



Writing malware configuration parsers

Mark Lim & Zong-Yu Wu



Mark Lim

- Principal Malware Researcher at Palo Alto Networks
- Based in Singapore
- Love to go for long jogs



Zong-Yu Wu

- Reverse engineer
- Based in London
- Joined Palo Alto Networks since 2022
- Experienced in malware detection and threat intelligence



OUTLINE

- Introduction
- MCE workflow
- Case Study 1 - Guloader
- Case Study 2 - RedLine
- Summary



WARNING!

1. Isolate infected systems: Disconnect from networks and external devices.
2. Utilize virtual machines
3. Backup data: Ensure critical data is safely backed up.
4. You release and hold harmless Palo Alto Networks and Virus Bulletin, its affiliates, and contributors from any liability, claims, or damages arising from handling live computer viruses.



Get the stuff!

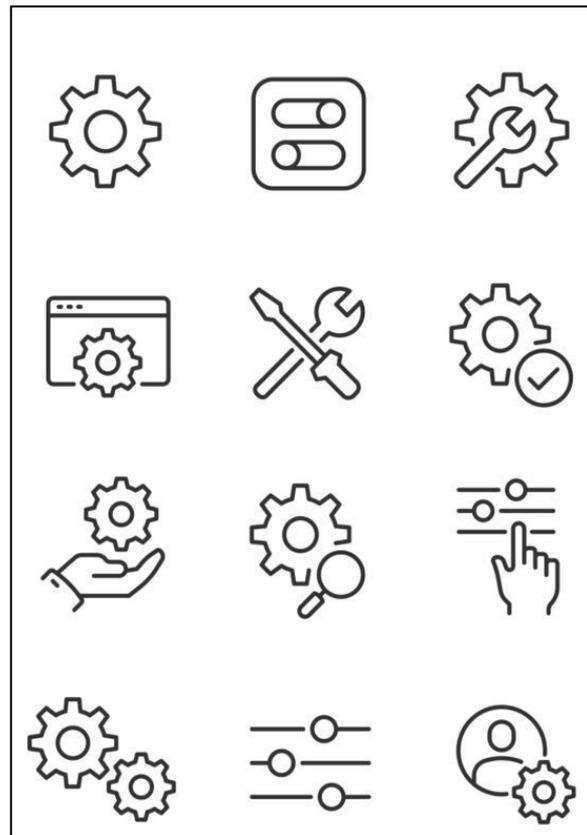
- <https://tinyurl.com/VB2024MCE>



Background

What are malware configurations?

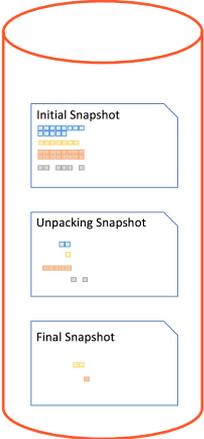
- Similar to 'settings' or 'preferences' in software
- Malware configuration defines the uniqueness of each instance
- C&C addresses, encryption keys, attack parameters and other IOCs
- Tough to obtain statically
- But can be extracted from process memory.



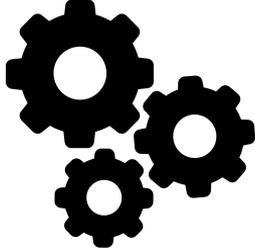
Malware Configuration Extraction Workflow



Sandbox



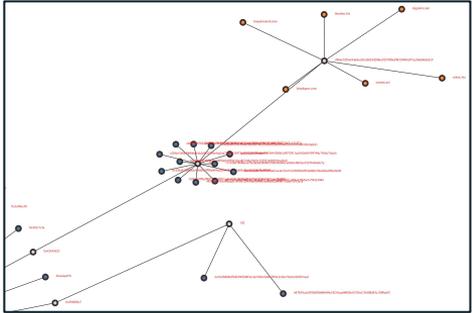
Snapshot Data



Memory Analysis and Malware Configuration Parsing

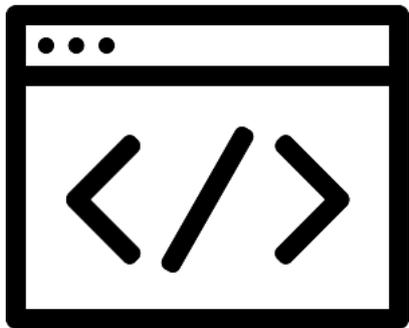


Malware Analysts Create Malware Config Extractors



Malware Configs indexed

Decrypting Malware Configuration



Encryption
routine



Encryption
key



Ciphertext

Case study:
Guloader

Evolutionary Journey Of Guloader's Configuration Tactics

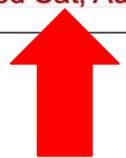
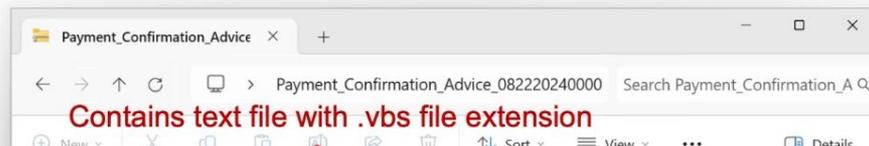
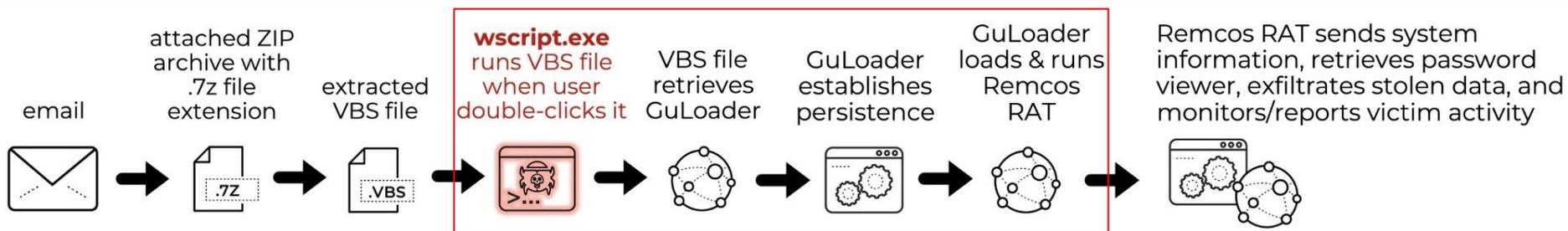
Ciphertext Splitting

Ciphertext has to be decoded in blocks from a function before it can be used



Control flow obfuscation progressively applied to increase the complexity of retrieving the ciphertext

2024 August



https://x.com/Unit42_Intel/status/1828444963001995599

Dump Memory

1. Use System Informer (a.k.a. Process Hacker)
2. VBS -> Ps1 -> wab.exe
3. Look for wab.exe after Guloader is injected
4. Locate the RWX memory pages
5. Dump the memory pages after the sample has detonated

System Informer [PETER-PC\User1] (Administrator)

System View Tools Users Help

Refresh Options Find handles or DLLs System information

Processes Services Network Disk Firewall Devices

Name	PID	C...	User name	Description	Network t...	Bits
wab.exe	78...		PETER-PC\User...	Windows Contacts		32

wab.exe (7808) Properties

General Statistics Performance Threads Token Modules Memory Environment Handles GPU Disk Network Comment Windows

