Related Insights

info.phishlabs.com/blog/dissecting-the-qadars-banking-trojan

February 23, 2017

By Stacy Shelley | February 22, 2017

Qadars is a sophisticated and dangerous trojan used for crimeware-related activities including banking fraud and credential theft. Qadars targets users through exploit kits and is installed using Powershell Scripts. We have observed Qadars targeting multiple well-known banks in UK and Canada and is capable of stealing infected users' two-factor authentication codes and banking credentials through the deployment of webinjects. While not as well known or widespread as other Trojans, the operators have shown commitment to development of Qadars' on-board evasion techniques and its advanced and adaptable privilege escalation module. This emphasis on persistence alongside the frequent shifts in both industry and geographic targeting indicate Qadars will remain a potent threat through 2017.

```
bank.barclays.co.uk/olb/auth/LoginStep1WithoutAssistCookie_display.action
business.hsbc.co.uk/1/2/!ut/p/c5
security.hsbc.co.uk/gsa/?idv_cmd=idv.Authentication
retail.santander.co.uk/LOGSUK_NS_ENS/ChannelDriver.ssobto
business.santander.co.uk/LGSBBI_NS_ENS/ChannelDriver.ssobto
corporate.santander.co.uk/LOGSCU_NS_ENS/ChannelDriver.bto
personal.metrobankonline.co.uk/MetroBankRetail/ajaxservletcontroller
corporate.metrobankonline.co.uk/modelbank/unprotected/LoginServlet
```

List of webinject targets from a Qadars configuration file

In this technical blog post, we will analyze a Qadars binary file and provide code and a Yara rule to aid in the analysis and detection of this banking Trojan. First, we will examine Qadars' methods of thwarting reverse engineering through the utilization of a dynamically resolved Import Address Table with obfuscated functions and strings. We then will detail the trojan's behaviour and dynamically-generated command and control centers with which it communicates. The C2s are not utilized solely for the collection of stolen credentials. We have also observed them delivering a module to Qadars samples operating in a low privilege environment that employs social engineering to trick the user into allowing higher level access.

Import Address Table (IAT) and String Obfuscation

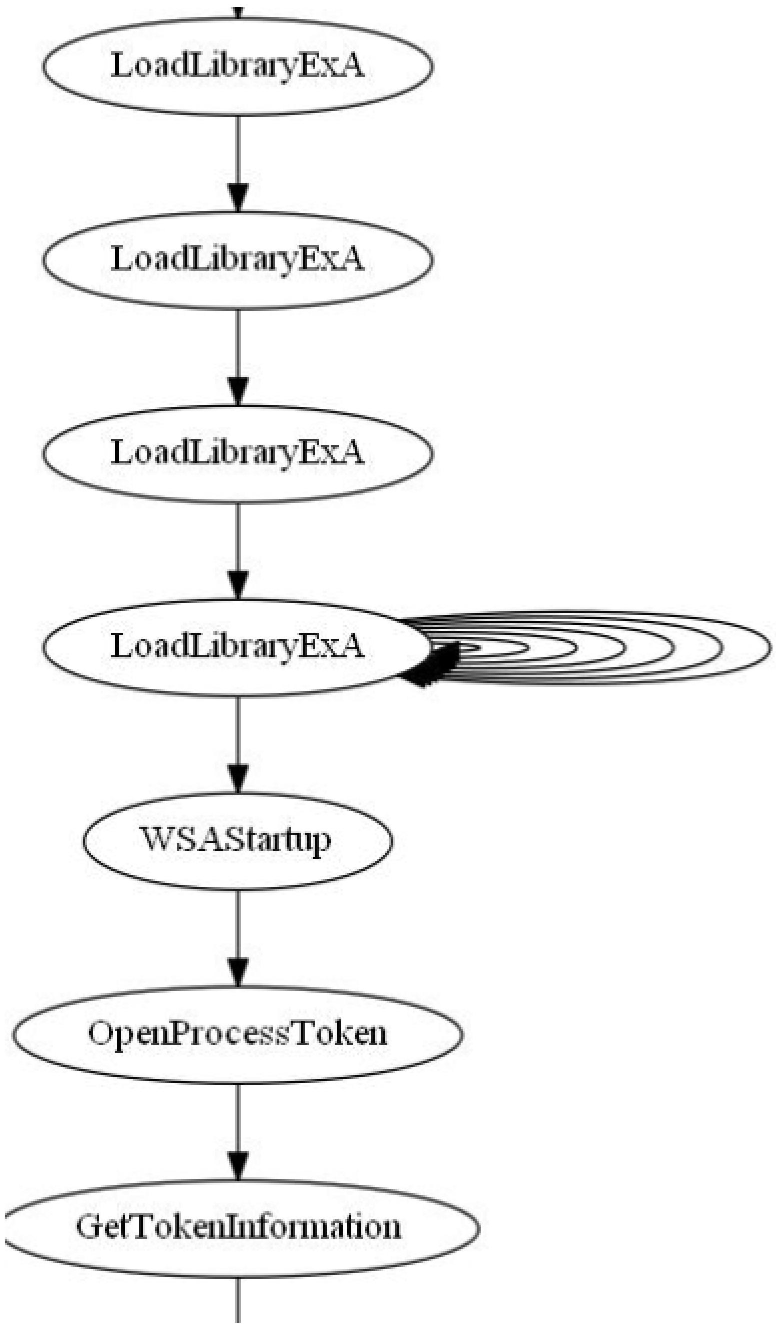
In its pure form, Qadars has built-in protection to make reverse engineering difficult, such as dynamic resolution of the Import Address Table (IAT) and obfuscation of the IAT functions and strings. At the beginning of execution, it calls a subroutine responsible for resolving and concealing IAT entries.

