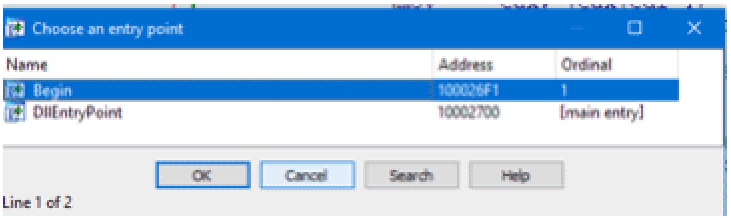
Dissecting DEloader malware with obfuscation

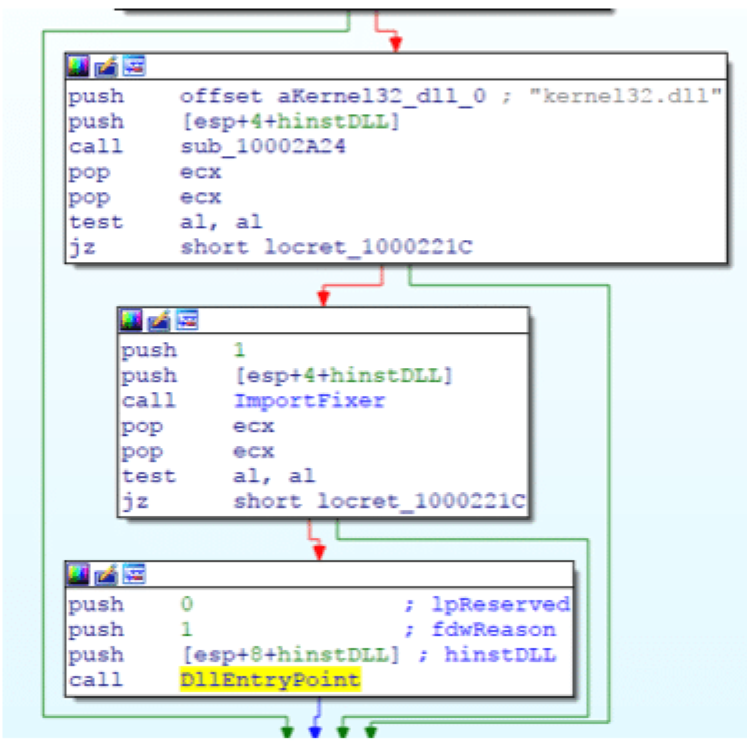
int0xcc.svbtle.com/dissecting-obfuscated-deloader-malware

September 6, 2018

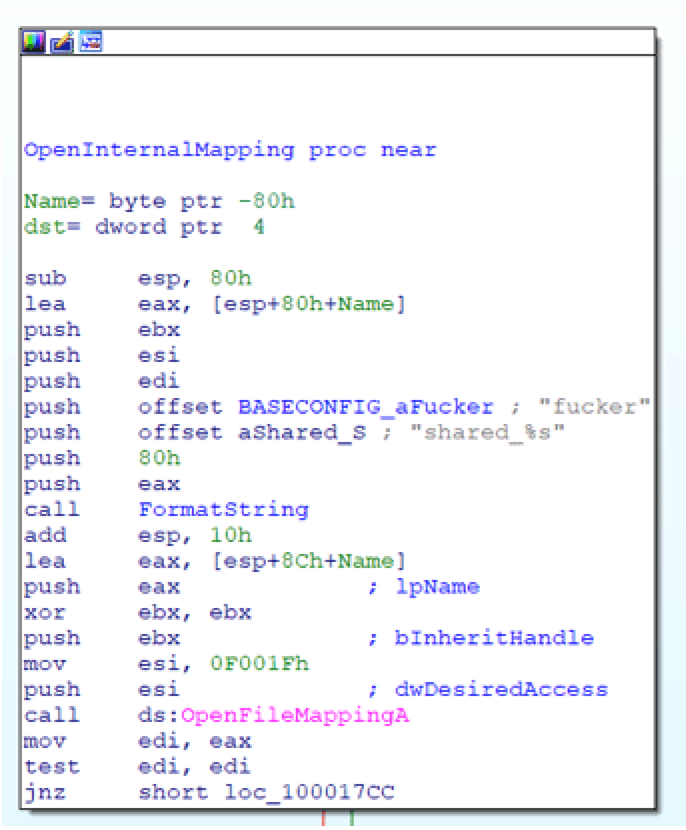
DEloader is a loader malware which is mostly used to load Zeus banking trojan . It is a stealth malware designed to keep the payload hidden and encrypted in the memory . A payload is dynamically retrieved from a remote https server So far there have been 3 versions of DEloader captured in the wild . Version **0x10E0700** , **0x1050500h** and **0x1120300h**. More recently in version **0x1120300h** they added code obfuscation