Options Refresh Search Memory (Ctrl+K) Aa * *

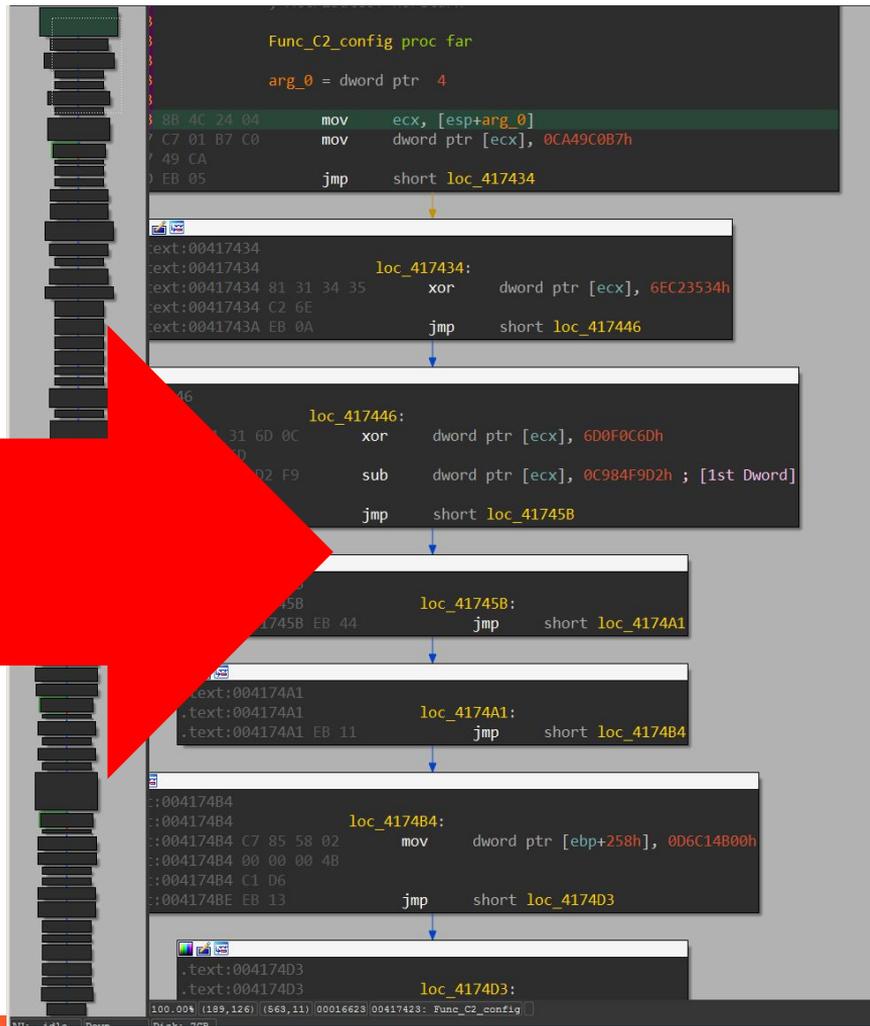
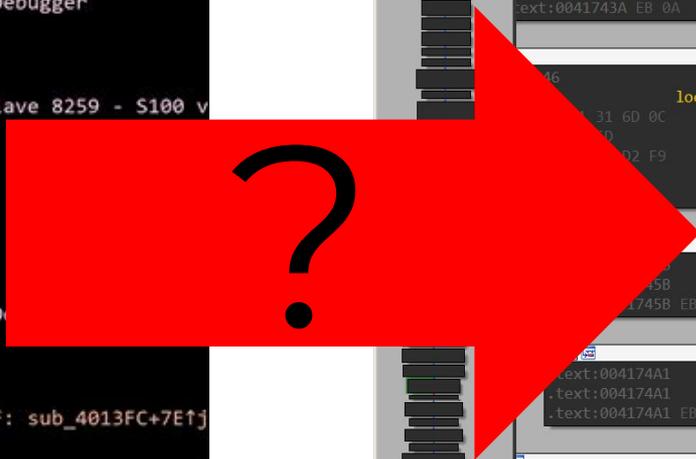
Base address	Type	Size	Pröf...	Use	Total WS	Private ...	Sharea...	Shared...	Locked...
0x6dcf1000	Image: ...	2.04 MB	RX	C:\Windows\SysWOW6...	664 kB		2.04 MB		
0x6dc21000	Image: ...	716 kB	RX	C:\Windows\SysWOW6...	292 kB		716 kB		
0x6da71000	Image: ...	1.2 MB	RX	C:\Windows\SysWOW6...	196 kB		1.2 MB		
0x6d9f1000	Image: ...	444 kB	RX	C:\Windows\SysWOW6...	84 kB		444 kB		
0x6d9c1000	Image: ...	136 kB	RX	C:\Windows\SysWOW6...	28 kB		136 kB		
0xcc1000	Image: ...	12 kB	RX	C:\Program Files (x86)...	12 kB		12 kB		
0x7ffc1384...	Image: ...	20 kB	RWX	C:\Windows\System32\...	20 kB	20 kB			
0x42b0000	Private:...	14.3 MB	RWX		14.3 MB	14.3 MB			

wab.exe (7808) (0x42b0000 - 0x50fc000)

```
00000000 9d 5e b8 ab bb e7 bb 22 ec d2 55 a3 9a 66 f8 .^.....".U..f.
00000010 96 5c 33 de c7 e7 fc 0e e6 52 e7 a5 1e df 2f dc .\3.....R.../.
00000020 a3 af a2 f0 7e 77 e7 a6 58 21 b7 f9 d5 98 e2 0f .....w..X!.....
00000030 f4 6e a1 13 2f 5e c1 17 db 71 00 a0 e4 72 1d d6 ..n..^..g...E..
00000040 4e 53 66 63 05 de 28 de 56 53 66 78 00 d5 48 af NSfc...W3fx..H.
00000050 a3 2a 6b 57 7e 65 94 98 a6 94 e2 9f e5 db 4b 55 .*kM-e.....KU
00000060 22 5e 4a 61 b5 7f e7 fa 0f 5c 66 57 ee 5e 5b a1 ^Ja.....fW..^].
00000070 a3 db 4b 55 22 5e 0b 87 cb f5 e7 d2 0f 5c 66 57 ..KU^.....fW
00000080 14 e4 dd 7f a2 a7 01 16 dd d3 4b 55 22 5e 13 a3 .....KU^..
00000090 1b 87 27 de 97 20 67 57 22 38 e3 87 9c 54 e4 07 ..'.gW8...T..
000000a0 c4 df a0 42 ff af 43 d6 d4 ee a3 92 78 df a0 9e ...B.C.....x..
000000b0 74 26 cf 31 a7 8f 5f a6 a9 eb 18 56 22 5e 69 d2 t6.l.....W^1.
000000c0 4a a1 99 a8 44 db bc de 5f 22 d8 34 22 5e 69 d8 J...D...".4^1.
000000d0 2b 9c 66 57 ab db 1e 56 22 5e a0 44 67 ac 6f +.EW...V^AA.Dg.o
000000e0 fb d7 fb 1b 23 5e 66 04 ca 88 6c 52 22 da ae de ...#E...1R"...
000000f0 67 1a 5e ab a6 b1 8f a2 ea 5a 66 31 1b 9c 00 6e g.^.....Zf1..n
00000100 f8 d1 a3 33 23 5e 66 3f 7d 82 16 3b ca fd b4 53 3^f21...s
```

Control Flow Obfuscation

```
call sub_40D945
mov [ebp+18h], ebx
mov ecx, [ebp+18h]
int 3 ; Trap to Debugger
movsd [esi+6F348E7Bh], cl
int 4Ch ; Z100 - Slave 8259 - S100 v
and dh, [ebx-7465A846h]
movsd sub_40CE8A
call sub_40CE8A
mov [ebp+98h], eax
int 3 ; Trap to D
cmpsd
jle short $+2
loc_40147C: ; CODE XREF: sub_4013FC+7E1j
jb short loc_40140B
test al, 87h
push ds
hlt
```



Diving into implementation of control flow obfuscation

1. SEH vs VEH
2. Locating the Vectored Exception Handler (VEH)
3. No easy way to do it via debugger unlike SEH
4. Locate VEH from NTDLL.dll walking the structure



Diving into implementation of control flow obfuscation

Demo 1.

- Locating VEH via the IDAPython script



Analysing the VEH

1. How VEH handles the exceptions ?
2. How many junk bytes to skip ? (offset)
3. How is the EIP updated ?



Analysing the VEH

```
1  int __stdcall FN_VEH(PEXCEPTION_POINTERS ExceptionInfo)
2  {
3      DWORD ExceptionCode;
4
5      ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
6      if ( ExceptionCode != EXCEPTION_ACCESS_VIOLATION )
7      {
8          if ( ExceptionCode != EXCEPTION_ILLEGAL_INSTRUCTION
9              && ExceptionCode != EXCEPTION_PRIV_INSTRUCTION
10             && ExceptionCode != EXCEPTION_SINGLE_STEP
11             && ExceptionCode != EXCEPTION_BREAKPOINT )
12          {
13              return 0;           // exception NOT handled
14          }
15      FN_VEH_handles_exception:
16          FN_Get_Offset_Decompile_key(ExceptionInfo);
17          FN_Get_Offset_Update_EIP();
18          return -1;           // exception handled
19      }
```

Analysing the VEH

```
pop     eax
xor     dword ptr [ebx], 0D62B9FCCh
int     3 ; EXCEPTION_BREAKPOINT triggered!
-----
db  3,13h,19h,15h,'2',12h,0AFh,'Imp|',0E9h,4Dh,4Dh ; junk bytes
-----
add     dword ptr [ebx], 3FA3DA06h
```

EXCEPTION_ACCESS_VIOLATION

```
adc     byte ptr [ebx-781E6E5Ah], 3Ch ; '<'
test   [ebx-76655A7Bh], edx
; -----
db  65h ; e ; exception triggered!
a4s db '4s',0FDh,0A0h,'%~',12h,'f9',0C8h,5Fh,85h,0CBh,5Ah,5,'N'
db  0C5h,0E5h,1Fh,0F3h,0CCh,0FBh,12h,0Dh,';',0EEh,0E4h,89h,0DCh,4
```

EXCEPTION_PRIV_INSTRUCTION

```
mov     dword ptr [ebx], 0ACDB2BA7h
add     dword ptr [ebx], 9B37543h
xor     dword ptr [ebx], 80317A66h
sub     dword ptr [ebx], 36BFDA32h
sysret ; trigger exception!
; -----
a4h db 'Hz',0,0,0,0,0,0,0,0,0F7h,9Bh,19h,0D7h,'{',13h,1Ch,0A8h
```

EXCEPTION_BREAKPOINT

```
mov     esi, 0D37212A5h
add     esi, 4C655390h
xor     esi, 1FD76635h ; esi=0
mov     [esi], esi ; EXCEPTION_ACCESS_VIOLATION
-----
db  1Ch,0C5h,74h,0C3h,4Ch,0F6h,8Fh,0FBh,0FAh,'36',92h,0Eh,0 ; junk bytes
-----
pop     esi
```