It locates API entries using a well-known hashing method. For example, in the code depicted below, 9B102E2Dh corresponds to LoadLibraryA:

```
mov     [ebp+var_4], eax
push    1
push    9B102E2Dh
mov     eax, [ebp+var_4]
push    eax
call    LocateExport
```

Resolving API Calls via Hashing Mechanism

Dynamic Link Libraries (DLLs) are loaded using LoadLibrary and API names are located by parsing the export address table as show in the trace file below:



Loading DLLs Using LoadLibrary Function

Furthermore, Qadars conceals an API function by XORing the address of an API call with a 4-byte XOR-Key. Wherever there is a call to a particular API function, the original value is reverted back to its XOR-encoded value.

```
0040AED8
0040AED8 loc_40AED8:
0040AED8 mov     eax, GlobalXorKey
0040AEDD xor     eax, CreateMutexA ; Retrieve Orginal Value
0040AEE3 lea     ecx, [ebp+var_5D0]
0040AEE9 push    ecx
0040AEEA push    ebx
0040AEEB push    1
0040AEED push    ebx
0040AEEE call    eax               ; Call
0040AEF0 mov     [ebp+var_54], eax
0040AEF3 cmp     eax, ebx
0040AEF5 jnz     short loc_40AF02
```

Decoding XOR-encoded API Call

In order to simplify the analysis, we can utilize one of two methods: create an IDA script to statically resolve the import addresses, or create an IDA script to rebuild the IAT. We will utilize the latter method.

Reconstructing Imports by Instruction Patching

In order to restore the imported function, we would need our instructions to specify CALL [APIPointer] instead of CALL . However, patching an indirect call would not be allowed because the size of an indirect call is only 2 bytes, while the size of a referenced call is 6 bytes. We could accommodate these additional 4 bytes by NOP'ing the previous XOR operation which is used to retrieve the original value. In this manner, we could keep the offsets at their specified and original locations. The following code comparison (also known as diff) illustrates this concept:

Original Instructions

```
0040AE72 mov     ecx, GlobalXorKey
0040AE78 xor     ecx, WSAStartup_Xored
0040AE7E xor     eax, eax
0040AE80 add     esp, 30h
0040AE83 mov     [ebp+var_1C], eax
0040AE86 mov     [ebp+var_18], eax
0040AE89 mov     [ebp+var_48], eax
0040AE8C mov     [ebp+var_44], eax
0040AE8F lea     eax, [ebp+var_760]
0040AE95 push    eax
0040AE96 push    202h
0040AE9B mov     [ebp+var_24], ebx
0040AE9E mov     [ebp+var_20], ebx
0040AEA1 mov     [ebp+var_50], ebx
0040AEA4 mov     [ebp+var_4C], ebx
0040AEA7 call    ecx                     ; WSAStartup()
```

