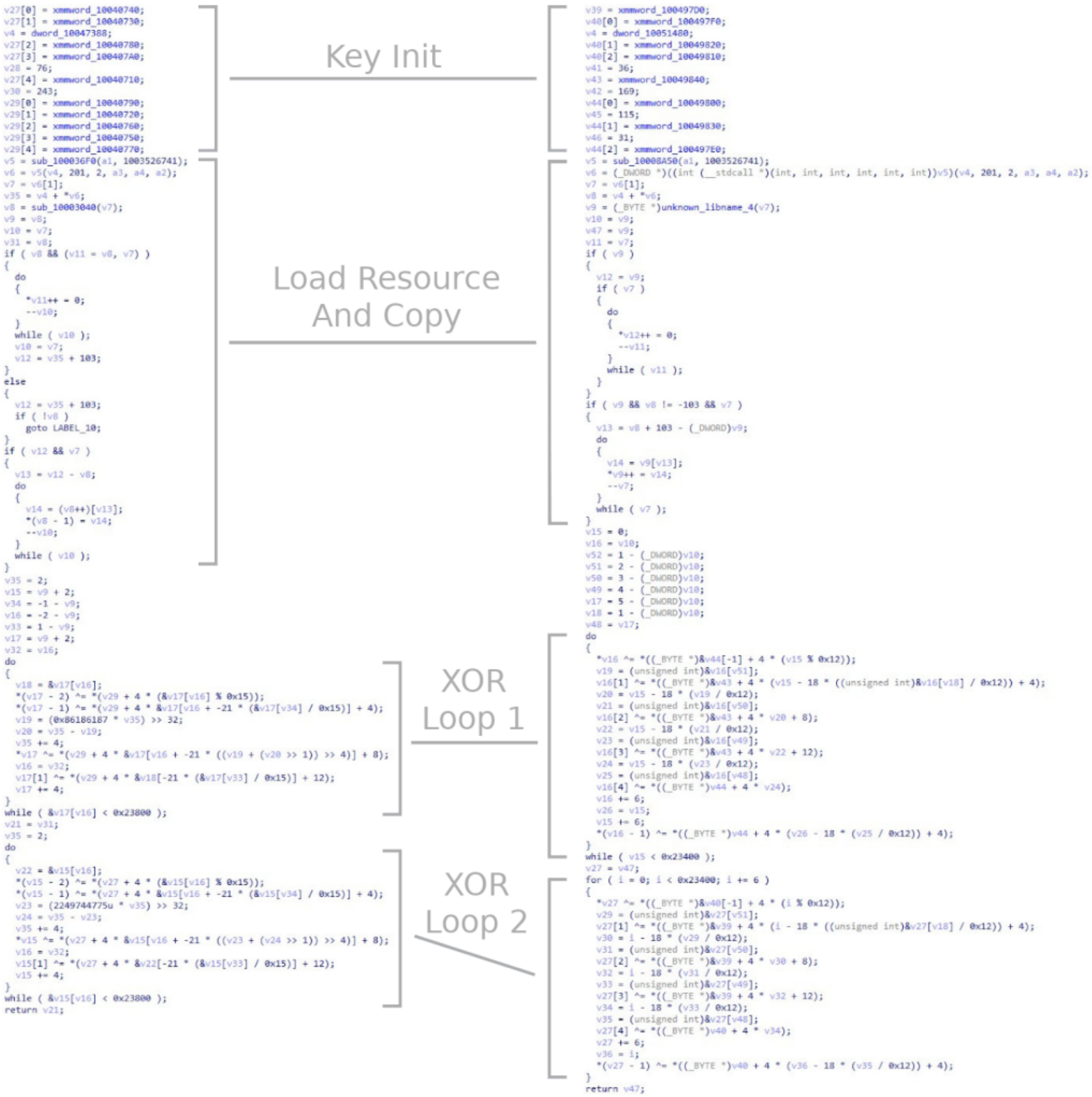
Historically, Emotet has been observed to assemble infected systems into three botnets dubbed Epoch 1, Epoch 2, and Epoch 3. After the takedown and the later resurrection, there seems to only be two botnets which have subsequently been dubbed Epoch 4 and Epoch 5. The differences between the old and the new core of the botnets are significant on the technical side – however, the old Epochs 1 through 3 shared the same core and so do the recent Epoch 4 and Epoch 5. The only noticeable difference between Epochs 1 through 3 was the config which was embedded into the Emotet core before a sample was rolled out to the victims. The same also applies to the more recent Epochs 4 and 5.

However, there is a significant difference in the operation carried out by the botnets between what happened before the disruption and what was observed since the rebirth. In the past, observations showed that Emotet bots used to drop whatever their operator's customers paid them for. Brad Duncan alone already observed Emotet dropping QakBot/QBot, Trickbot, and Gootkit. Of these, the Trickbot group seemed to be their best and longest-running customer based on the numerous observations of Trickbot being dropped by Emotet. But after the resurrection, there were no longer observations of additional malware being dropped by Emotet. Instead, starting in December 2021, researchers observed a CobaltStrike beacon being dropped onto an infected machine without any evidence that there was another malware involved. Emotet has since been reportedly and repeatedly seen to deploy CobaltStrike beacons to infected machines, so this was definitely not a one-off drop and drew the attention of our researchers.

With the context of this analysis being setup properly, we can finally come back to the actual topic of this blog post: breaking taboos by analyzing packing stubs. Enjoy!

Poking (in) Packing Stubs

For the first period of time after the resurrection, the Emotet core seems to have used XOR encryption to hide their bot from static analysis. It can easily be seen that the algorithms appear to be (almost) identical between Epoch 4 (left) and Epoch 5 (right) – disregarding a few compiler optimizations due to different key lengths:

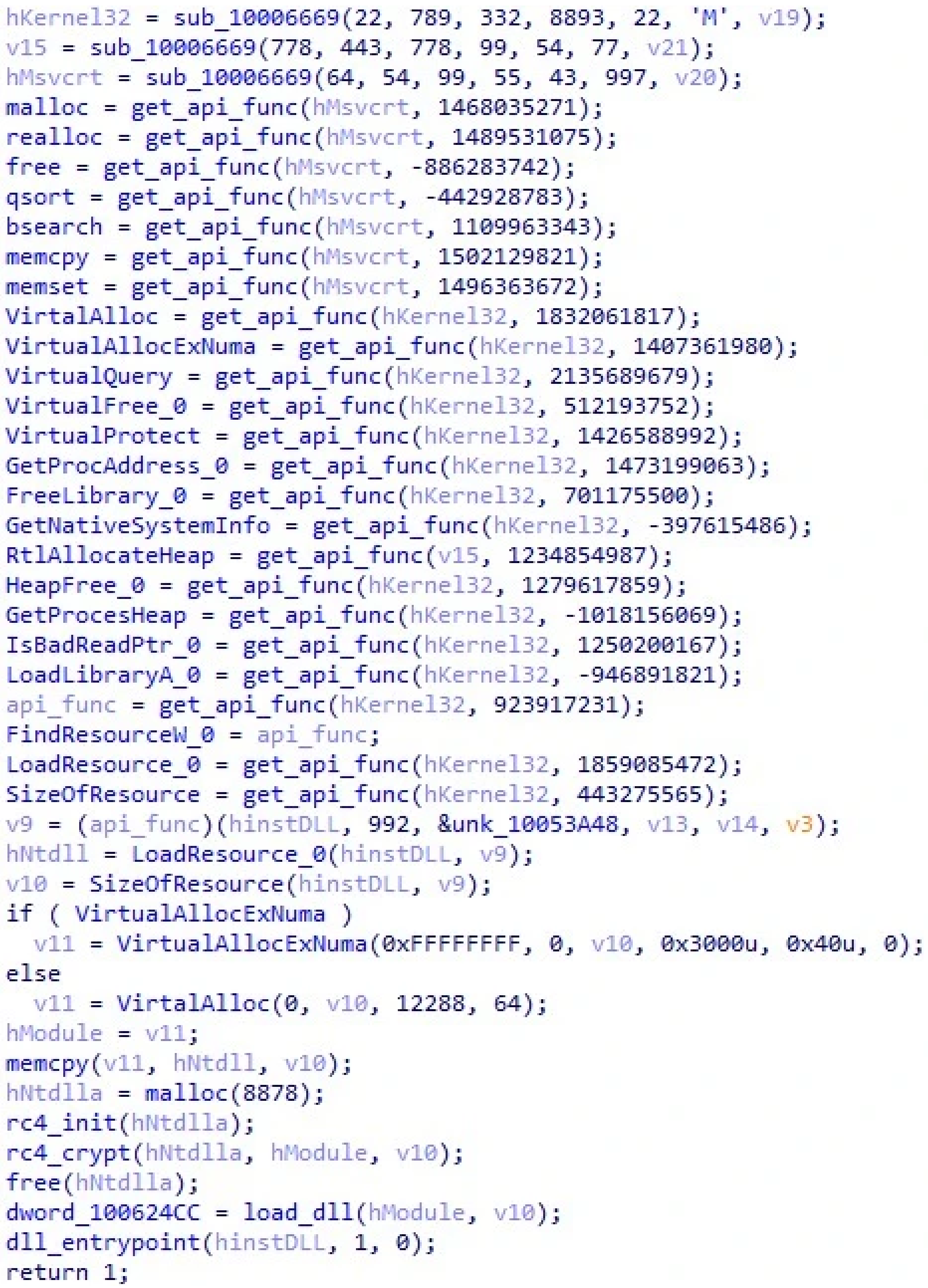


```
v27[0] = xmmword_10040740;
v27[1] = xmmword_10040730;
v4 = dword_10047388;
v27[2] = xmmword_10040780;
v27[3] = xmmword_100407A0;
v28 = 76;
v27[4] = xmmword_10040710;
v30 = 243;
v29[0] = xmmword_10040790;
v29[1] = xmmword_10040720;
v29[2] = xmmword_10040760;
v29[3] = xmmword_10040750;
v29[4] = xmmword_10040770;
v5 = sub_100036F0(a1, 1003526741);
v6 = v5(v4, 201, 2, a3, a4, a2);
v7 = v6[1];
v35 = v4 + *v6;
v8 = sub_10003040(v7);
v9 = v8;
v10 = v7;
v31 = v8;
if ( v8 && (v11 = v8, v7) )
{
  do
  {
    *v11++ = 0;
    --v10;
  }
  while ( v10 );
  v10 = v7;
  v12 = v35 + 103;
}
else
{
  v12 = v35 + 103;
  if ( !v8 )
    goto LABEL_10;
}
if ( v12 && v7 )
{
  v13 = v12 - v8;
  do
  {
    v14 = (v8++)[v13];
    *(v8 - 1) = v14;
    --v10;
  }
  while ( v10 );
}
v35 = 2;
v15 = v9 + 2;
v34 = -1 - v9;
v16 = -2 - v9;
v33 = 1 - v9;
v17 = v9 + 2;
v32 = v16;
do
{
  v18 = &v17[v16];
  *(v17 - 2) ^= *(v29 + 4 * (&v17[v16] % 0x15));
  *(v17 - 1) ^= *(v29 + 4 * &v17[v16 + -21 * (&v17[v34] / 0x15)] + 4);
  v19 = (0x86186187 * v35) >> 32;
  v20 = v35 - v19;
  v35 += 4;
  *v17 ^= *(v29 + 4 * &v17[v16 + -21 * ((v19 + (v20 >> 1)) >> 4)] + 8);
  v16 = v32;
  v17[1] ^= *(v29 + 4 * &v18[-21 * (&v17[v33] / 0x15)] + 12);
  v17 += 4;
}
while ( &v17[v16] < 0x23800 );
v21 = v31;
v35 = 2;
do
{
  v22 = &v15[v16];
  *(v15 - 2) ^= *(v27 + 4 * (&v15[v16] % 0x15));
  *(v15 - 1) ^= *(v27 + 4 * &v15[v16 + -21 * (&v15[v34] / 0x15)] + 4);
  v23 = (2249744775u * v35) >> 32;
  v24 = v35 - v23;
  v35 += 4;
  *v15 ^= *(v27 + 4 * &v15[v16 + -21 * ((v23 + (v24 >> 1)) >> 4)] + 8);
  v16 = v32;
  v15[1] ^= *(v27 + 4 * &v22[-21 * (&v15[v33] / 0x15)] + 12);
  v15 += 4;
}
while ( &v15[v16] < 0x23800 );
return v21;
```

```
v39 = xmmword_100497D0;
v40[0] = xmmword_100497F0;
v4 = dword_10051480;
v40[1] = xmmword_10049820;
v40[2] = xmmword_10049810;
v41 = 36;
v43 = xmmword_10049840;
v42 = 169;
v44[0] = xmmword_10049800;
v45 = 115;
v44[1] = xmmword_10049830;