EXCEPTION_ILLEGAL_INSTRUCTION

Analysing the VEH

EXCEPTION_SINGLE_STEP

```
02B99416 ;  
02B99416 push    ecx  
02B99417 mov     eax, 433D4EDBh  
02B9941C xor     eax, 9436930Fh  
02B99421 xor     eax, 0A78DF586h  
02B99428 sub     eax, 392289D1h  
02B9942E sub     eax, 37639D81h          ; eax=0x100  
02B99435 push    ebx  
02B99436 pushf  
02B99437 mov     ebx, esp  
02B99438 or      [ebx], eax          ; enable Trap flag  
02B99439 popf  
02B9943A test    edi, ecx          ; EXCEPTION_SINGLE_STEP triggered!  
02B9943B ;  
02B99419 aW db 'w',6,0B7h,0B6h,0CFh,14h,'\\',0ACh,0A6h,0B8h,'% ',0  
02B99425 ;  
02B99425 cmp     ebx, ecx  
02B99427 pop     ebx  
02B99428 cmp     eax, edx  
02B9942A pop     eax
```

Analysing the VEH

1. How VEH handles the exceptions ?
- 2. How many junk bytes to skip ? (offset)**
3. How is the EIP updated ?



Analysing the VEH

Trigger !

```
047AED76 C7 03 A7 2B DB AC    mov     dword ptr [ebx], 0ACDB2BA7h
047AED7C 81 03 43 75 B3 09    add     dword ptr [ebx], 9B37543h
047AED82 81 33 66 7A 31 80    xor     dword ptr [ebx], 80317A66h
047AED88 81 2B 22 DA BF 36    sub     dword ptr [ebx], 36BFDA32h
047AED8E 0F 07                sysret
;
047AED8E
047AED90 48                db     48h ; H
047AED91 7A                db     7Ah ; z
047AED92 00                db     0
047AED93 00                db     0
047AED94 00                db     0
047AED95 00                db     0
047AED96 00                db     0
047AED97 00                db     0
047AED98 00                db     0
047AED99 00                db     0
047AED9A F7                db     0F7h
047AED9B 9B                db     9Bh
047AED9C 19                db     19h
047AED9D D7                db     0D7h
047AED9E 7B                db     7Bh ; {
047AED9F 13                db     13h
047AEDA0 1C                db     1Ch
047AEDA1 A8                db     0A8h
047AEDA2 4E                db     4Eh ; N
047AEDA3 71                db     71h ; q
;
047AEDA4 81 C3 E4 44 6C 01    add     ebx, 16C44E4h
047AEDA5 81 EB E0 44 6C 01    sub     ebx, 16C44E0h
```

Junk
bytes
???

Updated EIP

Analysing the VEH

```
1  int __stdcall FN_VEH(PEXCEPTION_POINTERS ExceptionInfo)
2  {
3      DWORD ExceptionCode;
4
5      ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
6      if ( ExceptionCode != EXCEPTION_ACCESS_VIOLATION )
7      {
8          if ( ExceptionCode != EXCEPTION_ILLEGAL_INSTRUCTION
9              && ExceptionCode != EXCEPTION_PRIV_INSTRUCTION
10             && ExceptionCode != EXCEPTION_SINGLE_STEP
11             && ExceptionCode != EXCEPTION_BREAKPOINT )
12         {
13             return 0;           // exception NOT handled
14         }
15     FN_VEH_handles_exception:
16         FN_Get_Offset Decode_key(ExceptionInfo);
17         FN_Get_Offset Update_EIP();
18         return -1;           // exception handled
19     }
```

Analysing the VEH

```
1 int __usercall FN_Get_Offset Decode_key@<ecx>(_EXCEPTION_POINTERS *a1@<eax>)  
2 {  
3     PCONTEXT ContextRecord; // eax  
4     int result; // ecx  
5     int count; // edx  
6  
7     ContextRecord = a1->ContextRecord;  
8     result = 0x18;  
9     count = 0;  
10    while ( 1 )  
11    {  
12        count += 4;  
13        if ( *(DWORD *)((char *)&ContextRecord->ContextFlags + count) )// check all HW BP registers are zero  
14            break;  
15        if ( count == 0x18 )  
16            return 0xE1; // return offset decode key  
17    }  
18    return result;  
19 }
```

Analysing the VEH

```
call    FN_Get_Offset_Decode_key
test    cl, cl
cmp     edx, ebx
mov     edx, 63C7099Ch
xor     edx, 6649E730h
test    ch, bh
add     edx, 1691B385h
cmp     ah, ch
add     edx, 0E3DF5E87h    ; 0xB8
cmp     dx, 4070h
add     eax, edx           ; _Context.EIP = _CONTEXT+0xB8
test    dl, cl
mov     edx, [eax]
add     edx, 0Ch           ; Get enc offset*
cmp     dl, bl
call    FN_Get_Offset_Update_EIP
```

Analysing the VEH

Trigger !

0xC

Encrypted
count

Updated EIP

```
047AED76 C7 03 A7 2B DB AC    mov     dword ptr [ebx], 0ACDB2BA7h
047AED7C 81 03 43 75 B3 09    add     dword ptr [ebx], 9B37543h
047AED82 81 33 66 7A 31 80    xor     dword ptr [ebx], 80317A66h
047AED88 81 2B 22 DA BF 36    sub     dword ptr [ebx], 36BFDA32h
047AED8E 0F 07                sysret
047AED8E                ;
047AED90 48                    db     48h ; H
047AED91 7A                    db     7Ah ; z
047AED92 00                    db     0
047AED93 00                    db     0
047AED94 00                    db     0
047AED95 00                    db     0
047AED96 00                    db     0
047AED97 00                    db     0
047AED98 00                    db     0
047AED99 00                    db     0
047AED9A F7                    db     0F7h
047AED9B 98                    db     98h
047AED9C 19                    db     19h
047AED9D D7                    db     0D7h
047AED9E 7B                    db     7Bh ; {
047AED9F 13                    db     13h
047AEDA0 1C                    db     1Ch
047AEDA1 A8                    db     0A8h
047AEDA2 4E                    db     4Eh ; N
047AEDA3 71                    db     71h ; q
047AEDA4                ;
047AEDA4 81 C3 E4 44 6C 01    add     ebx, 16C44E4h
047AEDA5 81 EB E0 44 6C 01    sub     ebx, 16C44E0h
```

Analysing the VEH

```
1  _DWORD *__usercall FN_Get_Offset_Update_EIP@<eax>(
2      _DWORD *offset@<eax>,
3      unsigned __int8 *enc_offset@<edx>,
4      int offset_key@<ecx>)
5  {
6      *offset += offset_key ^ *enc_offset;
7      return offset;
8  }
```

Analysing the VEH

```
Name  
1 offset_read = 0xC  
2 key = 0xE1  
3 val_eip = get_screen_ea()  
4 enc_offset = idaapi.get_byte(val_eip + offset_read)  
5 clr_offset = enc_offset ^ key  
6 print(hex(val_eip + clr_offset))  
  
Line 1 of 1  
Line:1 Column:1  
Scripting language Python Tab size 4 Run Export Imp  
0x47aeda4
```

```
047AED76 C7 03 A7 2B DB AC mov dword ptr [ebx], 0ACDB2BA7h  
047AED7C 81 03 43 75 B3 09 add dword ptr [ebx], 9B37543h  
047AED82 81 33 66 7A 31 80 xor dword ptr [ebx], 80317A66h  
047AED88 81 2B 32 DA BE 36 sub dword ptr [ebx], 36BFDA32h  
047AED8E 0F 07 sysret  
047AED8E ;  
047AED90 48 db 48h ; H  
047AED91 7A db 7Ah ; Z  
047AED92 00 db 0  
047AED93 00 db 0  
047AED94 00 db 0  
047AED95 00 db 0  
047AED96 00 db 0  
047AED97 00 db 0  
047AED98 00 db 0  
047AED99 00 db 0  
047AED9A F7 db 0F7h  
047AED9B 9B db 9Bh  
047AED9C 19 db 19h  
047AED9D D7 db 0D7h  
047AED9E 7B db 7Bh ; {  
047AED9F 13 db 13h  
047AEDA0 1C db 1Ch  
047AEDA1 A8 db 0A8h  
047AEDA2 4E db 4Eh ; N  
047AEDA3 71 db 71h ; q  
047AEDA4 ;  
047AEDA4 81 C3 E4 44 6C 01 add ebx, 10000000h  
047AEDA5 81 EB E0 44 6C 01 sub ebx, 10000000h
```



Decrypting Malware Configuration



Simple
XOR



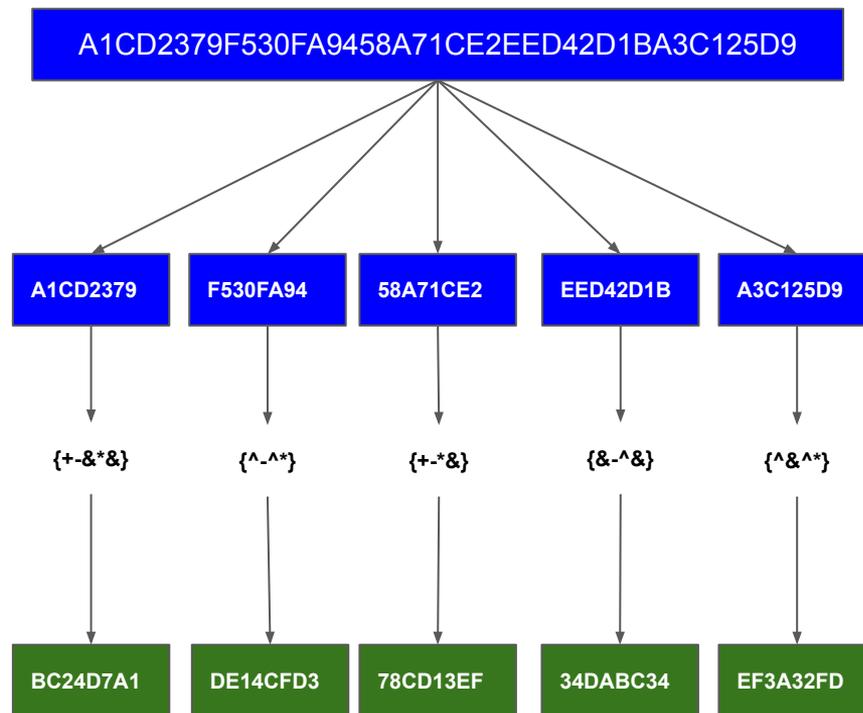
Predictable location of
encryption key



???