Patched and Shifted

```
0040B772 mov     ecx, GlobalXORKEY
0040B778 xor     eax, eax
0040B77A add     esp, 30h
0040B77D mov     [ebp+var_1C], eax
0040B780 mov     [ebp+var_18], eax
0040B783 mov     [ebp+var_48], eax
0040B786 mov     [ebp+var_44], eax
0040B789 lea     eax, [ebp+WSAData]
0040B78F push    eax                   ; lpWSAData
0040B790 push    202h                  ; wVersionRequested
0040B795 mov     [ebp+PtrSizeOut], ebx
0040B798 mov     [ebp+var_20], ebx
0040B79B mov     [ebp+var_50], ebx
0040B79E mov     [ebp+var_4C], ebx
0040B7A1 call    WSAStartup
0040B7A7 nop
0040B7A8 nop
```

NOP Inserted

Original Instructions shifted up

Maintaining Memory Offsets by Inserting NOP Instructions

All resolved entries are stored in an array 748 bytes in size consisting of 187 total API calls.

```
.data:004173BC APIAddressStart dd ?
.data:004173BC
.data:004173C0 dword_4173C0    dd ?
.data:004173C0
.data:004173C4 dword_4173C4    dd ?
.data:004173C4
.data:004173C8 dword_4173C8    dd ?
.data:004173C8
.data:004173CC dword_4173CC    dd ?
.data:004173CC
.data:004173D0 dword_4173D0    dd ?
```