v46 = 31;
v44[2] = xmmword_100497E0;
v5 = sub_10008A50(a1, 1003526741);
v6 = (_DWORD *)((int (__stdcall *)(int, int, int, int, int, int))v5)(v4, 201, 2, a3, a4, a2);
v7 = v6[1];
v8 = v4 + *v6;
v9 = (_BYTE *)unknown_libname_4(v7);
v10 = v9;
v47 = v9;
v11 = v7;
if ( v9 )
{
  v12 = v9;
  if ( v7 )
  {
    do
    {
      *v12++ = 0;
      --v11;
    }
    while ( v11 );
  }
}
if ( v9 && v8 != -103 && v7 )
{
  v13 = v8 + 103 - (_DWORD)v9;
  do
  {
    v14 = v9[v13];
    *v9++ = v14;
    --v7;
  }
  while ( v7 );
}
v15 = 0;
v16 = v10;
v52 = 1 - (_DWORD)v10;
v51 = 2 - (_DWORD)v10;
v50 = 3 - (_DWORD)v10;
v49 = 4 - (_DWORD)v10;
v17 = 5 - (_DWORD)v10;
v18 = 1 - (_DWORD)v10;
v48 = v17;
do
{
  *v16 ^= *((_BYTE *)&v44[-1] + 4 * (v15 % 0x12));
  v19 = (unsigned int)&v16[v51];
  v16[1] ^= *((_BYTE *)&v43 + 4 * (v15 - 18 * ((unsigned int)&v16[v18] / 0x12)) + 4);
  v20 = v15 - 18 * (v19 / 0x12);
  v21 = (unsigned int)&v16[v50];
  v16[2] ^= *((_BYTE *)&v43 + 4 * v20 + 8);
  v22 = v15 - 18 * (v21 / 0x12);
  v23 = (unsigned int)&v16[v49];
  v16[3] ^= *((_BYTE *)&v43 + 4 * v22 + 12);
  v24 = v15 - 18 * (v23 / 0x12);
  v25 = (unsigned int)&v16[v48];
  v16[4] ^= *((_BYTE *)v44 + 4 * v24);
  v16 += 6;
  v26 = v15;
  v15 += 6;
  *(v16 - 1) ^= *((_BYTE *)v44 + 4 * (v26 - 18 * (v25 / 0x12)) + 4);
}
while ( v15 < 0x23400 );
v27 = v47;
for ( i = 0; i < 0x23400; i += 6 )
{
  *v27 ^= *((_BYTE *)&v40[-1] + 4 * (i % 0x12));
  v29 = (unsigned int)&v27[v51];
  v27[1] ^= *((_BYTE *)&v39 + 4 * (i - 18 * ((unsigned int)&v27[v18] / 0x12)) + 4);
  v30 = i - 18 * (v29 / 0x12);
  v31 = (unsigned int)&v27[v50];
  v27[2] ^= *((_BYTE *)&v39 + 4 * v30 + 8);
  v32 = i - 18 * (v31 / 0x12);
  v33 = (unsigned int)&v27[v49];
  v27[3] ^= *((_BYTE *)&v39 + 4 * v32 + 12);
  v34 = i - 18 * (v33 / 0x12);
  v35 = (unsigned int)&v27[v48];
  v27[4] ^= *((_BYTE *)v40 + 4 * v34);
  v27 += 6;
  v36 = i;
  *(v27 - 1) ^= *((_BYTE *)v40 + 4 * (v36 - 18 * (v35 / 0x12)) + 4);
}
return v47;
```

