

Interesting FormBook Crypter - unconventional way to store encrypted data

Published: 2020-11-05 · Archived: 2026-04-02 11:20:56 UTC

This FORMBOOK CRYPTER loader contain a lot of interesting feature to bypassed sandbox, obfuscate its code and many more. It also show a unique way to store and parse its encrypted data to execute.

so let's start :).

FORMBOOK CRYPTER LOADER (ANTI-VM):

After decrypting some shellcode in the memory it will use several technique to check if its code is running in a virtual machine or not. The screenshot below show 3 techniques it use.

- ANTI-VM I : it use the cpuid with EAX=0x40000000 as input to determined the hypervisor brandname to check if it is running in a virtualize environment
- ANTI-VM II: use cpuid with EAX=1 as an input to check the 31st bit of its return value in ECX if set or not. if it is set then it is in VM.
- ANTI-VM III: check the existence of some known driver component of the virtual machine. for this example it checks the existence of the vmmouse driver in the machine.


```

v19 = 's';
v20 = 'a';
v21 = 'm';
v22 = 'p';
v23 = 'l';
v24 = 'e';
v25 = '\\0';
v11 = 's';
v12 = 'a';
v13 = 'n';
v14 = 'd';
v15 = 'b';
v16 = 'o';
v17 = 'x';
v18 = '\\0';
v3 = 'm';
v4 = 'a';
v5 = 'l';
v7 = 'a';
v8 = 'r';
v9 = 'e';
v10 = '\\0';
v6 = 'w';
result = 0;
if ( (*(int (__stdcall **)(_DWORD, char *, int))(a1 + 76))(0, v2, 260) )
{
    sub_2879(v2);
    if ( ((int (*)(void))sub_6BBA)()
        || ((int (__stdcall *)(char *))sub_6BBA)(v2)
        || ((int (__stdcall *)(char *))sub_6BBA)(v2) )
    {
        result = 1;
    }
}

```

ANTI-SANDBOX 1

```

seg000:00004866 55      push   ebp
seg000:00004867 8B EC   mov    ebp, esp
seg000:00004869 83 EC 0C   sub    esp, 0Ch
seg000:0000486C 8D 45 F4   lea   eax, [ebp+var_C]
seg000:0000486F 50      push   eax
seg000:00004870 8B 45 08   mov    eax, [ebp+arg_0]
seg000:00004873 C7 45 F4 73 62 69 65   mov    dword ptr [ebp+var_C], 'eibs'
seg000:0000487A C7 45 F8 64 6C 6C 2E   mov    dword ptr [ebp+var_C+4], '.lld'
seg000:00004881 C7 45 FC 64 6C 6C 00   mov    dword ptr [ebp+var_C+8], 'lld'
seg000:00004888 FF 50 78   call  dword ptr [eax+78h]
seg000:0000488B F7 D8   neg    eax
seg000:0000488D 1B C0   sbb   eax, eax
seg000:0000488F F7 D8   neg    eax
seg000:00004891 C9      leave
seg000:00004892 C3      retn

```

ANTI-SANDBOX 2

figure 2: Anti-Sandbox technique

FORMBOOK CRYPTER LOADER (PROCESS CHECK):

It also enumerate all the process running to the machine and try to check the existence of known debugging tools process if it is exist, if yes exit the process. For AV related process and services, it tries to create a counter how many AV product it saw in the machine max of 2 (it seems like it checks for a testing machine that contain several AV product on it).

below is the list of the process it checks related to malware analysis tools and AV product:



figure 3: Process checking to evade malware lab environment

DECRYPTING THE FORMBOOK IN RSRC:

The next thing it will do is to decrypt the encrypted Formbook malware in its resource section. It is done by looking to 2 entry in rsrc section. The first entry is with rsrc ID "14d" with rsrc type of 17 "RT_DLGINCLUDE" that contain the 16 bytes rc4 key to decrypt the rc4 key to decrypt the FORMBOOK.