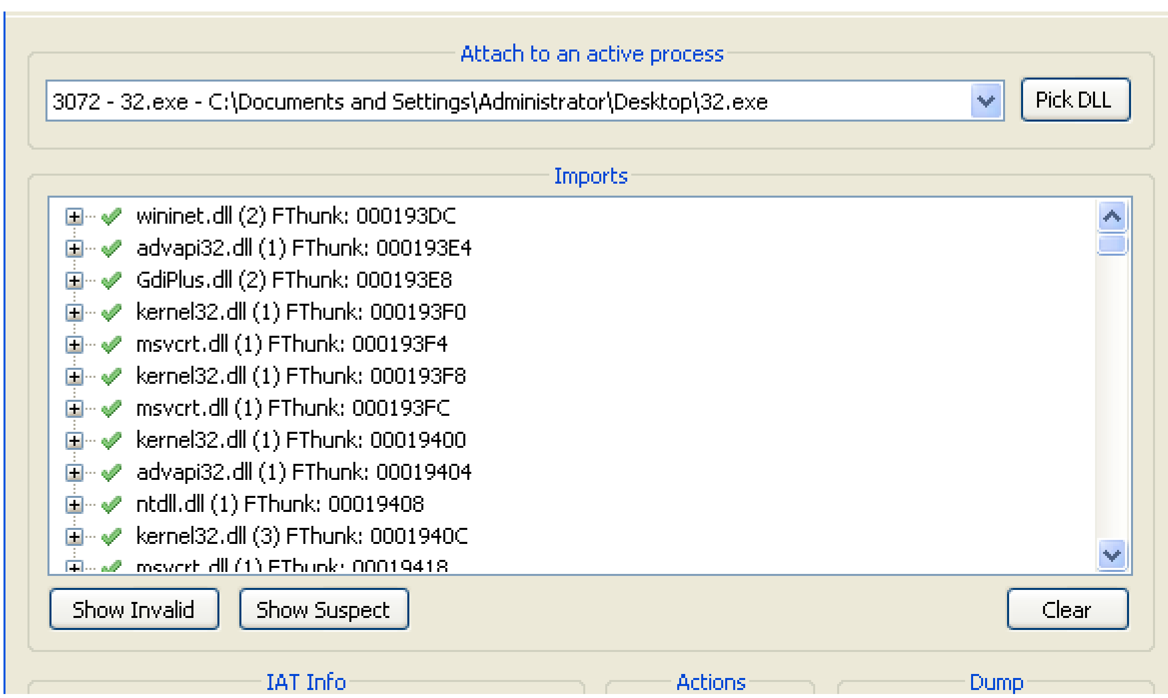
Resolved API Function Calls

We will use the following script to XOR the API address array with the original global XOR key. This allows us to patch and relocate the instructions.

```
# Raashid Bhat
# (C) PhishLabs 2017
# IAT Patch Script Qadars Banking Trojan

XORKey = 0x43B9A447 # 2017 v3
LoadLibException = 0x004196F0

ApiResolvRange = 0x00406150

ApiResolvRangeLen = 0x00409ACC - 0x00406150

from capstone import *
import struct
Debug = 1

def ReadMem(addr, n):
global Debug
if Debug:
return DbgRead(addr, n)
else:
return GetManyBytes(addr, n)
def WriteMem(addr, buff):
global Debug
if Debug:
        DbgWrite(addr, buff)
else:
for i in buff:
            PatchByte(addr, ord(i))
            addr = addr + 1
return



def PatchIndirectCall(MemAddr, Addrs, CallDst):
    Reg = ''
    md = Cs(CS_ARCH_X86, CS_MODE_32)
for i in md.disasm(MemAddr, Addrs):
print "0x%x:t%st%s" %(i.address, i.mnemonic, i.op_str)



if i.mnemonic == 'xor' and Reg == '':
print i.op_str[0:3]
            Reg = i.op_str[0:3]
if i.mnemonic == 'call':
if i.op_str == Reg:

                                                print "0x%x:t%st%s" %(i.address,
i.mnemonic, i.op_str)
                print "Size = %d"  % (i.address - ( Addrs + 6))
                Inst = ReadMem(Addrs + 6, (i.address - ( Addrs + 6))) # read
remaining instructions
                WriteMem( Addrs , 'x90' *  (i.address - ( Addrs) + 2)) # write NOPS
                WriteMem(Addrs, Inst)
                Inst = "xffx15" + struct.pack("
```

```
                WriteMem(i.address  - 6, Inst)
return


for i in range(0x004193DC, 0x004196F0, 4):


    PatchDword(i, DbgDword(i)  ^ XORKey)


if i == LoadLibException:
continue


    x = XrefsTo(i)

    for j in x:
        addr = j.frm
        print addr
        if addr > ApiResolvRange and addr             print "[] API Patch Subroutine
Skipping... "
            continue
        print hex(j.frm)
        PatchIndirectCall(ReadMem(addr, 0x32), addr, i)
```

Script to Patch API Address Array



Patching

Import Address Table

Upon opening this file in IDA, we are presented with an annotated Import Address Table:

```
if ( hConnect )
{
  dwFlags = -2071973376;
  lpOptional = (LPVOID)StringDecode((int *)"HTTP/1.1");
  v6 = HttpOpenRequestA(hConnect, lpszVerb, lpszObjectName, (LPCSTR)lpOptional, szReferrer, 0, dwFlags, 0);
  v17 = v6;
  HeapFreeX(&lpOptional);
  if ( v6 )
  {
    dwFlags = 600000;
    InternetSetOptionA(v6, 6u, &dwFlags, 4u);
    dwBufferLength = 4;
    dwFlags = 0;
    InternetQueryOptionA(v6, 0x1Fu, &dwFlags, &dwBufferLength);
    dwFlags |= 0x100u;
    InternetSetOptionA(v6, 0x1Fu, &dwFlags, 4u);
```

Address	Ordinal	Name	Library
004193DC		HttpAddRequestHeadersA	wininet
004193E0		HttpQueryInfoA	wininet
004193E4		OpenProcessToken	advapi32
004193E8		GdipDrawImageI	GdiPlus
004193EC		GdipCreateFromHDC	GdiPlus
004193F0		GetExitCodeThread	kernel32
004193F4		_time64	msvcrt
004193F8		WaitForSingleObject	kernel32
004193FC		wcscpy	msvcrt
00419400		LeaveCriticalSection	kernel32
00419404		GetTokenInformation	advapi32
00419408		NtQueryInformationProcess	ntdll
0041940C		VirtualAlloc	kernel32
00419410		ResetEvent	kernel32
00419414		QueueUserAPC	kernel32
00419418		wcslen	msvcrt
0041941C		HttpOpenRequestA	wininet
00419420		RegCloseKey	advapi32
00419424		strcmp	msvcrt
00419428		GetFileInformationByHandle	kernel32