Main loader file is a DLL with export named as **'start'** or **'begin'** . These exports are called by packer . Essentially because this DLL is memory loaded image , imports and images are relocated via the code in these exports


Choose an entry point
Name
Address
Ordinal
Begin
100026F1
1
DllEntryPoint
10002700
[main entry]
OK
Cancel
Search
Help
Line 1 of 2

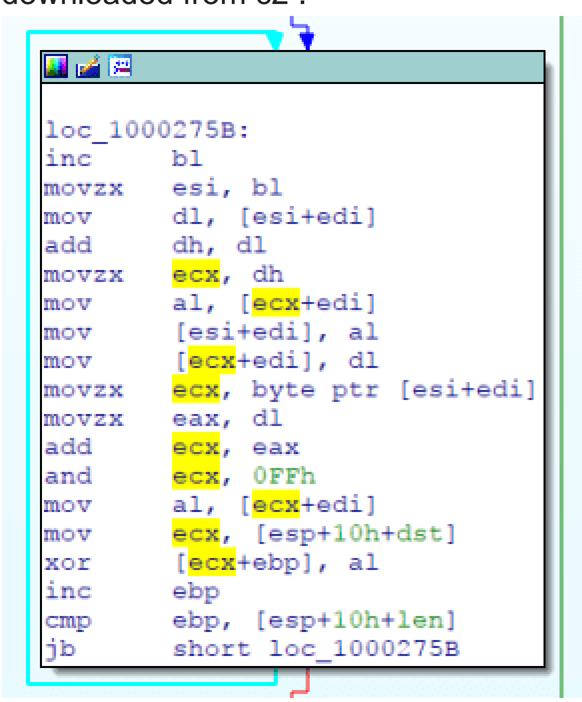

push offset aKernel32_dll_0 ; "kernel32.dll"
push [esp+4+hinstDLL]
call sub_10002A24
pop ecx
pop ecx
test al, al
jz short locret_1000221C
push 1
push [esp+4+hinstDLL]
call ImportFixer
pop ecx
pop ecx
test al, al
jz short locret_1000221C
push 0 ; lpReserved
push 1 ; fdwReason
push [esp+8+hinstDLL] ; hinstDLL
call DllEntryPoint

Earlier version included a share file map as a marker for infection . Shared file mapping would contain necessary information for the Deloader to run



```
OpenInternalMapping proc near

Name= byte ptr -80h
dst= dword ptr  4

sub     esp, 80h
lea     eax, [esp+80h+Name]
push    ebx
push    esi
push    edi
push    offset BASECONFIG_aFucker ; "fucker"
push    offset aShared_s ; "shared_%s"
push    80h
push    eax
call    FormatString
add     esp, 10h
lea     eax, [esp+8Ch+Name]
push    eax             ; lpName
xor     ebx, ebx
push    ebx             ; bInheritHandle
mov     esi, 0F001Fh
push    esi             ; dwDesiredAccess
call    ds:OpenFileMappingA
mov     edi, eax
test    edi, edi
jnz     short loc_100017CC
```

If the mapping is found, the data from the map is fed to decoding algorithm which is based on Rc4 and decodes using a fixed state buffer . This Algorithm is later used to decode buffer downloaded from c2 .



```
loc_1000275B:
inc     bl
movzx   esi, bl
mov     dl, [esi+edi]
add     dh, dl
movzx   ecx, dh
mov     al, [ecx+edi]
mov     [esi+edi], al
mov     [ecx+edi], dl
movzx   ecx, byte ptr [esi+edi]
movzx   eax, dl
add     ecx, eax
and     ecx, 0FFh
mov     al, [ecx+edi]
mov     ecx, [esp+10h+dst]
xor     [ecx+ebp], al
inc     ebp
cmp     ebp, [esp+10h+len]
jb      short loc_1000275B
```