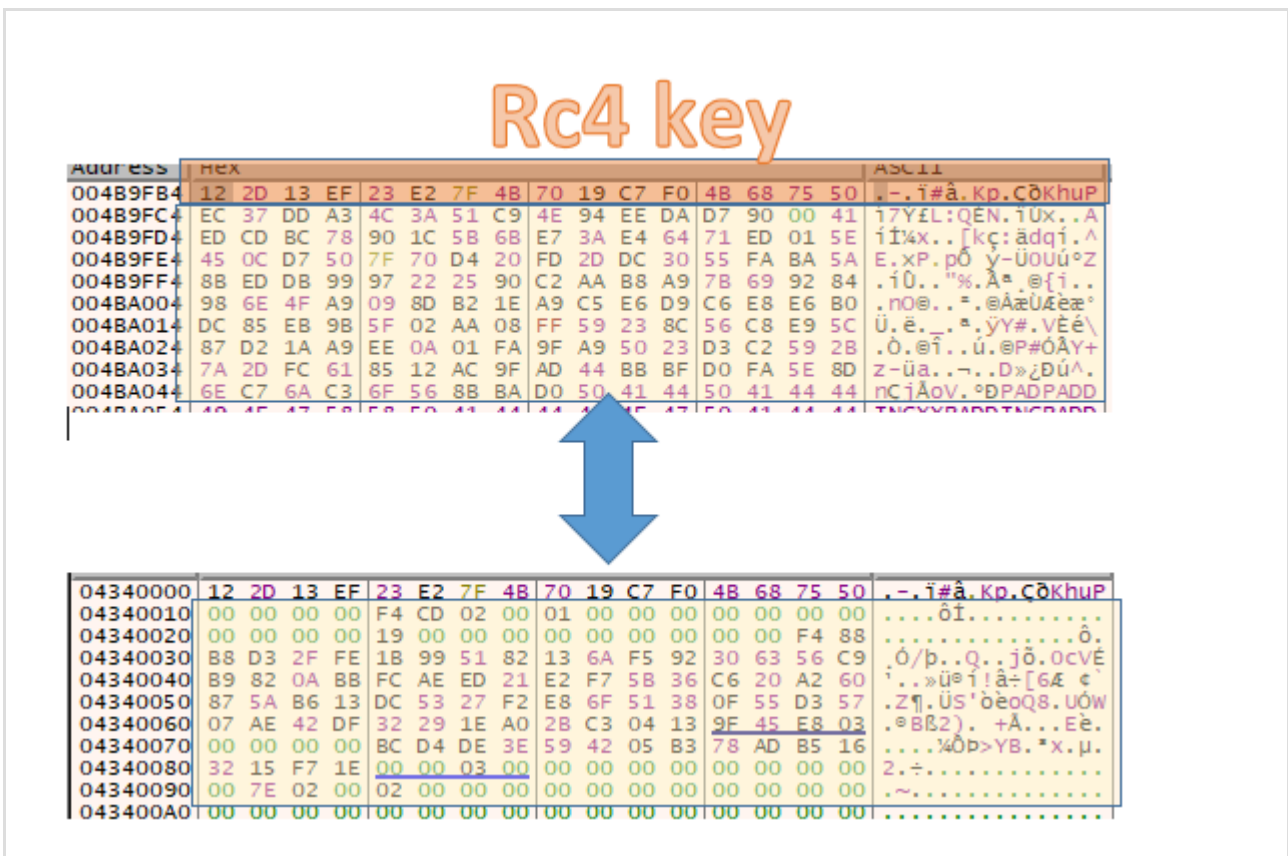


figure 4: decrypting the rc4 key for FORMBOOK

Once the Rc4 key was parse, it will decrypt the encrypted formbook malware, it will load another resource entry with rsrc id "3e8" type "2". Then it will remove 3 dummy bytes to the data blob before decrypting it using rc4 algorithm.