Ciphertext Splitting

1. Ciphertext splitted into multiple DWORD
2. Each DWORD is encoded with different arithmetic operations
3. Stored as local variables in functions



Ciphertext Splitting

- Function starts with loading of address of encrypted cipher text
- Values from local variables are written into the address

```
mov     ebx, [esp+4]
mov     dword ptr [ebx], 0ACDB2BA7h
add     dword ptr [ebx], 9B37543h
xor     dword ptr [ebx], 80317A66h
sub     dword ptr [ebx], 36BFDA32h
sysret
; -----
aHz db 'Hz',0,0,0,0,0,0,0,0,0F7h,9Bh,19h
```

Ciphertext Splitting

```
B = [0xACDB2BA7, 0x9B37543, 0x80317A66, 0x36BFDA32]
B1 = (B[0] + B[1]) & 0xFFFFFFFF
B2 = (B1 ^ B[2]) & 0xFFFFFFFF
result = (B2 - B[3]) & 0xFFFFFFFF
```

```
mov     ebx, [esp+4]
mov     dword ptr [ebx], 0ACDB2BA7h
add     dword ptr [ebx], 9B37543h
xor     dword ptr [ebx], 80317A66h
sub     dword ptr [ebx], 36BFDA32h
sysret
; -----
aHz db 'Hz',0,0,0,0,0,0,0,0,0F7h,9Bh,19h
```

First DWORD is the length of the ciphertext!

Locating the Encrypted Configs

- Function starts with “0x8B ?? 24 04”
- Yara it ?
- FP prone ?

```
8B 5C 24 04      mov     ebx, [esp+4]
C7 03 E4 9F 2F CE  mov     dword ptr [ebx],
81 2B A3 95 E0 08  sub     dword ptr [ebx],
81 33 3B 28 3A 4A  xor     dword ptr [ebx],
66 85 D8         test    ax, bx
```

```
8B 4C 24 04      mov     ecx, [esp+arg_0]
66 39 DA        cmp     dx, bx
66 85 C2        test    dx, ax
C7 01 C2 90 59 78  mov     dword ptr [ecx],
```

```
8B 5C 24 04      mov     ebx, [esp+arg_0]
84 EF          test    bh, ch
39 CB          cmp     ebx, ecx
C7 03 E3 6B D3 8A  mov     dword ptr [ebx],
```

Locating the Encrypted Configs

Demo 2 (DIY)

- Writing a yara rule to locate functions containing the encrypted configuration
- Using “Findcrypt-yara” IDA Pro plug-in <https://github.com/polymorf/findcrypt-yara>



Decrypting Malware Configuration



Simple
XOR



Predictable location of
encryption key



???

Locating the Decryption key

Demo 3

- Locate the decryption routine
- Locate the decryption key
- Locate the decryption key length



The Solution!



Unicorn
The Ultimate CPU emulator



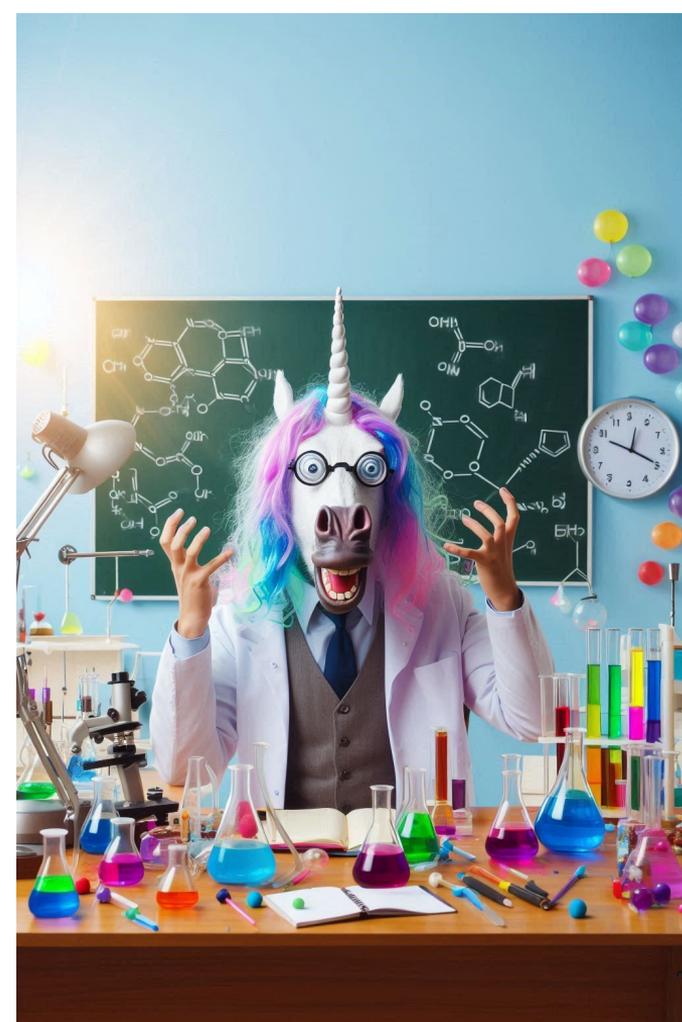
 **yara**

The pattern matching swiss knife for malware researchers (and everyone else)

The Solution!

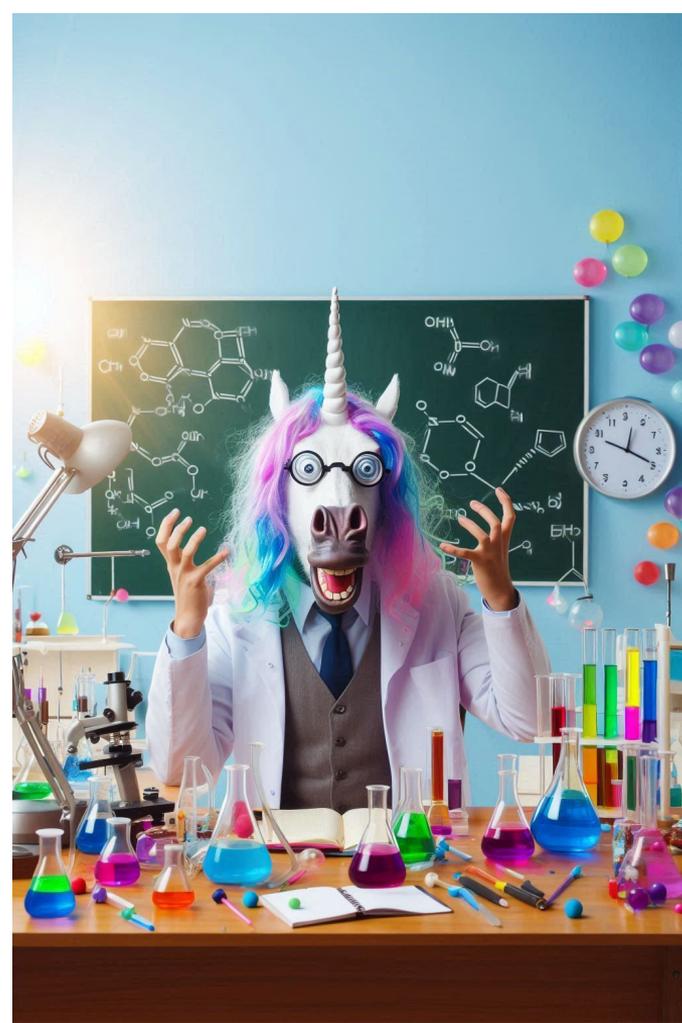
Demo 4

Putting it all together!



The Solution!