Buffer can be either downloaded from c2 or the previously saved one is extracted from registry , which is later decoded using an embedded rc4 state buffer .

```
loc_10001E2A:               ; "laevwoa"
push    offset aLaevwoa
push    offset aSoftwareMicros ; "Software\\Microsoft\\%s"
lea     eax, [esp+708h+SubKey]
```

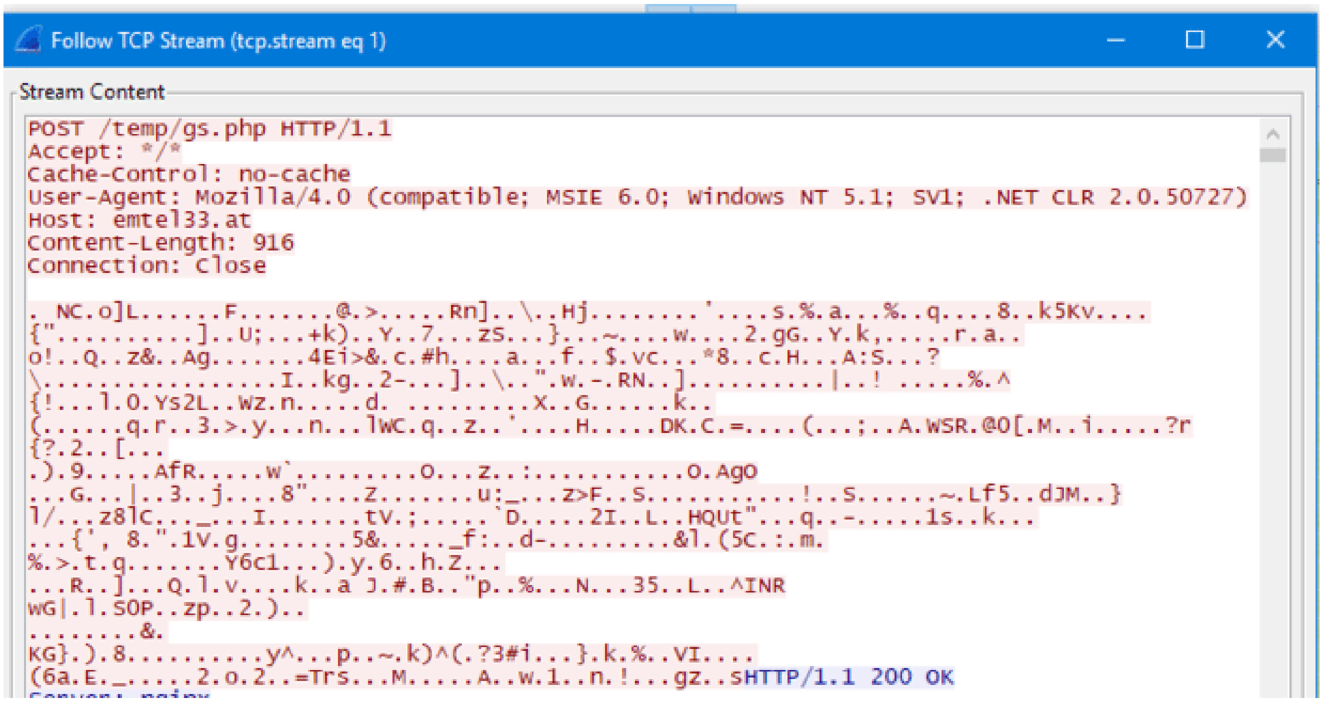
```
push    40h
pop     ecx
mov     esi, offset StateBuffer
lea     edi, [esp+70Ch+var_5B4]
rep movsd
lea     eax, [esp+70Ch+var_5B4]
push    eax
push    29Ch
push    ebp             ; Bytes
movsw
call    DecodeChunk
lea     eax, [esp+718h+mem]
push    eax
push    ebp
call    LoadSavedConfigData
add     esp, 14h
test    al, al
jz      short loc_10001F13
```

C2's are present in an embedded structure known baseconifg which consists configuration and c2's address necessary for the loader to operate . In both the versions static config in encoded state

It can have single or multiple c2's . Each of them is separated by a semi-colon ';' .
In earlier versions c2 url was present as an encoded resource on a remote https server . And was downloaded using a get HTTP/HTTPSs request