Address	Hex	ASCII
02910000	E8 B1 B1 B1 32 21 69 78 51 DA 6C 88 61 40 E1 09	e+++2!i{Q0l.a@a.
02910010	0D 91 15 36 4F EF CC 29 83 55 B8 EE 98 B1 B1 B1	...60iI).U.i.+++
02910020	89 2F 3E 87 2A 55 D5 04 86 78 EC 27 FO FA 55 B1	./>.*U0..xi'0uU±
02910030	FA A8 6C 6A DE E0 10 21 C8 B1 B1 B1 FB 42 71 98	ü ljpa. !E+++üBq.
02910040	E5 68 ED FD 37 55 42 A7 D5 8F E3 8D E0 DF D0 4A	ähly7UB§0.ä.aB§D
02910050	61 60 D1 7D D1 B1 B1 B1 38 61 67 78 4F 34 86 2D	a`N)N+++8agx0.-
02910060	CE FA EC 4D 04 4B 76 48 74 4A 21 A9 89 A4 30 28	IúIM.KvHTJ!@.0(
02910070	38 B1 B1 B1 0D 6C C2 B5 8E 21 98 7F 04 E1 A7 67	8+++lÄµ.!.!..ásg
02910080	BE 3E 8D E6 72 EF 41 AE 9E 8A B7 4B 20 B1 B1 B1	%>.æriAe...K +++
02910090	84 A1 21 05 2C 41 07 3A F3 FE 16 2D BB 38 38 8A	.i!.,A.:óp.->88.
029100A0	86 BA 7C 64 5C 54 5B BE 53 B1 B1 B1 58 D0 FB 71	.° d\T[%S+++XDuq
029100B0	DD	
029100C0	71	
029100D0	82	
029100E0	EE	
029100F0	96	
02910100	A7	
02910110	54	
02910120	00	
02910130	00	
02910140	00	
02910150	00	
02910160	E0	
02910170	76	
02910180	40	
02910190	04	

Rc4 key

BC D4 DE 3E 59 42 05 B3 78 AD B5 16 32 15 F7 1E %0p>YB.*x.mu.2.-.

Address	Hex	ASCII
02970000	4D 5A 45 52 E8 00 00 00 00 58 83 E8 09 8B C8 83	MZERè....X
02970010	C0 3C 8B 00 03 C1 83 C0 28 03 08 FF E1 90 00 00	A<...A.A(.
02970020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!l.Li!Th
02970050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
02970060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
02970070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.....
02970080	38 2D 5F B9 7C 4C 31 EA 7C 4C 31 EA 7C 4C 31 EA	8-_' Liè Liè Liè
02970090	13 3A 9A EA 37 4C 31 EA 13 3A AF EA 7F 4C 31 EA	..:è7Liè.:è.Liè
029700A0	13 3A AC EA 7D 4C 31 EA 52 69 63 68 7C 4C 31 EA	..:-è}LièRich Liè
029700B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
029700C0	50 45 00 00 4C 01 01 00 9A 4E 59 3A 00 00 00 00	PE..L....NY:....
029700D0	00 00 00 00 E0 00 02 01 0B 01 0A 00 00 6C 02 00	a.....l.
029700E0	00 00 00 00 00 00 00 00 E0 C9 01 00 00 10 00 00af.
029700F0	00 80 02 00 00 00 40 00 00 10 00 00 00 02 00 00@.
02970100	05 00 01 00 00 00 00 00 05 00 01 00 00 00 00 00@.
02970110	00 80 02 00 00 02 00 00 00 00 00 00 02 00 40 81@.
02970120	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00
02970130	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
02970140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02970190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

figure 5: decrypting FORMBOOK

FORMBOOK MZ HEADER:

```
.00400000: 4D          dec     ebp
.00400001: 5A          pop     edx
.00400002: 45          inc     ebp
.00400003: 52          push   edx
.00400004: E8000000   call   .000400009 --↓1
.00400009: 58          1pop   eax
.0040000A: 83E809     sub     eax,9
.0040000D: 8BC8       mov     ecx,eax
.0040000F: 83C03C     add     eax,03C ; '<'
.00400012: 8B00       mov     eax,[eax]
.00400014: 03C1       add     eax,ecx
.00400016: 83C028     add     eax,028 ; '('
.00400019: 0308       add     ecx,[eax]
.0040001B: FFE1       jmp     ecx
```

figure 6: MZ header shellcode

INTERESTING STORING AND PARSING ENCRYPTED DATA:

The Formbook obfuscate its code. One interesting feature of this is how it store and parse its needed bytes to decrypt or to hash to perform its task. Malware commonly used "stack string technique" to initialized its string or data in stack or in an allocated memory space like the screenshot we saw in anti-vm and anti-sandbox headings of this post.

But for this variant it used another technique where it save its needed bytes in a code like structure, then it will parse each instruction to check its opcode if it will passed its requirements, if yes it will parse the operand or opcode that is part of its needed bytes to decrypt or to hash.

requirement:

- I. if opcode is 0x40-0x5f just grab the opcode itself.
- II. if opcode is 0x70-0x7f which is mostly a conditional jump mnemonics then skip that instruction.
- III. (if opcode - 0x40 > 0x1f) and (opcode - 0x70 > 0x0f) then it will check what opcode is that (opcode range from 0x00 to 0xFF) to know what other opcode or how big is the operand it will parsed.