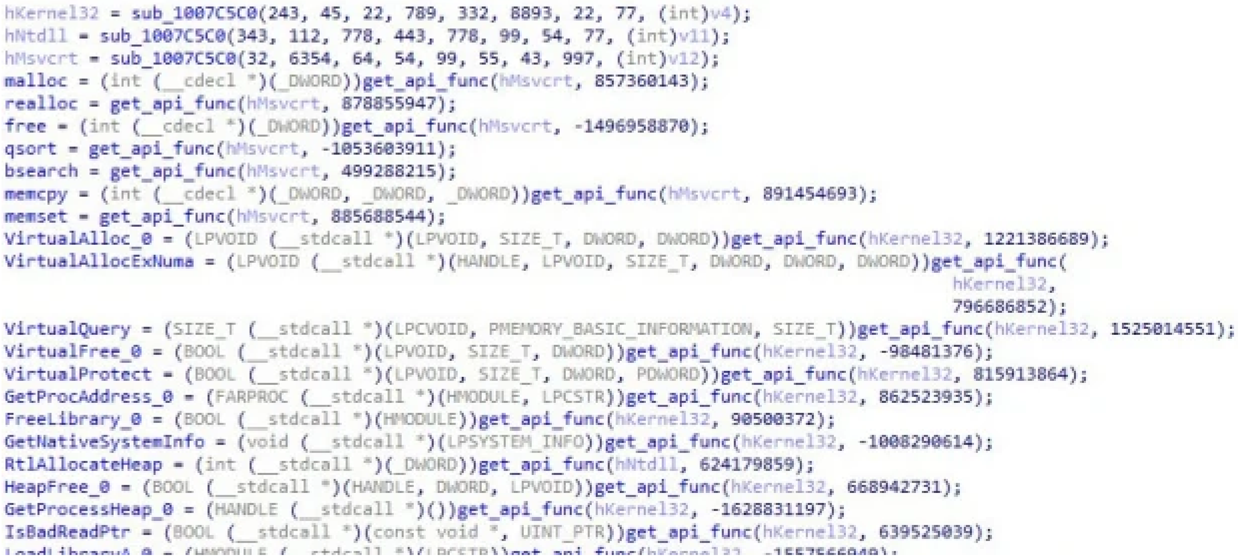
Emotet XOR Decrypt for Payload – Epoch 4 (left) vs Epoch 5 (right)

At some point, the authors changed the encryption scheme to use RC4 instead of plain XOR. Although the code applying the RC4 algorithm looks different thanks to a substantial amount of superfluous API calls, there are obvious similarities between Epoch 4 on the left and Epoch 5 on the right:

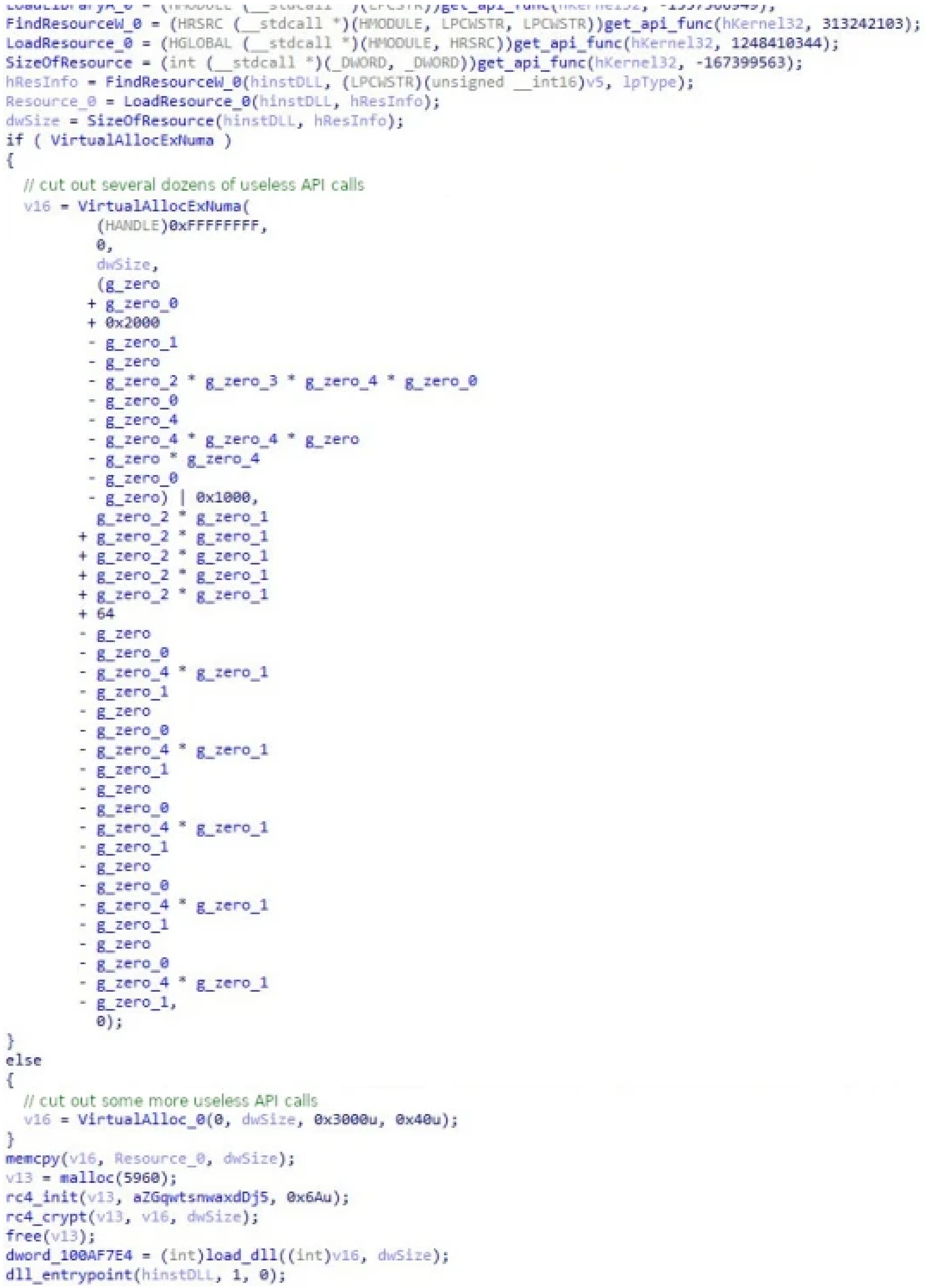


```
hKernel32 = sub_10006669(22, 789, 332, 8893, 22, 'M', v19);
v15 = sub_10006669(778, 443, 778, 99, 54, 77, v21);
hMsvcrt = sub_10006669(64, 54, 99, 55, 43, 997, v20);
malloc = get_api_func(hMsvcrt, 1468035271);
realloc = get_api_func(hMsvcrt, 1489531075);
free = get_api_func(hMsvcrt, -886283742);
qsort = get_api_func(hMsvcrt, -442928783);
bsearch = get_api_func(hMsvcrt, 1109963343);
memcpy = get_api_func(hMsvcrt, 1502129821);
memset = get_api_func(hMsvcrt, 1496363672);
VirtualAlloc = get_api_func(hKernel32, 1832061817);
VirtualAllocExNuma = get_api_func(hKernel32, 1407361980);
VirtualQuery = get_api_func(hKernel32, 2135689679);
VirtualFree_0 = get_api_func(hKernel32, 512193752);
VirtualProtect = get_api_func(hKernel32, 1426588992);
GetProcAddress_0 = get_api_func(hKernel32, 1473199063);
FreeLibrary_0 = get_api_func(hKernel32, 701175500);
GetNativeSystemInfo = get_api_func(hKernel32, -397615486);
RtlAllocateHeap = get_api_func(v15, 1234854987);
HeapFree_0 = get_api_func(hKernel32, 1279617859);
GetProcesHeap = get_api_func(hKernel32, -1018156069);
IsBadReadPtr_0 = get_api_func(hKernel32, 1250200167);
LoadLibraryA_0 = get_api_func(hKernel32, -946891821);
api_func = get_api_func(hKernel32, 923917231);
FindResourceW_0 = api_func;
LoadResource_0 = get_api_func(hKernel32, 1859085472);
SizeOfResource = get_api_func(hKernel32, 443275565);
v9 = (api_func)(hinstDLL, 992, &unk_10053A48, v13, v14, v3);
hNtdll = LoadResource_0(hinstDLL, v9);
v10 = SizeOfResource(hinstDLL, v9);
if ( VirtualAllocExNuma )
  v11 = VirtualAllocExNuma(0xFFFFFFFF, 0, v10, 0x3000u, 0x40u, 0);
else
  v11 = VirtalAlloc(0, v10, 12288, 64);
hModule = v11;
memcpy(v11, hNtdll, v10);
hNtdlla = malloc(8878);
rc4_init(hNtdlla);
rc4_crypt(hNtdlla, hModule, v10);
free(hNtdlla);
dword_100624CC = load_dll(hModule, v10);
dll_entrypoint(hinstDLL, 1, 0);
return 1;
```