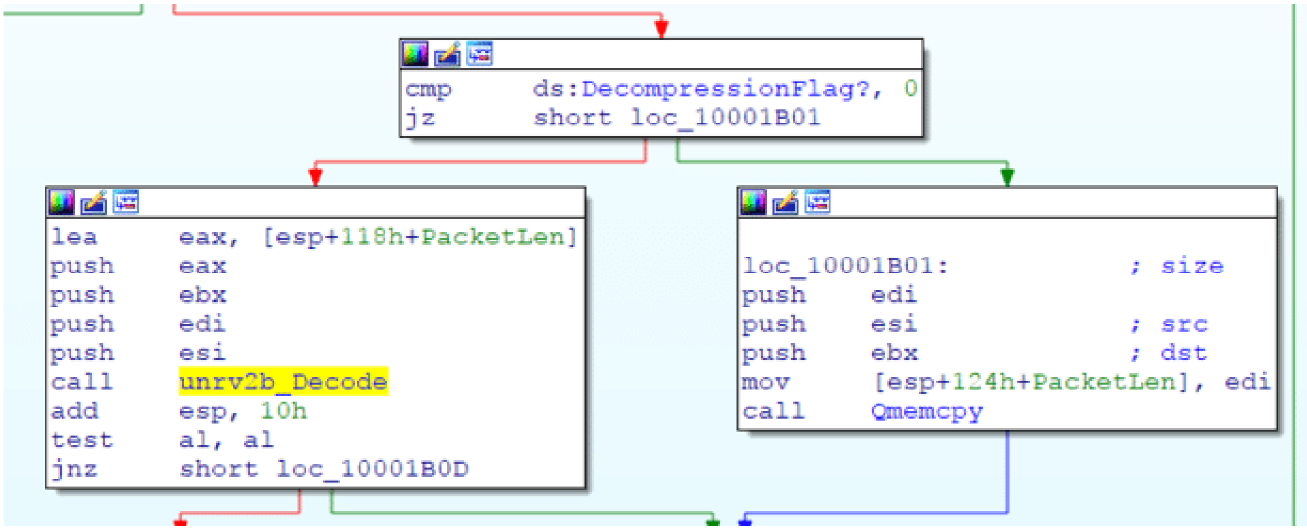
```
.rdata:100043CA                 db    0
.rdata:100043CB aHttpsAspecto_t db 'https://aspecto.top/dpr.bin; https://prisectos.top/dpr.bin',0
.rdata:100043CB                                         ; DATA XREF: c2Comm+5↑o
.rdata:10004406                 align 4
.rdata:10004408 a_xs            db '_Xæ|',0
.rdata:1000440D                 align 2
.rdata:1000440E                 db    2
.rdata:1000440F                 db    0
.rdata:10004410                 db  18h
```

However in the latest version, it includes a URL where encoded system internal data is posted and in return an encoded data buff is returned back .



This data is encoded with the same rc4state buffer extracted from static config embedded in the binary .Depending upon an internal flag it could be compressed as well . The compression algorithm used is unrv2b which happens to be the same one used in traditional Zeus malware .Also integrity of data is checked against a CRC32 hash DWORD present at the end of the data packet
raw response can represented as

```
struct RawResponse
{
        BYTE Data[len - 4];
        DWORD CRC32Data;

};



 struct
{
        __int64 DecompressionLength;
        BYTE CompressedData[]
};
```



After decompression data packet is arranged in a structure which consists of

```
struct InternalC2Parsed
{
        unsigned int PlaceHolder = 0x1000000;
        unsigned int Version; // 4
        void *PEBuffer_32bit;
        unsigned int PEBuffer_32bit_len;
        void *PEBuffer_64bit;
        unsigned int PEBuffer_64bit_len;
        void *C2StructDecompressed;
        int C2StructDecompressed_len;

};
```

Depending upon the type of system a particular type of payload(32bit or 64bit) payload in injected in process memory . If the system happens to be 64bit , a well known technique “heavens gate” is used to inject to 64bit process from a 32 bit running process

```
.text:100016A5                    movlpd  [ebp+var_34], xmm0
.text:100016AA                    mov     dword ptr [ebp+var_2C], eax
.text:100016AD                    mov     dword ptr [ebp+var_2C+4], edx
.text:100016B0                    mov     [ebp+var_4], esp
.text:100016B3                    and     esp, 0FFFFFFF0h
.text:100016B6                    push    33h
.text:100016B8                    call    $+5             ; JUMP 0x33SEG
.text:100016BD                    add     [esp+48h+var_48], 5
.text:100016C1                    retf
.text:100016C1 sub_10001601       endp ; sp-analysis failed
.text:100016C1
.text:100016C2 ; ---------------------------------------------------------------
.text:100016C2                    dec     eax             ; 64bit payload for injection
.text:100016C3                    mov     ecx, [ebp-0Ch]
```