40	inc	AX		50	push	AX		70	jo	rel8
		EAX				EAX		71	jno	rel8
41	inc	CX		51	push	CX		72	jb	rel8
		ECX				ECX			jnae	
42	inc	DX		52	push	DX		72	jc	rel8
		EDX				EDX		73	jae	rel8
43	inc	BX		53	push	BX			jnb	
		EBX				EBX		73	jnc	rel8
44	inc	SP		54	push	SP		74	je	rel8
		ESP				ESP			iz	
45	inc	BP		55	push	BP		75	jne	rel8
		EBP				EBP			jnz	
47	inc	SI		56	push	SI		76	jbe	rel8
		ESI				ESI			jna	
48	dec	AX		57	push	DI		77	ja	rel8
		EAX				EDI			jnbe	
48	inc	DI		58	pop	AX		78	js	rel8
		EDI				EAX		79	ins	rel8
49	dec	CX		59	pop	CX		7A	jp	rel8
		ECX				ECX			jpe	
4A	dec	DX		5A	pop	DX		7B	jnp	rel8
		EDX				EDX			ipo	
4B	dec	BX		5B	pop	BX		7C	jl	rel8
		EBX				EBX			jnge	
4C	dec	SP		5C	pop	SP		7D	jge	rel8
		ESP				ESP			jnl	
4D	dec	BP		5D	pop	BP		7E	jle	rel8
		EBP				EBP			jng	
4E	dec	SI		5E	pop	SI		7F	jg	rel8
		ESI				ESI			jnle	
4F	dec	DI		5F	pop	DI				
		EDI								

figure 7.A: initial opcode it tries to grab and opcode it skip

```
int __cdecl Func_GrabEncryptedData(int destBuff, int VA_41BF1F, unsigned int SizeOfNeededBytes)
{
    _BYTE *VA_41BF1F_1; // eax
    unsigned int NumberOfOpcodeBytesRead; // esi
    int NextOpcode; // edi
    unsigned __int8 Opcode; // [esp+4h] [ebp-Ch]
    unsigned int destBuffPtr; // [esp+8h] [ebp-8h] BYREF
    int ptrNumberOfOpcodereadFromStart; // [esp+Ch] [ebp-4h] BYREF

    VA_41BF1F_1 = sub_412D20(VA_41BF1F);
    if ( *VA_41BF1F_1 != 0x55 || VA_41BF1F_1[1] != 0x8B )
        return 0;
    NumberOfOpcodeBytesRead = 0;
    NextOpcode = (VA_41BF1F_1 + 3);
    destBuffPtr = 0;
    ptrNumberOfOpcodereadFromStart = 0;
    while ( NumberOfOpcodeBytesRead < SizeOfNeededBytes )
    {
        Opcode = *(ptrNumberOfOpcodereadFromStart + NextOpcode);
        if ( (Opcode - 0x40) > 0x1Fu ) // inc-dec opcode
        {
            if ( (Opcode - 0x70) > 0xFu ) // conditional jump opcode
            {
                FuncCheckOpcode(Opcode, destBuff, NextOpcode, &destBuffPtr, &ptrNumberOfOpcodereadFromStart);
                NumberOfOpcodeBytesRead = destBuffPtr;
            }
            else
            {
                ptrNumberOfOpcodereadFromStart += 2;
            }
        }
        else
        {
            Func_GrabNeededOpcode(NumberOfOpcodeBytesRead + destBuff, (NextOpcode + ptrNumberOfOpcodereadFromStart), 1);
            ++NumberOfOpcodeBytesRead;
            ++ptrNumberOfOpcodereadFromStart;
            destBuffPtr = NumberOfOpcodeBytesRead;
        }
    }
    return destBuff;
}
```

figure 7.B: FormBook opcode condition for parsing its data



figure 8: the parse stored data that either to be decrypt or hash it.