Patched Import Address Table in IDA

Similarly, we can use an IDA script to deal with Qadars string obfuscation which is simply a XOR-based decoding algorithm in which each of the encoded strings has the following structure:

```
struct EncodedString
{
        DWORD len;
char Encodedbuf[len]; // XOR encoded with a key
}
```

```
XORKEY = “4B57A7E012368BE9AA48” // found in sample

    while ( v12      {
      *(_BYTE *)(v12++ + v13) ^= v15[v14];
      v14 = (v14 + 1) % v11;
    }
    result = v13;
```

The code can be simply represented in Python as follows:

```
def DecodeString(Ea):
    XORBuff = "4B57A7E012368BE9AA48".decode("hex")

    BuffLen = Dword(Ea)
print "[] Buffer Len = %d " % BuffLen
    dst = "



for i in range(0, BuffLen):
        dst = dst + chr( (Byte(Ea + 4 + i) & 0xff) ^ ord(XORBuff[i % (10)]))
print len(dst)
    j = 0
for i in dst:
        PatchByte(Ea + j, ord(i))
        j = j + 1
```

We will use the following IDA Python script to help us with decoding all encoded strings present in Qadars:

```
# IDAPython String Decoder For Qadars
# Raashid Bhat
# (C) PhishLabs 2017

import struct
procesed = []
def DecodeString(Ea):
    XORBuff = "4B57A7E012368BE9AA48".decode("hex") #xorkey

    BuffLen = Dword(Ea)
print "[] Buffer Len = %d " % BuffLen
    dst = "



for i in range(0, BuffLen):
        dst = dst + chr( (Byte(Ea + 4 + i) & 0xff) ^ ord(XORBuff[i % (10)]))
print len(dst)
    j = 0
for i in dst:
        PatchByte(Ea + j, ord(i))
        j = j + 1



for i in CodeRefsTo(ScreenEA(),1):
print hex(i)
    ea = PrevAddr(i)
while "push    offset" notin GetDisasm(ea):
        ea = PrevAddr(ea)
print GetDisasm(ea)[19:]
if "asc_" in  GetDisasm(ea):
        addr = GetDisasm(ea)[19:].split(";")[0]
else:
        addr = GetDisasm(ea)[19:]
if int(addr, 16) in procesed:
continue



    DecodeString(int(addr, 16))
    procesed.append(int(addr, 16))

for i in procesed:
   MakeStr(i, BADADDR)
```

Running this script on the sample decodes all strings and makes them visible in the Strings window.

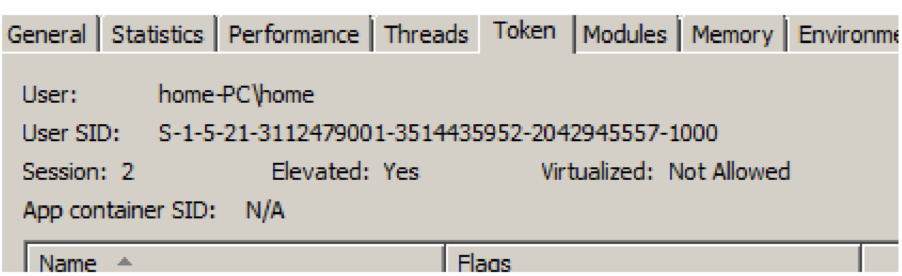
Address	Length	Type	String
.rdata:004129FC	00000017	C	Windows Server 2012 SP
.rdata:00412A24	0000000D	C	Windows 8 SP
.rdata:00412A44	0000001A	C	Windows Server 2012 R2 SP
.rdata:00412A70	0000000F	C	Windows 8.1 SP
.rdata:00412A90	0000000D	C	kdwTimestamp
.rdata:00412A9D	00000005	C	dData
.rdata:00412AA3	00000008	C	fLength
.rdata:00412AAB	00000009	C	flpData@
.rdata:00412AC4	00000008	C	gBitness
.rdata:00412AE0	0000000A	C	hmainType
.rdata:00412AEA	00000009	C	gsubType
.rdata:00412B04	0000000D	C	klpszVersion
.rdata:00412B24	0000000D	C	�ilpszBotIDx
.rdata:00412B44	00000008	C	flpData
.rdata:00412B5C	00000008	C	fLength
.rdata:00412B74	00000006	C	dData
.rdata:00412B8C	0000000D	C	kdwTimestamp
.rdata:00412BAC	0000000A	C	hdwStatus
.rdata:00412BC8	00000007	C	elmage
.rdata:00412BE0	0000000C	C	jmoduleSize
.rdata:00412BFC	0000000F	C	mlpProcessList
.rdata:00412C1C	0000000E	C	lProcessCount
.rdata:00412C3C	0000000C	C	jInjectType
.rdata:00412C58	0000000A	C	hAutoLoad
.rdata:00412C74	0000000C	C	jMainModule
.rdata:00412C90	0000000B	C	iszVersion
.rdata:00412CAC	00000008	C	fszName
.rdata:00412CC4	0000000A	C	hModule64
.rdata:00412CE0	0000000A	C	hModule32