Following python script demonstrates the ability to decode and decompress

```
#!/usr/bin/env python

import ucl
def PRGA(S):
    i = 0
    j = 0
    while True:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]  # swap


        K = S[(S[i] + S[j]) % 256]
        yield K


if __name__ == '__main__':
    plaintext = open("Bindata", "rb").read()
    import array

    keystream = [
    0xD7, 0x81, 0x83, 0xA6, 0x59, 0x4B, 0x88, 0x32, 0xFB, 0x8D, 0x7A, 0x64, 0x08,
0x9F, 0x6D, 0x01,
    0x2C, 0xD8, 0x50, 0xCE, 0xA3, 0x4A, 0xF9, 0x21, 0x40, 0x91, 0xE4, 0x28, 0x22,
0xAA, 0x41, 0x0D,
    0x68, 0x44, 0xA7, 0xB8, 0xA5, 0xFE, 0x3A, 0x2F, 0x7C, 0xDA, 0x37, 0x94, 0x46,
0x92, 0x86, 0x0A,
    0x25, 0xEA, 0x45, 0xB1, 0xAE, 0x7B, 0xE2, 0x3F, 0xBC, 0x7D, 0x84, 0x9A, 0xE5,
0x77, 0x0F, 0xA2,
    0xDD, 0x1A, 0x5F, 0xFA, 0x78, 0x67, 0x12, 0x02, 0x03, 0x3B, 0x65, 0x62, 0xF5,
0xBE, 0x8C, 0x27,
    0x9D, 0x69, 0xA8, 0x56, 0x5E, 0xE6, 0x61, 0xFF, 0x72, 0x5C, 0x19, 0xD6, 0xD4,
0x6A, 0x52, 0xD2,
    0xDC, 0x55, 0xDF, 0x70, 0x18, 0x0C, 0xEE, 0x87, 0x95, 0x07, 0xA1, 0x05, 0xA4,
0x5D, 0xE1, 0x06,
    0xB0, 0xC0, 0x29, 0x80, 0x53, 0xE7, 0xE3, 0x93, 0x16, 0xF2, 0x1B, 0x96, 0xDB,
0x90, 0xAC, 0xF6,
    0x7E, 0x6F, 0xF1, 0x6C, 0xB6, 0xF4, 0x63, 0xB3, 0x8A, 0xC3, 0xFC, 0x8F, 0x1F,
0x3D, 0x9C, 0x2B,
    0xB9, 0xCB, 0x35, 0x2D, 0xA0, 0xC6, 0x74, 0xFD, 0xBF, 0x23, 0xEB, 0xB5, 0x89,
0x82, 0x30, 0xBB,
    0x0B, 0x76, 0x17, 0x4F, 0x4E, 0x1E, 0xD9, 0x58, 0x13, 0x6B, 0x26, 0x9E, 0xD0,
0xE0, 0x48, 0xF0,
    0x6E, 0xB4, 0x0E, 0xC4, 0xEC, 0x00, 0xD1, 0xCF, 0xC8, 0x7F, 0x20, 0x38, 0x79,
0xCD, 0x49, 0xC7,
    0x47, 0xED, 0x31, 0xCA, 0xC1, 0x39, 0xC9, 0x98, 0x1D, 0x33, 0x5A, 0x3E, 0x51,
0x4C, 0x8B, 0x24,
    0xB2, 0xB7, 0x4D, 0xE8, 0x54, 0xEF, 0x9B, 0xC5, 0x09, 0xF7, 0x2A, 0x3C, 0xBD,
0x36, 0x71, 0x2E,
    0x15, 0xF3, 0xA9, 0x60, 0x10, 0xAF, 0xC2, 0x73, 0x97, 0x34, 0x66, 0x99, 0x8E,
0xDE, 0xAD, 0xAB,
    0xBA, 0xF8, 0x11, 0xD5, 0x75, 0x43, 0x57, 0x04, 0xCC, 0xE9, 0x42, 0x85, 0x14,
0x1C, 0x5B, 0xD3
]
```

```
arr = array.array("B", keystream)
keystream = PRGA(arr)
import sys
finBuf = array.array("B")

i = 0
for c in plaintext:

    finBuf.append(ord(c) ^ keystream.next())

    i = i + 1

open("FinalData.bin", "wb").write(finBuf.tostring())
```