1. Using memory dump
2. Locate function containing splitted cipher text using yara
3. Using Unicorn CPU emulator framework
4. Emulate the function containing the DWORD
5. Handle the 5 types of exceptions



Analysing the VEH

```
1 int __usercall FN_Get_Offset Decode_key@<ecx>(_EXCEPTION_POINTERS *a1@<eax>)  
2 {  
3     PCONTEXT ContextRecord; // eax  
4     int result; // ecx  
5     int count; // edx  
6  
7     ContextRecord = a1->ContextRecord;  
8     result = 0x18;  
9     count = 0;  
10    while ( 1 )  
11    {  
12        count += 4;  
13        if ( *(DWORD *)((char *)&ContextRecord->ContextFlags + count) )// check all HW BP registers are zero  
14            break;  
15        if ( count == 0x18 )  
16            return 0xE1; // return offset decode key  
17    }  
18    return result;  
19 }
```

Analysing the VEH

```
call    FN_Get_Offset_Decode_key
test    cl, cl
cmp     edx, ebx
mov     edx, 63C7099Ch
xor     edx, 6649E730h
test    ch, bh
add     edx, 1691B385h
cmp     ah, ch
add     edx, 0E3DF5E87h    ; 0xB8
cmp     dx, 4070h
add     eax, edx          ; _Context.EIP = _CONTEXT+0xB8
test    dl, cl
mov     edx, [eax]
add     edx, 0Ch          ; Get enc offset*
cmp     dl, bl
call    FN_Get_Offset_Update_EIP
```

The Solution! (unicorn_hello_world.py)

```
PS C:\Users\User1\Desktop> python -V
Python 3.11.9
PS C:\Users\User1\Desktop> pip list
Package          Version
-----
capstone         5.0.3
pip              24.0
setuptools       65.5.0
unicorn          2.1.1
yara-python      4.5.1
```

The Solution! (unicorn_hello_world.py)

```
8B 54 24 04          mov     edx, [esp+4]
C7 02 D9 1C 40 91    mov     dword ptr [edx], 91401CD9h
81 32 9F 0D B0 5A    xor     dword ptr [edx], 5AB00D9Fh
0F C7 3B             vmptrst qword ptr [ebx]      ; invalid instruction
; -----
CF 00 00 00 00 00 00 00 00 F2...  db  0CFh, 8 dup(0), 0F2h, 3Eh, 0B2h, 71h, 17h, 0ECh,
; -----
81 32 54 80 FB 26    xor     dword ptr [edx], 26FB8054h
51                  push   ecx
B9 6C 4B BD FE      mov     ecx, 0FEBD4B6Ch
81 F1 D0 E9 76 97    xor     ecx, 9776E9D0h
81 F1 4A CC B4 55    xor     ecx, 55B4CC4Ah
81 C1 0A 92 80 C3    add     ecx, 0C380920Ah      ; ECX=0x100
56                  push   esi
9C                  pushf
89 E6              mov     esi, esp
01 0E              add     [esi], ecx
9D                  popf      ; enable Trap Flag!
39 CA              cmp     edx, ecx
; -----
75 16 96 E6 5D 07 09 1E 46 C0...aU  db  'u',16h,96h,0E6h,5Dh,7,9,1Eh,'F',0C0h,0E7h,9Bh,0F
EF 9D B1 9E C5 1A 3B 52 81 07...    db  '@',9Eh,0C5h,1Ah,';R',81h,7,0B8h,80h,0FCh,0B1h,'^
; -----
81 02 00 6F F4 12    add     dword ptr [edx], 12F46F00h
```

The Solution! (emulate_config_dump.py)

```
python3.11 emulate_config_dmp.py
```

```
INFO:__main__:C2 url = https://softiq.ro/event/update/mCNQZhdQboPBW61.bin enc config addr = 0x7A54BF
```



Agenda for part 2

- We will deep dive into .NET and how configuration extraction can be achieved
- Most of the samples are not very difficult to extract config, but we got to learn .NET runtime to achieve it.

- We will be going through:
 - Identify .NET
 - .NET structure
 - Tools
 - Sources for research
 - Unpacking .NET payloads
 - Locate configs
 - Understanding CLR tokens
 - Extract configs

Introduction to .NET

- .NET is a free and open-source application platform supported by Microsoft.
 - The initial .NET release was 2016. It's also getting attentions among threat actors.
- C# is the main programming language for .NET
- Terminology:
 - CLR (Common Language **R**untime)
 - CIL: Common **I**ntermediate **L**anguage is the bytecode language that the just-in-time (JIT) compiler of the .NET Framework interprets.
- There are awful lot of malwares written in .NET (especially infostealer)

How to identify .NET?

Cmd command: `$ file sample.exe`

```
sample.exe: PE32+ executable (GUI) x86-64 Mono/.Net assembly, for MS Windows
```

YARA has *is_dotnet* in dotnet module. Let's take a look at how it's implemented

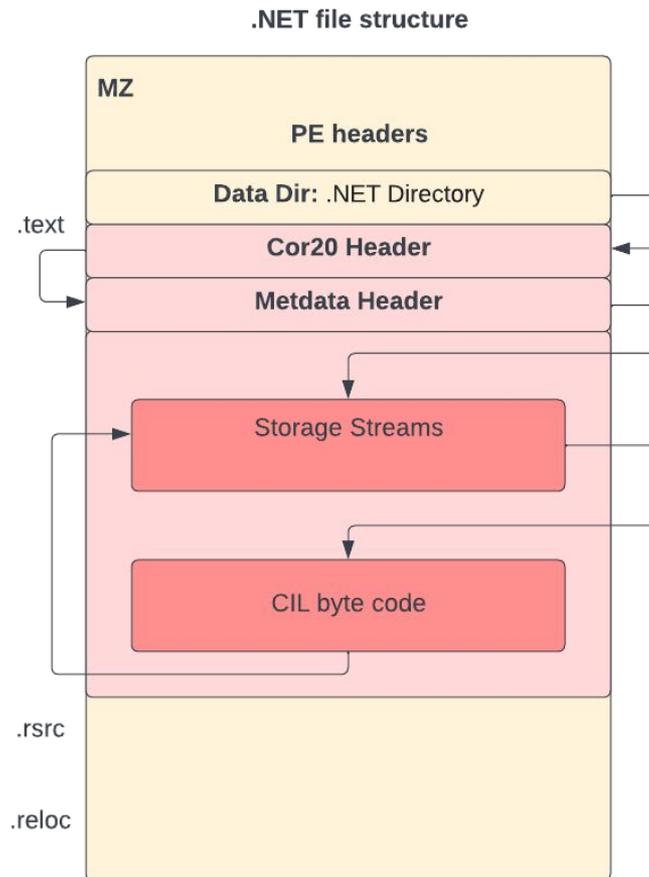
```
3229     static bool dotnet_is_dotnet(PE* pe)
3230     {
3231         PIMAGE_DATA_DIRECTORY directory = pe_get_directory_entry(
3232             pe, IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR);
3233
3234         if (!directory)
3235             return false;
3236
3237         int64_t offset = pe_rva_to_offset(pe, yr_le32toh(directory->VirtualAddress));
3238
3239         if (offset < 0 || !struct_fits_in_pe(pe, pe->data + offset, CLI_HEADER))
3240             return false;
3241
3242         CLI_HEADER* cli_header = (CLI_HEADER*) (pe->data + offset);
```

.NET header structure

- Reuse PE structure
- A special .NET data directory

IMAGE_DIRECTORY_ENTRY_COM_DESC

RIPTOR can be found for referencing the Cor20 header



.NET malware analysis tool I personally like

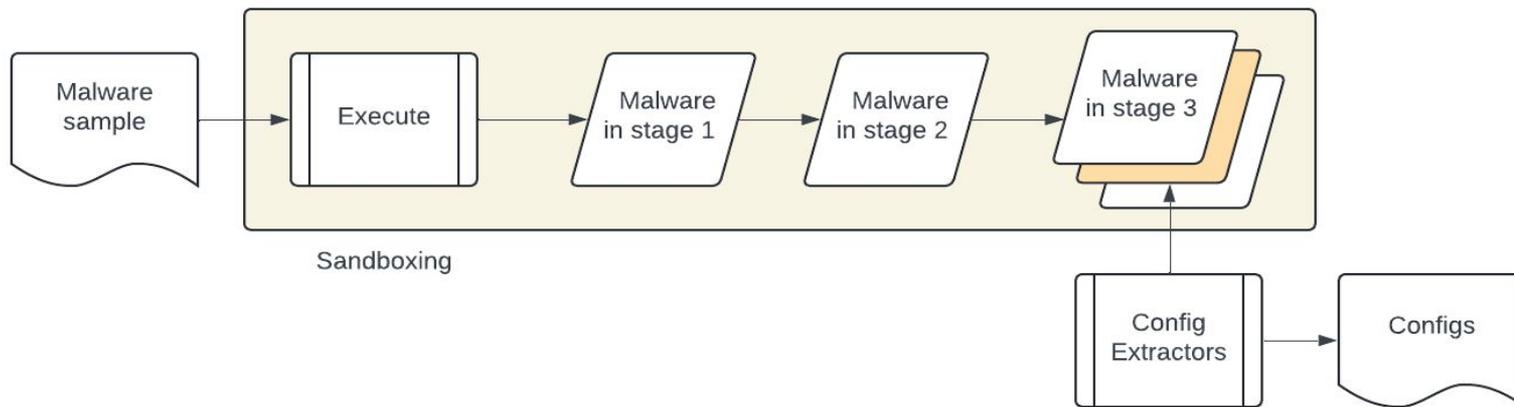
- dnSpy: a debugger and .NET assembly editor
 - The decompilation function is provided by ILSpy
 - The repo was archived since 2020; but it's still arguably the most popular tool for analysing .NET malware
- Megadumper: a handy .NET payload memory dump tool.
 - Very useful for unpacking
- IDAPro: not necessary, it's just my prefer tool to show CIL and decompiled code side-by-side.
- x32/x64 dbg: sometimes, when sample jumps from managed into unmanaged and vice versa. Most of the time I use dnSpy to debug .NET malwares.