Deobfuscated

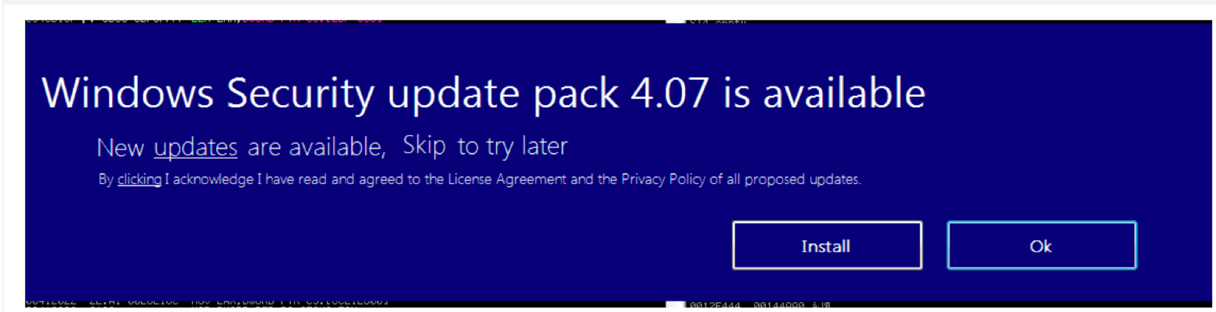
Strings

Privilege Escalation / Social Engineering and Spoofing Adobe Update

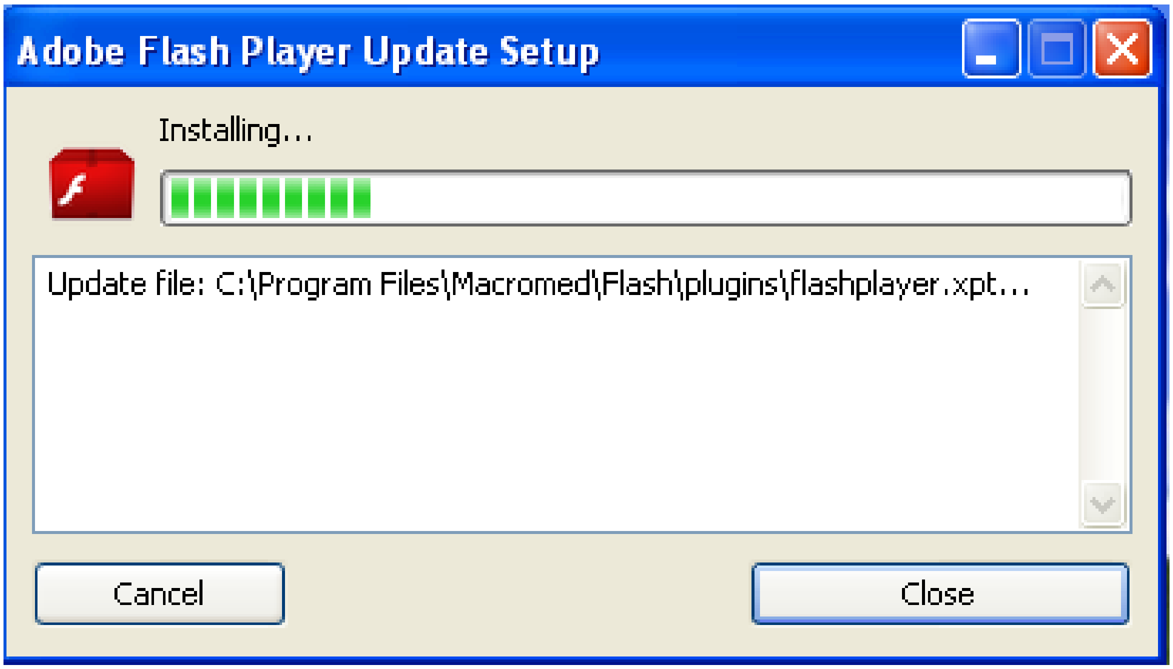
If Qadars is not presented with a specific set of privileges, it tries to contact and download a module from the command and control center. This module is then loaded in memory and an export, aptly named "Exploit" is invoked to complete the privilege escalation. Currently, a known vulnerability in how the Win32k.sys kernel-mode driver handles objects in memory is exploited for this purpose (CVE-2015-1701).