and to finally decomrpess the data we can use CTYPES to call the following subroutine in python
https://github.com/wt/coreboot/blob/master/payloads/bayou/nrv2b.c

```
#ifndef ENDIAN
#define ENDIAN   0
#endif
#ifndef BITSIZE
#define BITSIZE 32
#endif

#define GETBIT_8(bb, src, ilen) \
    (((bb = bb & 0x7f ? bb*2 : ((unsigned)src[ilen++]*2+1)) >> 8) & 1)

#define GETBIT_LE16(bb, src, ilen) \
    (bb*=2,bb&0xffff ? (bb>>16)&1 : (ilen+=2,((bb=(src[ilen-2]+src[ilen-
1]*256)*2+1)>>16)&1))
#define GETBIT_LE32(bb, src, ilen) \
    (bc > 0 ? ((bb>>--bc)&1) : (bc=31,\
    bb=*(const uint32_t *)((src)+ilen),ilen+=4,(bb>>31)&1))

#if ENDIAN == 0 && BITSIZE == 8
#define GETBIT(bb, src, ilen) GETBIT_8(bb, src, ilen)
#endif
#if ENDIAN == 0 && BITSIZE == 16
#define GETBIT(bb, src, ilen) GETBIT_LE16(bb, src, ilen)
#endif
#if ENDIAN == 0 && BITSIZE == 32
#define GETBIT(bb, src, ilen) GETBIT_LE32(bb, src, ilen)
#endif

static unsigned long unrv2b(uint8_t * src, uint8_t * dst, unsigned long *ilen_p)
{
        unsigned long ilen = 0, olen = 0, last_m_off = 1;
        uint32_t bb = 0;
        unsigned bc = 0;
        const uint8_t *m_pos;

        // skip length
        src += 4;
        /* FIXME: check olen with the length stored in first 4 bytes */

        for (;;) {
                unsigned int m_off, m_len;
                while (GETBIT(bb, src, ilen)) {
                        dst[olen++] = src[ilen++];
                }

                m_off = 1;
                do {
                        m_off = m_off * 2 + GETBIT(bb, src, ilen);
                } while (!GETBIT(bb, src, ilen));
                if (m_off == 2) {
                        m_off = last_m_off;
                } else {
                        m_off = (m_off - 3) * 256 + src[ilen++];
                        if (m_off == 0xffffffffU)
                                break;
                        last_m_off = ++m_off;
```

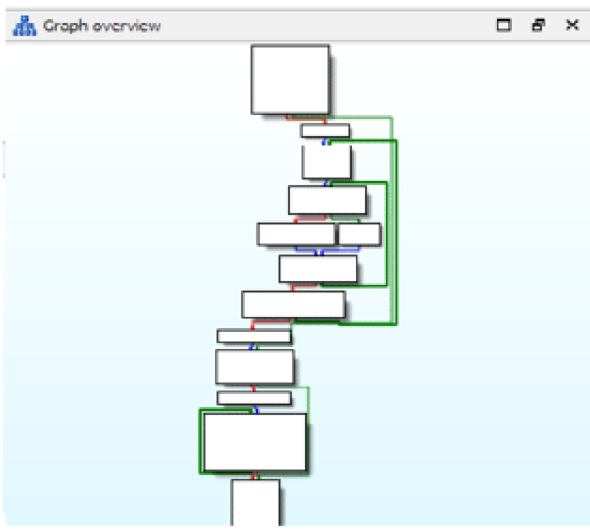
```
                }

                m_len = GETBIT(bb, src, ilen);
                m_len = m_len * 2 + GETBIT(bb, src, ilen);
                if (m_len == 0) {
                        m_len++;
                        do {
                                m_len = m_len * 2 + GETBIT(bb, src, ilen);
                        } while (!GETBIT(bb, src, ilen));
                        m_len += 2;
                }
                m_len += (m_off > 0xd00);

                m_pos = dst + olen - m_off;
                dst[olen++] = *m_pos++;
                do {
                        dst[olen++] = *m_pos++;
                } while (--m_len > 0);
        }

        *ilen_p = ilen;

        return olen;

}
```

Finally after decoding and decompression a vaid PE file is obtained . A file size of 1.05MB.