Where to find .NET samples for research

- VirusTotal
- Malware bazaar Abuse.ch <https://bazaar.abuse.ch/browse/>

Fact is most of the samples are packed

- This means the Redline sample you downloaded from VT or malware bazaar doesn't look like Redline.
- However, configuration extractor requires Redline payload to work on.
- To defeat packing, in a production environment, the extractor is run on top of memory analysis framework:



Today, let's exercise unpack .NET samples

Terminology:

- packer/crypter/protector: the tool hides the real payload

Pre-requirement:

- A windows Virtual Machine with dnSpy installed

Sample:

- 458e5bd8e3508c15449bfd4c9931a59cd2a6a95ed9e6bb5b0090aa6641a29c77
 - It's a fresh sample on malware bazaar which is labeled as Agent-Tesla

Let's exercise how to unpack .NET samples (cont)

Step 1. Throw the sample into dnSpy and find the entrypoint

```
1 // C:\Users\McGreenn\Desktop\458e5bd8e3508c15449bfd4c9931a59cd2
2 // UKHN, Version=2.0.0.2, Culture=neutral, PublicKeyToken=null
3
4 // Entry point: Students.Program.Main
5 // Timestamp: 66F070D0 (9/23/2024 3:32:32 AM)
6
7 using System;
8 using System.Diagnostics;
9 using System.Reflection;
10 using System.Runtime.CompilerServices;
```

Step 2. Search for dynamic IL loading APIs and place a breakpoint at the instruction

- `typeof(Assembly).InvokeMember`, `assembly.GetType`, `.CurrentDomain.Load()`, etc

```
303 this.addStudentGroupBox.Text = "Add student";
304 Type type = AppDomain.CurrentDomain.Load(array2).GetExportedTypes()[0];
305 string @namespace = typeof(Form1).Namespace;
306 object[] args = new string[]
307
```

Let's exercise how to unpack .NET samples (cont)

Step 3. Run the sample and hope we are lucky enough to intercept the payload.

```
303     this.addStudentGroupBox.Text = "Add student";
304     Type type = AppDomain.CurrentDomain.Load(array2).GetExportedTypes()[0];
305     string @namespace = typeof(Form1).Namespace;
306     object[] args = new string[]
307     {
308         this.WA,
309         this.WZ,
310         @namespace
311     };
312     Activator.CreateInstance(type, args);
313     this.newStudentAverageGrade.Location = new Point(19, 234);
314     this.newStudentAverageGrade.Margin = new Padding(8, 7, 8, 7);
315     this.newStudentAverageGrade.Name = "newStudentAverageGrade";
316     this.newStudentAverageGrade.Size = new Size(260, 38);
```

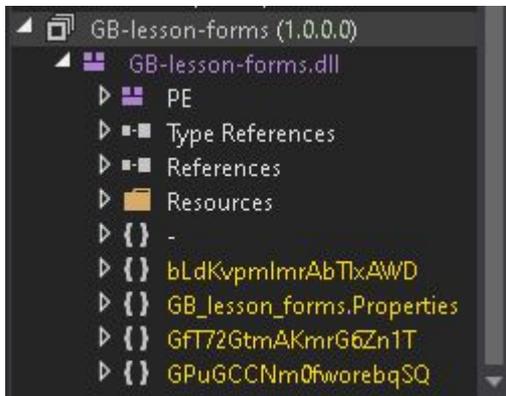
100 %

Locals

Name	Value
array2	byte[0x0000C400]
[0]	0x4D
[1]	0x5A
[2]	0x90
[3]	0x00
[4]	0x03
[5]	0x00
[6]	0x00

Let's exercise how to unpack .NET samples (cont)

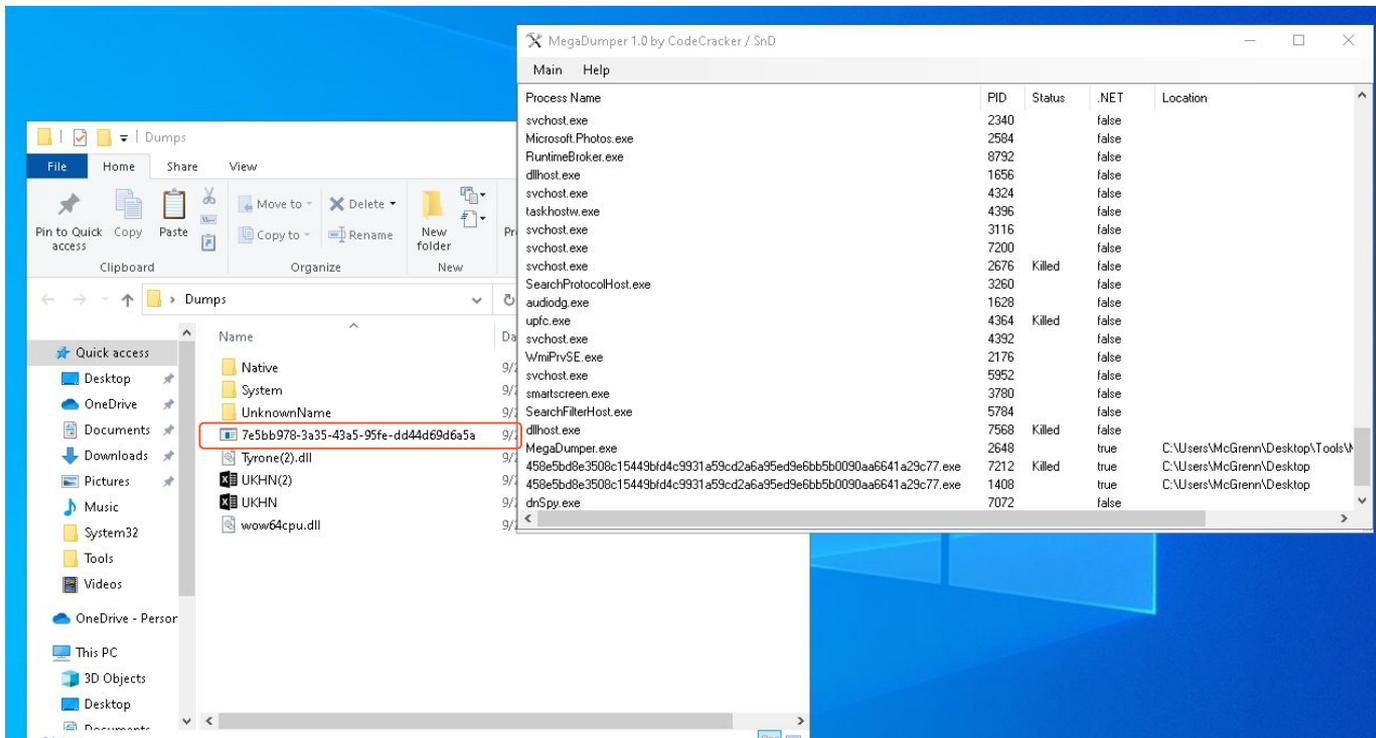
Step 4. Right click on the `array2` and save it to the disk. Reload the payload back to dnSpy.



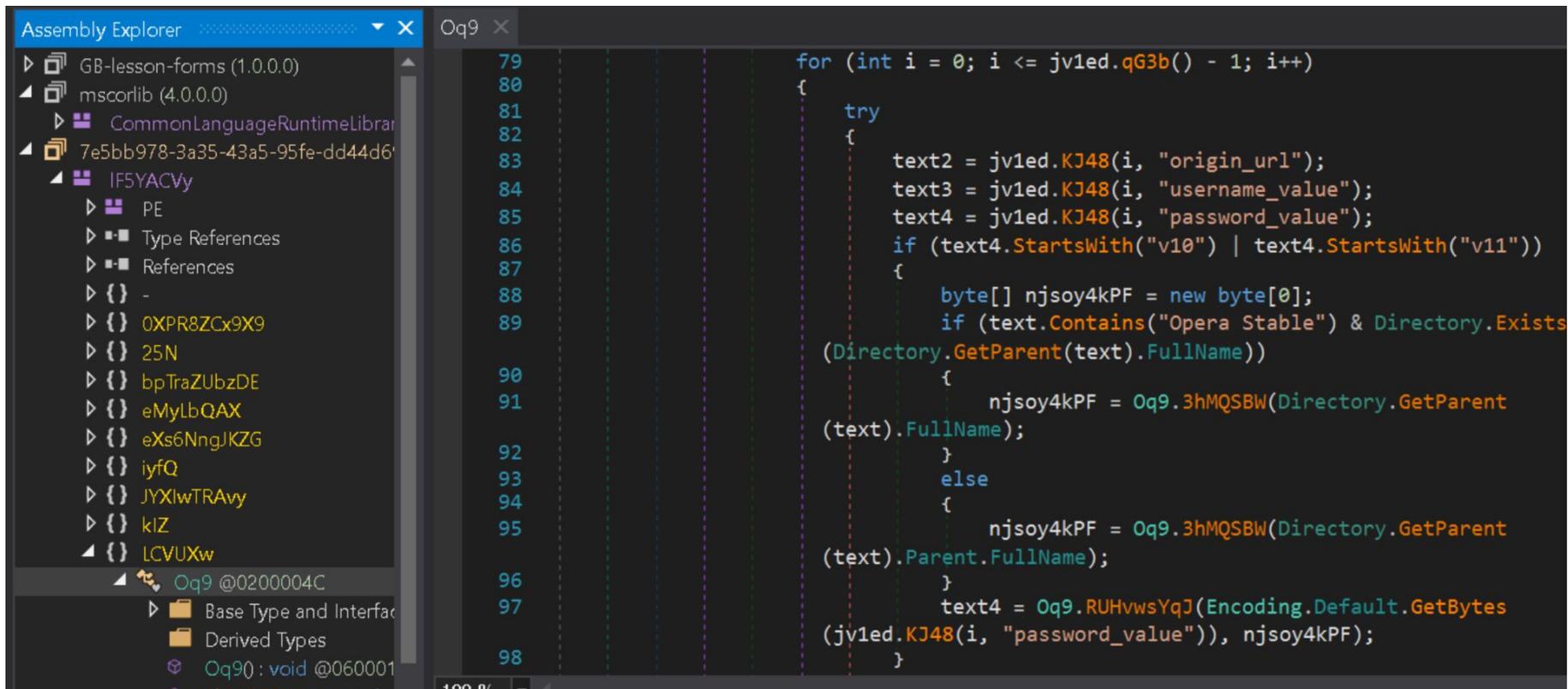
- It is not Agent-Tesla that we long for. We demonstrated that malware nowadays are multistage and **manually unpack them seems not easy to scale up**. So what's now?