RC4 decryption routine of a recent Epoch 4 sample



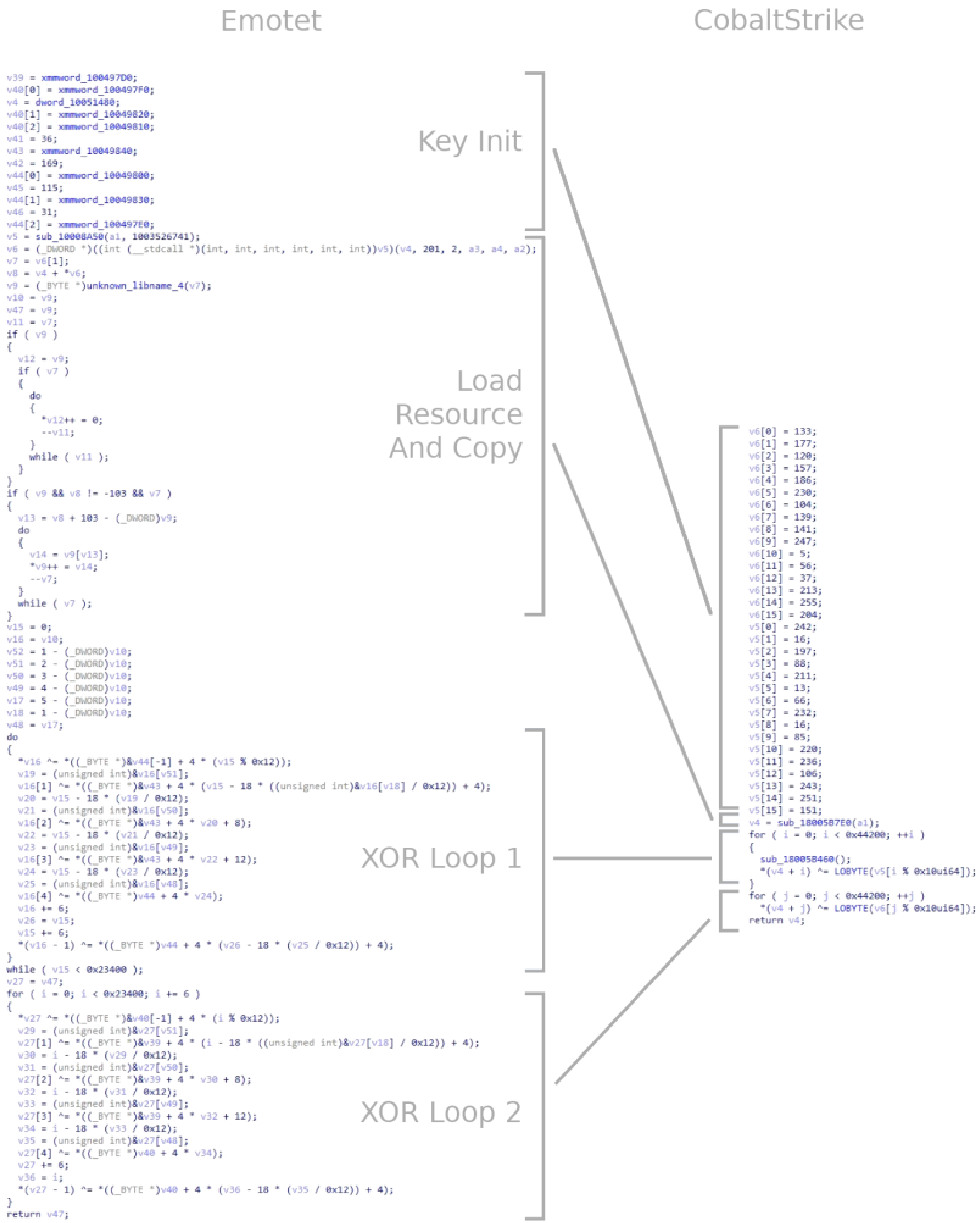
```
hKernel32 = sub_1007C5C0(243, 45, 22, 789, 332, 8893, 22, 77, (int)v4);
hNtdll = sub_1007C5C0(343, 112, 778, 443, 778, 99, 54, 77, (int)v11);
hMsvcrt = sub_1007C5C0(32, 6354, 64, 54, 99, 55, 43, 997, (int)v12);
malloc = (int (__cdecl *)(_DWORD))get_api_func(hMsvcrt, 857360143);
realloc = get_api_func(hMsvcrt, 878855947);
free = (int (__cdecl *)(_DWORD))get_api_func(hMsvcrt, -1496958870);
qsort = get_api_func(hMsvcrt, -1053603911);
bsearch = get_api_func(hMsvcrt, 499288215);
memcpy = (int (__cdecl *)(_DWORD, _DWORD, _DWORD))get_api_func(hMsvcrt, 891454693);
memset = get_api_func(hMsvcrt, 885688544);
VirtualAlloc_0 = (LPVOID (__stdcall *)(LPVOID, SIZE_T, DWORD, DWORD))get_api_func(hKernel32, 1221386689);
VirtualAllocExNuma = (LPVOID (__stdcall *)(HANDLE, LPVOID, SIZE_T, DWORD, DWORD, DWORD))get_api_func(
                                                                                         hKernel32,
                                                                                         796686852);
VirtualQuery = (SIZE_T (__stdcall *)(LPCVOID, PMEMORY_BASIC_INFORMATION, SIZE_T))get_api_func(hKernel32, 1525014551);
VirtualFree_0 = (BOOL (__stdcall *)(LPVOID, SIZE_T, DWORD))get_api_func(hKernel32, -98481376);
VirtualProtect = (BOOL (__stdcall *)(LPVOID, SIZE_T, DWORD, PDWORD))get_api_func(hKernel32, 815913864);
GetProcAddress_0 = (FARPROC (__stdcall *)(HMODULE, LPCSTR))get_api_func(hKernel32, 862523935);
FreeLibrary_0 = (BOOL (__stdcall *)(HMODULE))get_api_func(hKernel32, 90500372);
GetNativeSystemInfo = (void (__stdcall *)(LPSYSTEM_INFO))get_api_func(hKernel32, -1008290614);
RtlAllocateHeap = (int (__stdcall *)(_DWORD))get_api_func(hNtdll, 624179859);
HeapFree_0 = (BOOL (__stdcall *)(HANDLE, DWORD, LPVOID))get_api_func(hKernel32, 668942731);
GetProcessHeap_0 = (HANDLE (__stdcall *)())get_api_func(hKernel32, -1628831197);
IsBadReadPtr = (BOOL (__stdcall *)(const void *, UINT_PTR))get_api_func(hKernel32, 639525039);
LoadLibraryA_0 = (HMODULE (__stdcall *)(LPCSTR))get_api_func(hKernel32, -1557566940);
```



```
LoadLibraryA_0 = (HMODULE (__stdcall *)(LPCSTR))get_api_func(hKernel32, -1557300949);
FindResourceW_0 = (HRSRC (__stdcall *)(HMODULE, LPCWSTR, LPCWSTR))get_api_func(hKernel32, 313242103);
LoadResource_0 = (HGLOBAL (__stdcall *)(HMODULE, HRSRC))get_api_func(hKernel32, 1248410344);
SizeOfResource = (int (__stdcall *)(_DWORD, _DWORD))get_api_func(hKernel32, -167399563);
hResInfo = FindResourceW_0(hinstDLL, (LPCWSTR)(unsigned __int16)v5, lpType);
Resource_0 = LoadResource_0(hinstDLL, hResInfo);
dwSize = SizeOfResource(hinstDLL, hResInfo);
if ( VirtualAllocExNuma )
{
  // cut out several dozens of useless API calls
  v16 = VirtualAllocExNuma(
          (HANDLE)0xFFFFFFFF,
          0,
          dwSize,
          (g_zero
         + g_zero_0
         + 0x2000
         - g_zero_1
         - g_zero
         - g_zero_2 * g_zero_3 * g_zero_4 * g_zero_0
         - g_zero_0
         - g_zero_4
         - g_zero_4 * g_zero_4 * g_zero
         - g_zero * g_zero_4
         - g_zero_0
         - g_zero) | 0x1000,
          g_zero_2 * g_zero_1
        + g_zero_2 * g_zero_1
        + g_zero_2 * g_zero_1
        + g_zero_2 * g_zero_1
        + g_zero_2 * g_zero_1
        + 64
        - g_zero
        - g_zero_0
        - g_zero_4 * g_zero_1
        - g_zero_1
        - g_zero
        - g_zero_0
        - g_zero_4 * g_zero_1
        - g_zero_1
        - g_zero
        - g_zero_0
        - g_zero_4 * g_zero_1
        - g_zero_1
        - g_zero
        - g_zero_0
        - g_zero_4 * g_zero_1
        - g_zero_1
        - g_zero
        - g_zero_0
        - g_zero_4 * g_zero_1
        - g_zero_1,
          0);
}
else
{
  // cut out some more useless API calls
  v16 = VirtualAlloc_0(0, dwSize, 0x3000u, 0x40u);
}
memcpy(v16, Resource_0, dwSize);
v13 = malloc(5960);
rc4_init(v13, aZGqwtsnwaxdDj5, 0x6Au);
rc4_crypt(v13, v16, dwSize);
free(v13);
dword_100AF7E4 = (int)load_dll((int)v16, dwSize);
dll_entrypoint(hinstDLL, 1, 0);
```