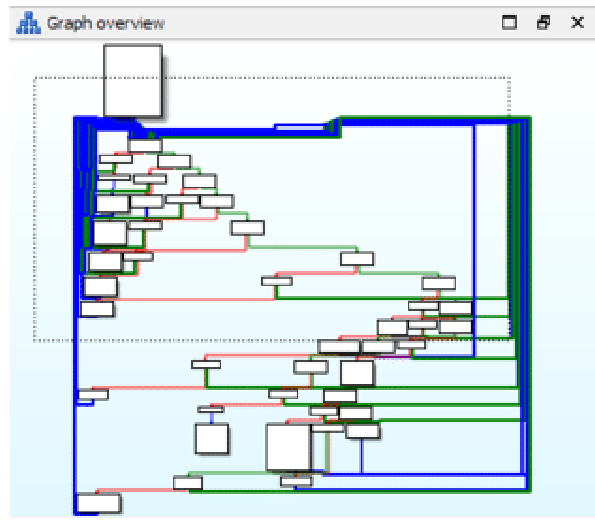
Source code level obfuscation .

In a more recent version 0x1120300h source code level obfuscation was added . This type of obfuscation is known as opaque predicates which makes the process of reverse engineering bit difficult . The basic Idea behind this technique is to include calculation based comparison instruction which end with a conditional jump , which are not the part of the original code , but are the part of code path .

In the images below a comparison is shown between a CRC32() function in version **0x1120300h** and an earlier version **0x1050500h**. Which demonstrates the multiple junk instructs and paths added with inclusion of opaque predicates

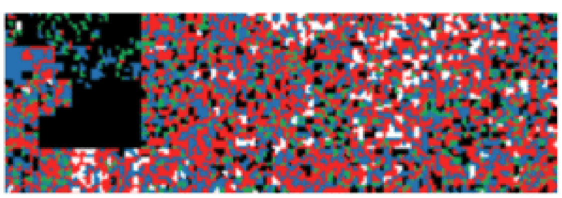


CRC32() in 0x1050500h Version

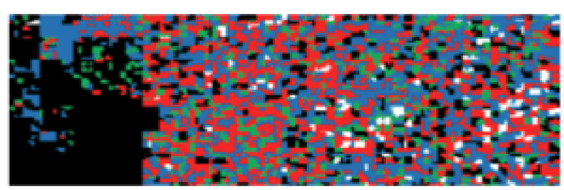


CRC32() in 0x1120300h version

This happens to be quite evident in the entropy comparison of the binary in whole .



Binary 1



Binary 2

Even the downloaded payload which happens to be a version of traditional Zeus banking malware is also obfuscated , which generally in its unpacked form is detected by most of then antivirus scans , but due to code level obfuscation is marked clean by most of the major anti virus engines

AhnLab-V3	Clean	Antiy-AVL	Clean
Avast	Clean	Avast Mobile Security	Clean
AVG	Clean	Avira	Clean
AVware	Clean	Baidu	Clean
Bkav	Clean	CAT-QuickHeal	Clean
ClamAV	Clean	CMC	Clean
Cylance	Clean	Cyren	Clean
eGambit	Clean	Endgame	Clean
ESET-NOD32	Clean	F-Prot	Clean
Fortinet	Clean	Ikarus	Clean
Jiangmin	Clean	K7AntiVirus	Clean
K7GW	Clean	Kaspersky	Clean
Kingsoft	Clean	Malwarebytes	Clean
McAfee	Clean	McAfee-GW-Edition	Clean
Microsoft	Clean	NANO-Antivirus	Clean
nProtect	Clean	Palo Alto Networks	Clean
Panda	Clean	Qihoo-360	Clean
Rising	Clean	SentinelOne	Clean

conclusion :

Deloader is still under heavy development . DeLoader has consistently evolved since past few years . With the addition of a hard obfuscation technique is it quite sure that the authors of deloader want to make this analysis hard and apparently makes it slip the anti virus filter . The use of encryption and compression make the data sent around the command and control server cryptic and hard to detect using a pattern . The payload which s mostly being delivered is a financial malware , designed to steal banking credentials , which makes it clear that authors are inclined towards monetization of injecting machines .

13

Kudos

13

Kudos