Let's exercise how to unpack .NET samples (cont)

- Use my holy grail .NET unpacking tool: MegaDumper!



Let's exercise how to unpack .NET samples (cont)



```
79 for (int i = 0; i <= jv1ed.qG3b() - 1; i++)
80 {
81     try
82     {
83         text2 = jv1ed.KJ48(i, "origin_url");
84         text3 = jv1ed.KJ48(i, "username_value");
85         text4 = jv1ed.KJ48(i, "password_value");
86         if (text4.StartsWith("v10") | text4.StartsWith("v11"))
87         {
88             byte[] njsoy4kPF = new byte[0];
89             if (text.Contains("Opera Stable") & Directory.Exists
(Directory.GetParent(text).FullName))
90             {
91                 njsoy4kPF = Oq9.3hMQSBW(Directory.GetParent
(text).FullName);
92             }
93             else
94             {
95                 njsoy4kPF = Oq9.3hMQSBW(Directory.GetParent
(text).Parent.FullName);
96             }
97             text4 = Oq9.RUHvwsYqJ(Encoding.Default.GetBytes
(jv1ed.KJ48(i, "password_value")), njsoy4kPF);
98         }
99     }
100 }
```

Preparing to extract a configuration

Sample: 101b9564ba11aa44372b37b1143eac0d5dd1e3f38c6a35517de843b9f23b3704

Family: RedLine v2

Unpacks to 47d6bf807e275d25a63015ef106fb2548b5394342ec8fd7f809e1699810330

The sample gives away it's a RedLine:

```
1 using System;
2
3 // Token: 0x02000081 RID: 129
4 public static class Program
5 {
6     // Token: 0x060002AC RID: 684 RVA: 0x00007E92 File Offset: 0x00006092
7     private static void Main()
8     {
9         Program.ReadLine();
10    }
11
12    // Token: 0x060002AD RID: 685 RVA: 0x00018AE8 File Offset: 0x00016CE8
13    public static void ReadLine()
14    {
15        try
16        {
17            if (SystemInfoHelper.Close())
18            {
19                Delegate311.smeth0_0(0, Delegate311.delegate311_0);
20            }
21            GClass1 gclass = new GClass1();
22            bool flag = false;
23            while (!flag)
24            {
25                foreach (string address in Delegate42.smeth0_0(Strings.Get(0), new char[]
26                {
```

Where is the configuration:

Each malware family has their own design; but a general approach to find the config is:

- Configuration entries are always together. Because config controls the program, config is prepared altogether when sample is produced.
 - In .NET sample, configs can likely to be all in one class
- If the configuration is encrypted, malware is likely calling a same decryption routine over-and-over.
- Configuration often is associated with network connection part.
 - A trick is to locate the API that is related network and trace where the argument come from.

Where is the configuration in the sampe:

Sample: 101b9564ba11aa44372b37b1143eac0d5dd1e3f38c6a35517de843b9f23b3704

```
13 public static void Readline()
14 {
15     try
16     {
17         if (SystemInfoHelper.Close())
18         {
19             Delegate311.smethod_0(0, Delegate311.delegate311_0);
20         }
21         GClass1 gclass = new GClass1();
22         bool flag = false;
23         while (!flag)
24         {
25             foreach (string address in Delegate42.smethod_0(Strings.Get(0), new char[]
26             {
27                 '|',
28             }, Delegate42.delegate42_0))
29             {
30                 if (gclass.net_method_3(address) && gclass.method_0())
31                 {
32                     flag = true;
33                     break;
34                 }
35             }
36         }

```

The argument

Network related proc

Where is the configuration in the sampe (cont):

- Config entries are together
- Config is encrypted
 - call the same decryption func

```
9 public static class Strings
10 {
11     // Token: 0x06000197 RID: 407 RVA: 0x0001505C File Offset: 0x0001325C
12     static Strings()
13     {
14         Class8.smethod_0();
15         Strings.Keys = new string[3];
16         Strings.Keys[0] = "Proscribe";
17         Strings.Keys[1] = StringDecrypt.Read("NR8mCwAYAREEKignKVMjWjYjGUciHylGOFMjADw
18         Strings.Keys[2] = StringDecrypt.Read("Bxw5PDQlBQ8xYB0XKSs40TU0IEY6JTUAJDQ/JiM
19         Strings.Array = new List<string>();
20         List<string> array = Strings.Array;
21         byte[] array2 = new byte[32];
22         Delegate136.smethod_0(array2, fieldof(<PrivateImplementationDetails>.F495C984
23         array.Add(Strings.Decrypt(array2));
24         List<string> array3 = Strings.Array;
25         byte[] array4 = new byte[16];
26         Delegate136.smethod_0(array4, fieldof(<PrivateImplementationDetails>.struct2_
27         array3.Add(Strings.Decrypt(array4));
28         List<string> array5 = Strings.Array;
29         byte[] array6 = new byte[16];
30         Delegate136.smethod_0(array6, fieldof(<PrivateImplementationDetails>.struct2_
31         array5.Add(Strings.Decrypt(array6));
32         List<string> array7 = Strings.Array;
33         byte[] array8 = new byte[16];
34         Delegate136.smethod_0(array8, fieldof(<PrivateImplementationDetails>.struct2_
35         array7.Add(Strings.Decrypt(array8));
36         List<string> array9 = Strings.Array;
37         byte[] array10 = new byte[16];
38         Delegate136.smethod_0(array10, fieldof(<PrivateImplementationDetails>.D814B26
39         array9.Add(Strings.Decrypt(array10));
```

How config works?

- Set BreakPoint at line 22 and step over until line 24

```
9 public static class Strings
10 {
11     // Token: 0x06000197 RID: 407 RVA: 0x0001505C File Offset: 0x0001325C
12     static Strings()
13     {
14         Class8.smethod_0();
15         Strings.Keys = new string[3];
16         Strings.Keys[0] = "Proscribe";
17         Strings.Keys[1] = StringDecrypt.Read("NR8mCwAYAREEKignKVMjWjYjGUc");
18         Strings.Keys[2] = StringDecrypt.Read("Bxw5PDQlBQ8xYB0XKSs40TU0IEV");
19         Strings.Array = new List<string>();
20         List<string> array = Strings.Array;
21         byte[] array2 = new byte[32];
22         Delegate136.smethod_0(array2, fieldof(<PrivateImplementationDetail>));
23         array.Add(Strings.Decrypt(array2));
24         List<string> array3 = Strings.Array;
```

100 %

Locals

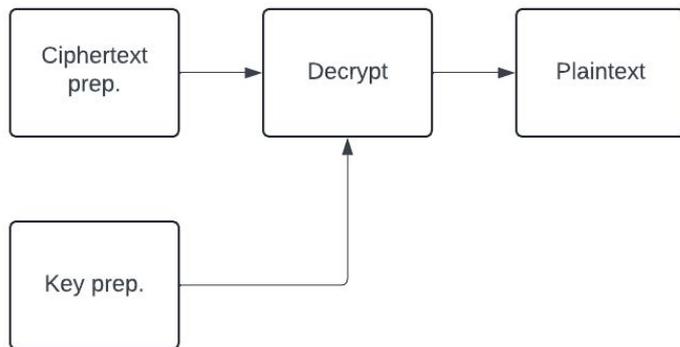
Name	Value
Strings.Decrypt returned	"82.147.85.205:24010"

Observations:

1. Key prep.
2. Ciphertext prep.
3. Decryption

Steps to extract config

- Config init.



1. **Get the ciphertext from binary**
2. Prepare the key
3. Reverse engineer the decryption

Extract config: get the ciphertext

Ciphertext is at <PrivateImplementationDetails>.F495C9...DA

```
Delegate136.smethod_0(array2, fieldof(<PrivateImplementationDetails>.F495C984B051BFF089D74440FC9FD44D1B9C4BDA).  
array.Add(Strings.Decrypt(array2));  
List<string> array3 = Strings.Array;
```

PrivateImplementationDetails is a class with all the structure inside:

```
6 [CompilerGenerated]  
7 internal sealed class <PrivateImplementationDetails>  
8 {  
9     // Token: 0x0400018C RID: 396 RVA: 0x00002050 File Offset: 0x00000250  
10     internal static readonly <PrivateImplementationDetails>.Struct2 struct2_0;  
11  
12     // Token: 0x0400018D RID: 397 RVA: 0x00002060 File Offset: 0x00000260  
13     internal static readonly <PrivateImplementationDetails>.Struct10 struct10_0;  
14 }
```

Pay attention to the comment in dnSpy.