RC4 decryption routine of a recent Epoch 5 sample Emotet RC4 Decrypt for Payload – Epoch 4 (left) vs Epoch 5 (right)

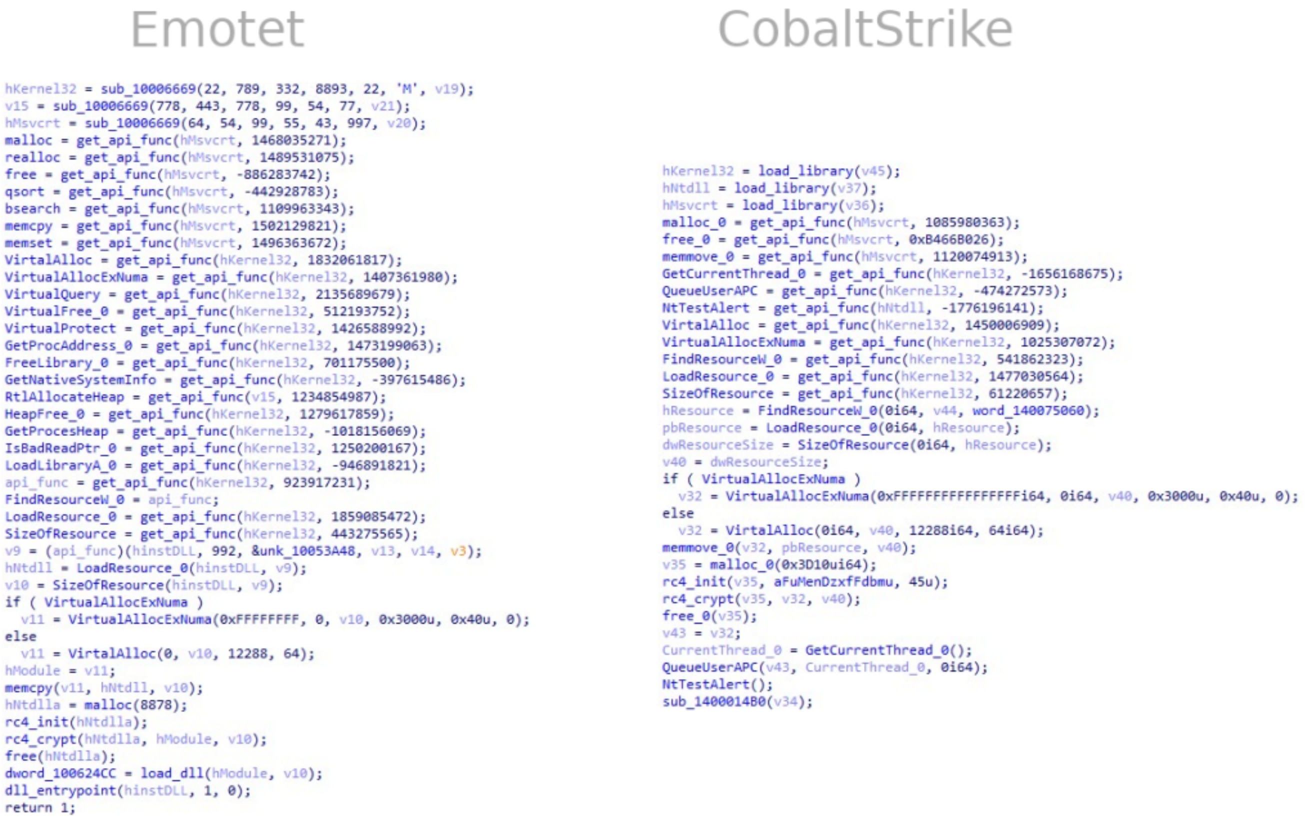
The surprising discovery we made during the week preceeding the publication of this post is related to the CobaltStrike drops. Assuming from what was observed for Epochs 1 through 3, thoughts were that some other party paid the Emotet operators to drop CobaltStrike as their desired payload. Having a closer look at the samples reveals an interesting observation: all of the CobaltStrike drops used packing stubs which looked extremely familiar. The drops referred to in the following were received on March 11th, however, these specific packing stubs were already observed earlier for Emotet drops. Unfortunately, we did not see the connection until a couple of days ago. But have a look for yourself:



```
v39 = xmmword_100497D0;
v40[0] = xmmword_100497F0;
v4 = dword_10051480;
v40[1] = xmmword_10049820;
v40[2] = xmmword_10049810;
v41 = 36;
v43 = xmmword_10049840;
v42 = 169;
v44[0] = xmmword_10049800;
v45 = 115;
v44[1] = xmmword_10049830;
v46 = 31;
v44[2] = xmmword_100497E0;
v5 = sub_10008A50(a1, 1003526741);
v6 = (_DWORD *)((int (__stdcall *)(int, int, int, int, int, int))v5)(v4, 201, 2, a3, a4, a2);
v7 = v6[1];
v8 = v4 + *v6;
v9 = (_BYTE *)unknown_libname_4(v7);
v10 = v9;
v47 = v9;
v11 = v7;
if ( v9 )
{
  v12 = v9;
  if ( v7 )
  {
    do
    {
      *v12++ = 0;
      --v11;
    }
    while ( v11 );
  }
}
if ( v9 && v8 != -103 && v7 )
{
  v13 = v8 + 103 - (_DWORD)v9;
  do
  {
    v14 = v9[v13];
    *v9++ = v14;
    --v7;
  }
  while ( v7 );
}
v15 = 0;
v16 = v10;
v52 = 1 - (_DWORD)v10;
v51 = 2 - (_DWORD)v10;
v50 = 3 - (_DWORD)v10;
v49 = 4 - (_DWORD)v10;
v17 = 5 - (_DWORD)v10;
v18 = 1 - (_DWORD)v10;
v48 = v17;
do
{
  *v16 ^= *((_BYTE *)&v44[-1] + 4 * (v15 % 0x12));
  v19 = (unsigned int)&v16[v51];
  v16[1] ^= *((_BYTE *)&v43 + 4 * (v15 - 18 * ((unsigned int)&v16[v18] / 0x12)) + 4);
  v20 = v15 - 18 * (v19 / 0x12);
  v21 = (unsigned int)&v16[v50];
  v16[2] ^= *((_BYTE *)&v43 + 4 * v20 + 8);
  v22 = v15 - 18 * (v21 / 0x12);
  v23 = (unsigned int)&v16[v49];
  v16[3] ^= *((_BYTE *)&v43 + 4 * v22 + 12);
  v24 = v15 - 18 * (v23 / 0x12);
  v25 = (unsigned int)&v16[v48];
  v16[4] ^= *((_BYTE *)v44 + 4 * v24);
  v16 += 6;
  v26 = v15;
  v15 += 6;
  *(v16 - 1) ^= *((_BYTE *)v44 + 4 * (v26 - 18 * (v25 / 0x12)) + 4);
}
while ( v15 < 0x23400 );
v27 = v47;
for ( i = 0; i < 0x23400; i += 6 )
{
  *v27 ^= *((_BYTE *)&v40[-1] + 4 * (i % 0x12));
  v29 = (unsigned int)&v27[v51];
  v27[1] ^= *((_BYTE *)&v39 + 4 * (i - 18 * ((unsigned int)&v27[v18] / 0x12)) + 4);
  v30 = i - 18 * (v29 / 0x12);
  v31 = (unsigned int)&v27[v50];
  v27[2] ^= *((_BYTE *)&v39 + 4 * v30 + 8);
  v32 = i - 18 * (v31 / 0x12);
  v33 = (unsigned int)&v27[v49];
  v27[3] ^= *((_BYTE *)&v39 + 4 * v32 + 12);
  v34 = i - 18 * (v33 / 0x12);
  v35 = (unsigned int)&v27[v48];
  v27[4] ^= *((_BYTE *)v40 + 4 * v34);
  v27 += 6;
  v36 = i;
  *(v27 - 1) ^= *((_BYTE *)v40 + 4 * (v36 - 18 * (v35 / 0x12)) + 4);
}
return v47;
```

```
v6[0] = 133;
v6[1] = 177;
v6[2] = 120;
v6[3] = 157;
v6[4] = 186;
v6[5] = 230;
v6[6] = 104;
v6[7] = 139;
v6[8] = 141;
v6[9] = 247;
v6[10] = 5;
v6[11] = 56;
v6[12] = 37;
v6[13] = 213;
v6[14] = 255;
v6[15] = 204;
v5[0] = 242;
v5[1] = 16;
v5[2] = 197;
v5[3] = 88;
v5[4] = 211;
v5[5] = 13;
v5[6] = 66;
v5[7] = 232;
v5[8] = 16;
v5[9] = 85;
v5[10] = 220;
v5[11] = 236;
v5[12] = 106;
v5[13] = 243;
v5[14] = 251;
v5[15] = 151;
v4 = sub_1800587E0(a1);
for ( i = 0; i < 0x44200; ++i )
{
  sub_180058460();
  *(v4 + i) ^= LOBYTE(v5[i % 0x10ui64]);
}
for ( j = 0; j < 0x44200; ++j )
  *(v4 + j) ^= LOBYTE(v6[j % 0x10ui64]);
return v4;
```