And also not all stored data that it will parse to its code will be decrypted, some of those stored data is designed to compute sha1 hash that will serve as the decryption key (rc4 algorithm) to decrypt another blob of code.

```

_BYTE *_cdecl Func_DecryptBytesGrabbed(int DestBuff)
{
    int VA_41C3A6; // eax
    int VA_41C50B; // eax
    int VA_41BEF1; // eax
    char dest_buff; // [esp+Ch] [ebp-140h] BYREF
    char v6[215]; // [esp+Dh] [ebp-13Fh] BYREF
    _DWORD sha1_ctx[26]; // [esp+E4h] [ebp-68h] BYREF

    dest_buff = 0;
    Func_MemSet(v6, 0, 0xD4u);
    VA_41C3A6 = sub_41C3A1();
    Func_GrabEncryptedData(&dest_buff, VA_41C3A6 + 2, 0xD3u);
    VA_41C50B = sub_41C506();
    Func_GrabEncryptedData(DestBuff + 0x444, VA_41C50B + 2, 0x2F0u);
}

```

```
VA_41BEF1 = sub_41BEEC();
Func_GrabEncryptedData(DestBuff + 0x7B8, VA_41BEF1 + 2, 0x14u);
Func_SHA1_Init(sha1_ctx);
Func_Sha1_Update(sha1_ctx, &dest_buff, 0xD3);
Func_Sha1_Final(sha1_ctx);
Func_GrabNeededOpcode(DestBuff + 0x7A4, sha1_ctx, 20);
Func_DecryptWithRc4((DestBuff + 0x444), 0x2F0u, DestBuff + 0x7A4);
Func_SHA1_Init(sha1_ctx);
Func_Sha1_Update(sha1_ctx, (DestBuff + 0x7B8), 0x14);
Func_Sha1_Final(sha1_ctx);
Func_DecryptWithRc4((DestBuff + 0x444), 0x2F0u, sha1_ctx);
Func_SHA1_Init(sha1_ctx);
Func_Sha1_Update(sha1_ctx, (DestBuff + 0x444), 752);
Func_Sha1_Final(sha1_ctx);
return Func_DecryptWithRc4((DestBuff + 0x7B8), 0x14u, sha1_ctx);
} tag
```

SAMPLES:

filename: Formbook_loader.bin

md5: 65880d23eb6051a1604707371ebb6d2c

sha1: 3f5d0833adb39715f1d45f1a3c8982c52519bc1

sha256: ac2e9615b368e00fb4bf4d5180bbfc0d6fb7bbce3fa1af603d346d7a8f2450e5

filename: formbook.bin

md5: df93eecd1799f9c9c674b8cdb2f1dad1

sha1: e66c893f39c7553f59a5381d23a5c65e5c2e84f7

sha256: 5d7eba73b4d29ee17529511bb8b0745e658bf2adfcae57bdfa8d0870f4732a18

YARA RULES:

```
import "pe"

rule formbook_loader_crypter {
  meta:
    author = "tcontre"
    description = "detecting formbook-loader-crypter malware"
    date = "2020-11-05"
    sha256 = "ac2e9615b368e00fb4bf4d5180bbfc0d6fb7bbce3fa1af603d346d7a8f2450e5"

  strings:
    $mz = { 4d 5a }
```

```
$dec = { 03 CE 8A 03 88 45 F9 8B C6 51 B9 03 00 00 00 33 D2 F7 F1 59 85 D2 75 14 8A 45 F9 32 45 FA 88 0
$rc4_key = {12 2D 13 EF 23 E2 7F 4B 70 19 C7 F0 4B 68 75 50}

condition:
    ($mz at 0) and ($dec ) or ($rc4_key)

}

rule formbook_crypter {
    meta:
        author = "tccontre"
        description = "detecting formbook-crypter malware"
        date = "2020-11-05"
        sha256 = "5d7eba73b4d29ee17529511bb8b0745e658bf2adfcae57bdfa8d0870f4732a18"

    strings:
        $mz = { 4d 5a }

        $shell = { 4D 5A 45 52 E8 00 00 00 00 58 83 E8 09 8B C8 83 C0 3C 8B 00 03 C1 83 C0 28 03 08 FF E1 90 00
            $opcode_check = {8B 4D FC 8A 04 39 03 CF 88 45 F4 8D 50 C0 80 FA 1F 77 18 6A 01 51 8D 04 1E 50 F

    condition:
        ($mz at 0) and ($shell at 0) or ($opcode_check)

}    tag
```

Source: <https://tccontre.blogspot.com/2020/11/interesting-formbook-crypter.html>