Extract config: get the ciphertext, CIL view, p1

dnSpy decompiled view of accessing C&C ciphertext

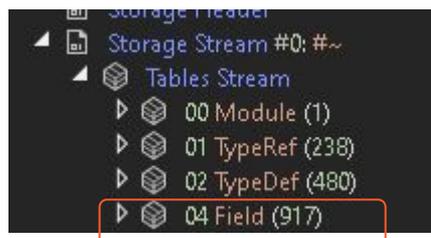
```
Delegate136.smethod_0(array2, fieldof(<PrivateImplementationDetails>.F495C984B051BFF089D74440FC9FD44D1B9C4BDA).  
array.Add(Strings.Decrypt(array2));  
List<string> array3 = Strings.Array;
```

In C, every object is reference by pointer. However, in .NET, ever object is accessed by token.

```
seg000:BF32 25      dup  
seg000:BF33 D0 1F 02 00 04  ldtoken  valuetype Struct4 <PrivateImplementationDetails>::F495C984B051BFF089D74440FC9FD44D1B9C4BDA  
seg000:BF38 7E D2 02 00 04  ldsfld   class Delegate136 Delegate136::delegate136_0  
seg000:BF3D 28 E0 04 00 06  call    void Delegate136::smethod_0(class [mscorlib]System.Array array_0, valuetype [mscorlib]Syste  
seg000:BF42 28 99 01 00 06  call    string Strings::Decrypt(unsigned int8[] chiperText)
```

Token(little endian) = 04 00 02 1F

- 04 is referencing Field Table Stream in #~
- The token is indexed 0x021F in Field Table



Extract config: get the ciphertext, CIL view, p2

In order to get the token, we want to anchor the code where C&C ciphertext token is access.

- Here is an example of YARA rule which we can use to anchor the offset of the instr
 - Or you can use regex
- Access the offset of YARA rule matches + 25 for the opcode of D0 (ldtoken), then the followed 4 bytes are ciphertext token

```
rule sig_redline_v2_c2
{
  strings:
    $ = {
      A2
      73 [3] 0A
      80 [3] 04
      7E [3] 04
      1F 20
      8D [3] 01
      25
      D0 [3] 04
      7E [3] 04
    }
  condition:
    all of them
}
```

seg000:BF32	25	dup	
seg000:BF33	D0 1F 02 00 04	ldtoken	valuetype Struct4 <PrivateImplementationDetails>::F495C984B051BFF089D74440FC9FD44D1B9C4BDA
seg000:BF38	7E D2 02 00 04	ldsflld	class Delegate136 Delegate136::delegate136_0
seg000:BF3D	28 E0 04 00 06	call	void Delegate136::smethod_0(class [mscorlib]System.Array array_0, valuetype [mscorlib]System
seg000:BF42	28 99 01 00 06	call	string Strings::Decrypt(unsigned int8[] chiperText)

Extract config: get the ciphertext, CIL view, p3

To get the data from the token, I found two different library are achievable

- <https://github.com/pan-unit42/dotnetfile>
- <https://github.com/malwarefrank/dnfile>

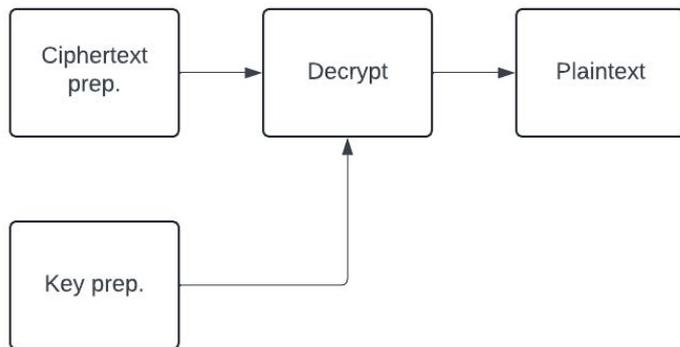
I'm using dnfile as an example for today. But they worked quite the same way.

By calling `get_field_data_from_token`, we get the actual data of C&C ciphertext

```
58 def get_field_data_from_token(token: int, dn: dnfile, size: int) -> Optional[bytes]:
59     token_mask = 0xFFFFFFFF
60     rva: Optional[int] = get_field_rva_by_index(index=token & token_mask, dn=dn)
61     if rva is None:
62         return None
63     return dn.get_data(rva, size)
64
65
66 def get_field_rva_by_index(index: int, dn: dnfile.dnPE) -> Optional[int]:
67     if not hasattr(dn, 'net') or not hasattr(dn.net, 'mdtables'):
68         return None
69
70     mdtables = dn.net.mdtables
71     for row in mdtables.FieldRva.rows:
72         if row.Field.row_index == index:
73             return row.Rva
74
75     return None
```

Steps to extract config

- Config init.



1. Get the ciphertext from binary
- 2. Prepare the key**
- 3. Reverse engineer the decryption**

Extract config: prepare the key and implement decryption, part 1

There's no trick, just pure reverse engineering.

One tip is debugger is helpful. I always make some guess, and prove my assumptions in debugging.

```
class RedlineV2Crypto:

    def __init__(self, key: bytes, iv: bytes, keykey: bytes):
        self.key: Optional[bytes] = self.xor_decrypt(data=key, key=keykey)
        self.iv: Optional[bytes] = self.xor_decrypt(data=iv, key=keykey)

    @property
    def ready(self) -> bool:
        return None not in [self.key, self.iv]

    @staticmethod
    def xor_decrypt(data: bytes, key: bytes) -> Optional[bytes]:
        try:
            return b64decode(b64decode(bytes([a ^ b for a, b in zip(b64decode(data), cycle(key))])))
        except ValueError:
            return None

    def decrypt(self, ciphertext: bytes) -> Optional[bytes]:
        cipher = AES.new(self.key, AES.MODE_CBC, self.iv)
        try:
            plaintext: bytes = unpad(cipher.decrypt(bytes(reversed(bytearray(ciphertext)))), 16)
            return plaintext
        except ValueError:
            return None
```

Extract config: prepare the key and implement decryption, part 2

Now, the same problem. We need to get the data for key prep.

```
Strings.Keys = new string[3];
Strings.Keys[0] = "Proscribe";
Strings.Keys[1] = StringDecrypt.Read("NR8mCwAYAREEKignKVMjWjYjGUciHy1GOFMjADwnKhsjBDA/HEE9PwY2BSyYBjg3JTY40iwxBicNCVNP", Strings.Keys[0]);
Strings.Keys[2] = StringDecrypt.Read("Bxw5PDQlBQ8xYB0XKSs40TU0IEY6JTUAJDQ/JiM7Q1o=", Strings.Keys[0]);
```

These data looks plaintext and static. However, we can expect every sample comes with different keys.

So, same approach. Looking at CIL first.

Token: 07 00 02 A7

07 is US (user) Table

```
seg000:BEE6 72 A7 02 00 70 ldstr aProscribe // "Proscribe"
seg000:BEEB A2 stelem.ref
seg000:BEEC 7E EE 00 00 04 ldsfld string[] Strings::Keys
seg000:BEF1 17 ldc.i4.1
seg000:BEF2 72 BB 02 00 70 ldstr aNr8mcwayareeki // "NR8mCwAYAREEKignKVMjWjYjGUciHy1GOFMjADw"..
seg000:BEF7 7E EE 00 00 04 ldsfld string[] Strings::Keys
seg000:BEFC 16 ldc.i4.0
seg000:BEFD 9A ldelem.ref
seg000:BEFE 28 39 01 00 06 call string StringDecrypt::Read(string b64, string stringKey)
seg000:BF03 A2 stelem.ref
seg000:BF04 7E EE 00 00 04 ldsfld string[] Strings::Keys
seg000:BF09 18 ldc.i4.2
seg000:BF0A 72 5E 03 00 70 ldstr aBxw5pdqlbq8xyb // "Bxw5PDQlBQ8xYB0XKSs40TU0IEY6JTUAJDQ/JiM"..
seg000:BF0F 7E EE 00 00 04 ldsfld string[] Strings::Keys
```

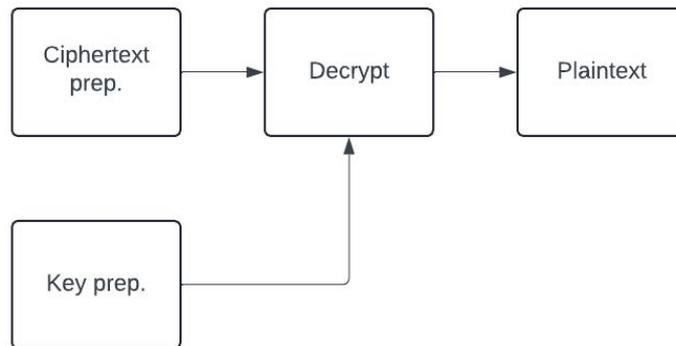
Extract config: prepare the key and implement decryption, part 3

To access the US stream, refer to this example:

```
34 def get_us_stream_by_token(token: int, dn: dnfile.dnPE) -> Optional[bytes]:
35     token_mask = 0xFFFFFFFF
36     return get_us_stream_by_offset(offset=token & token_mask, dn=dn)
37
38
39 def get_us_stream_by_offset(offset: int, dn: dnfile.dnPE) -> Optional[bytes]:
40     if not hasattr(dn, "net"):
41         return None
42
43     # get the (first) UserStrings stream
44     us: dnfile.stream.UserStringHeap = dn.net.metadata.streams.get(b"#US", None)
45     if us:
46         if not us.sizeof():
47             return None
48         ret = us.get_with_size(offset)
49         if ret is None:
50             return None
51         buf, _ = ret
52         try:
53             return dnfile.stream.UserString(buf[:-1]).value.encode()
54         except UnicodeDecodeError:
55             return None
```

Steps to extract config for .NET: rewind

1. Locate the config and analyse the configuration
2. Study CIL and prepare the anchor to get the token for ciphertext and keys
3. Get the data that token is pointing by the help of dnfile or dotnetfile
4. Reverse engineer the decryption routine
5. Put everything together



Thank you