XOR Decrypt of Payload – Emotet (left) vs CobaltStrike Drop A (right)



```
hKernel32 = sub_10006669(22, 789, 332, 8893, 22, 'M', v19);
v15 = sub_10006669(778, 443, 778, 99, 54, 77, v21);
hMsvcrt = sub_10006669(64, 54, 99, 55, 43, 997, v20);
malloc = get_api_func(hMsvcrt, 1468035271);
realloc = get_api_func(hMsvcrt, 1489531075);
free = get_api_func(hMsvcrt, -886283742);
qsort = get_api_func(hMsvcrt, -442928783);
bsearch = get_api_func(hMsvcrt, 1109963343);
memcpy = get_api_func(hMsvcrt, 1502129821);
memset = get_api_func(hMsvcrt, 1496363672);
VirtalAlloc = get_api_func(hKernel32, 1832061817);
VirtualAllocExNuma = get_api_func(hKernel32, 1407361980);
VirtualQuery = get_api_func(hKernel32, 2135689679);
VirtualFree_0 = get_api_func(hKernel32, 512193752);
VirtualProtect = get_api_func(hKernel32, 1426588992);
GetProcAddress_0 = get_api_func(hKernel32, 1473199063);
FreeLibrary_0 = get_api_func(hKernel32, 701175500);
GetNativeSystemInfo = get_api_func(hKernel32, -397615486);
RtlAllocateHeap = get_api_func(v15, 1234854987);
HeapFree_0 = get_api_func(hKernel32, 1279617859);
GetProcesHeap = get_api_func(hKernel32, -1018156069);
IsBadReadPtr_0 = get_api_func(hKernel32, 1250200167);
LoadLibraryA_0 = get_api_func(hKernel32, -946891821);
api_func = get_api_func(hKernel32, 923917231);
FindResourceW_0 = api_func;
LoadResource_0 = get_api_func(hKernel32, 1859085472);
SizeOfResource = get_api_func(hKernel32, 443275565);
v9 = (api_func)(hinstDLL, 992, &unk_10053A48, v13, v14, v3);
hNtdll = LoadResource_0(hinstDLL, v9);
v10 = SizeOfResource(hinstDLL, v9);
if ( VirtualAllocExNuma )
  v11 = VirtualAllocExNuma(0xFFFFFFFF, 0, v10, 0x3000u, 0x40u, 0);
else
  v11 = VirtalAlloc(0, v10, 12288, 64);
hModule = v11;
memcpy(v11, hNtdll, v10);
hNtdlla = malloc(8878);
rc4_init(hNtdlla);
rc4_crypt(hNtdlla, hModule, v10);
free(hNtdlla);
dword_100624CC = load_dll(hModule, v10);
dll_entrypoint(hinstDLL, 1, 0);
return 1;
```

```
hKernel32 = load_library(v45);
hNtdll = load_library(v37);
hMsvcrt = load_library(v36);
malloc_0 = get_api_func(hMsvcrt, 1085980363);
free_0 = get_api_func(hMsvcrt, 0xB466B026);
memmove_0 = get_api_func(hMsvcrt, 1120074913);
GetCurrentThread_0 = get_api_func(hKernel32, -1656168675);
QueueUserAPC = get_api_func(hKernel32, -474272573);
NtTestAlert = get_api_func(hNtdll, -1776196141);
VirtalAlloc = get_api_func(hKernel32, 1450006909);
VirtualAllocExNuma = get_api_func(hKernel32, 1025307072);
FindResourceW_0 = get_api_func(hKernel32, 541862323);
LoadResource_0 = get_api_func(hKernel32, 1477030564);
SizeOfResource = get_api_func(hKernel32, 61220657);
hResource = FindResourceW_0(0i64, v44, word_140075060);
pbResource = LoadResource_0(0i64, hResource);
dwResourceSize = SizeOfResource(0i64, hResource);
v40 = dwResourceSize;
if ( VirtualAllocExNuma )
  v32 = VirtualAllocExNuma(0xFFFFFFFFFFFFFFFFi64, 0i64, v40, 0x3000u, 0x40u, 0);
else
  v32 = VirtalAlloc(0i64, v40, 12288i64, 64i64);
memmove_0(v32, pbResource, v40);
v35 = malloc_0(0x3D10ui64);
rc4_init(v35, aFuMenDzxfFdbmu, 45u);
rc4_crypt(v35, v32, v40);
free_0(v35);
v43 = v32;
CurrentThread_0 = GetCurrentThread_0();
QueueUserAPC(v43, CurrentThread_0, 0i64);
NtTestAlert();
sub_1400014B0(v34);
```

RC4 Decrypt of Payload – Emotet (left) vs CobaltStrike Drop B (right) Decryption Routines for Payloads

As it can be seen in both examples, Drop A used the packer which was observed in the early days after the rebirth while Drop B used the same packer as the Emotet core itself at the time of writing this post.

Conclusion or (Educated) Guessing

Prior to the rebirth, drops were not bound to the operation of Emotet – the botnet was known to drop whatever their operator's customers paid them for; but since the resurrection, this seems to have shifted towards drops which are very tightly-bound to the Emotet core and thus the operation as well. Considering that Trickbot was used to revive the Emotet botnet back in november 2021 and the observation that Emotet since then only dropped CobaltStrike beacons to infected machines, one thought may arise: have the Trickbot operators perhaps invited their old friends from Emotet over to work for the Conti group as well? It has long been said that the Emotet operators are closely related to the Trickbot group because of their long-running partnership. The thought is also supported by information from the Conti playbook leak in 2021 where it can be seen that Conti makes heavy use of CobaltStrike as a reconnaissance tools before deploying their ransomware. AdvIntel also suspected that Emotet arose as part of the Conti group. The now-discovered use of identical packers for both the Emotet core and the CobaltStrike drops supports the claim in a fascinating way.

Alternatively, or additionally, the resurrection of Emotet may have been the final step in replacing Trickbot as the initial foothold of the Conti group in their victim’s networks by putting their remaining Trickbot bots to a last use. It cannot be denied that Emotet was a surprisingly efficient malware so the Conti operators may have gone for using both Emotet and BazarLoader to access their victim’s networks: with the Trickbot developers focusing solely on BazarLoader and the Emotet operators back into the business, this leaves the Conti group with two independent and powerful tools to access infected machines.

Remarks

Of course, at the same time the author made the aforementioned discovery, researchers observed another drop being delivered by Emotet: SystemBC. It remains to be seen whether this was a one-time delivery in the sense of a test or if researcher will see this drop more often in the future.

Reference Samples

c7574aac7583a5bdc446f813b8e347a768a9f4af858404371eae82ad2d136a01 – old Emotet Epoch 4 sample (2021-11-15)

1c9f611ce78ab0efd09337c06fd8c65b926ebe932bc91b272e97c6b268ab13a1 – old Emotet Epoch 5 sample (2021-11-18)

8494831bbfab5beb6a58d1370ac82a4b3caa1f655b78678c57ef93713c476f9c – recent Emotet Epoch 4 sample (2022-03-14)

31f7e5398c41d7eb8d033dbc7d3b90a2daf54995e20b5ab4a72956b41c8e1455 – recent Emotet Epoch 5 sample (2022-03-15)

cf7a53b0e07f4a1fabc40a5e711cf423d18db685ed4b3c6c87550fcbc5d1a036 – CobaltStrike Drop A (2022-03-11)

73aba991054b1dc419e35520c2ce41dc263ff402bcbbdcbe1d9f31e50937a88e – CobaltStrike Drop B (2022-03-11)