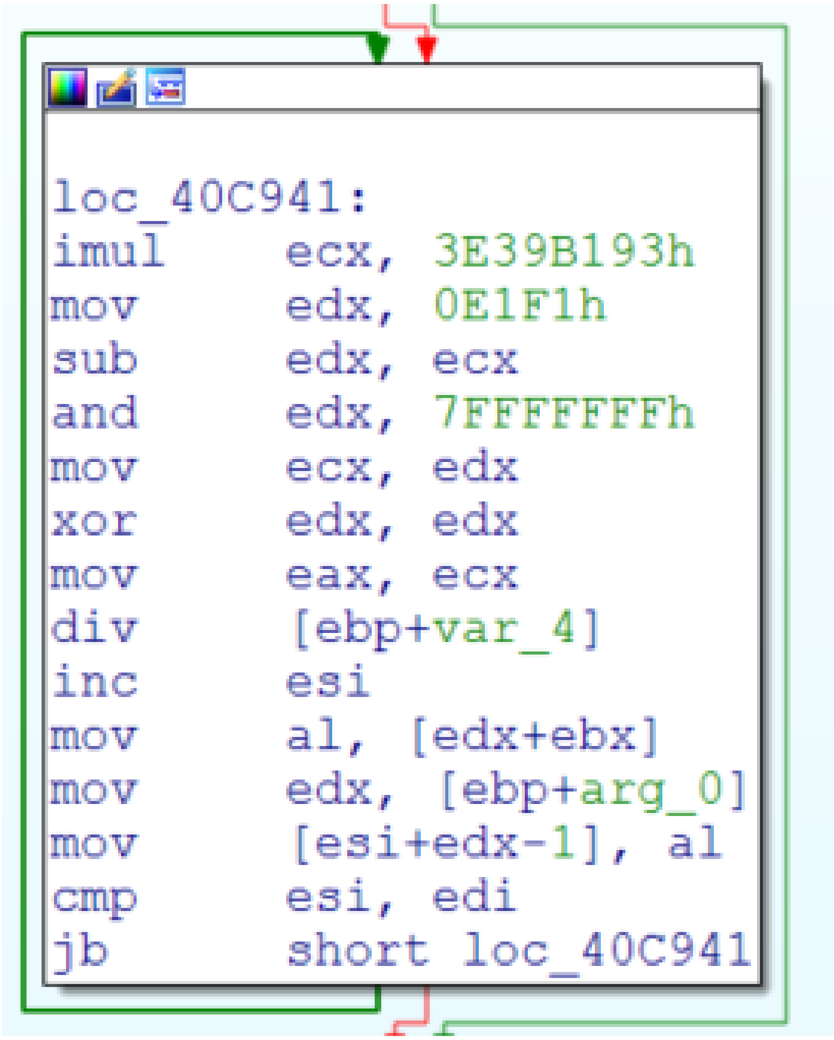
Decoding 'Exploit' Module



Debugging Symbols for 'Exploit' Module

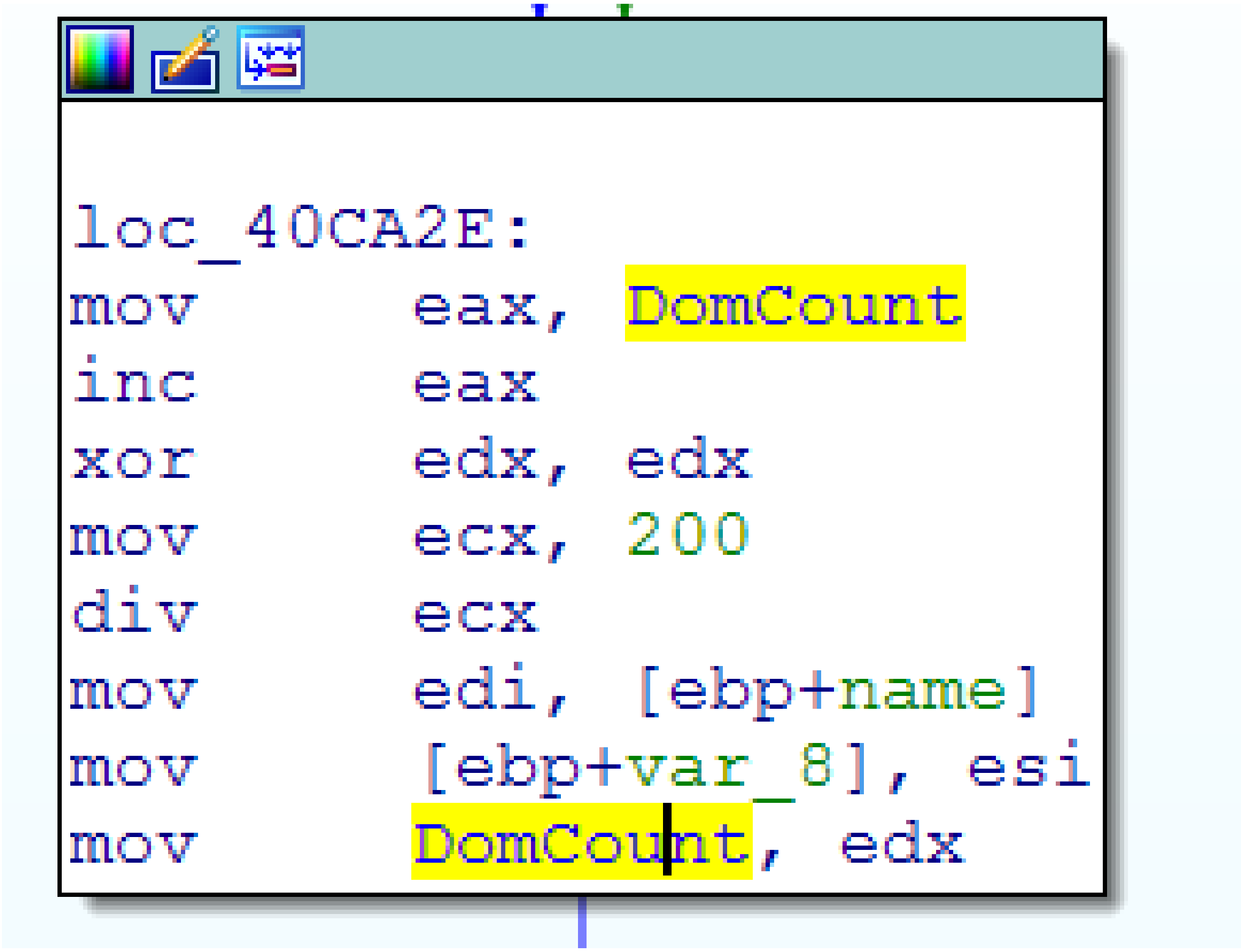


'Exploit' Module in DLL Exports



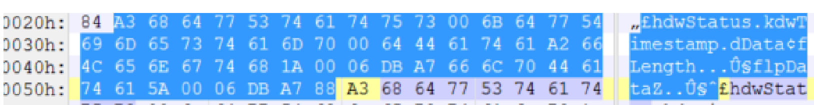
Elevated Permissions Following Invocation of 'Exploit' Module

If the privilege escalation code does not work, Qadars attempts to socially engineer the victim with a fake Windows security update prompt. This executes code that allows Qadars to run with higher privileges using the "runas" verb:



Fake Windows Security Prompt

Upon execution of the malware, it loads a fake window with a progress bar masquerading as an Adobe Updater application to provide a sense of legitimacy.



Fake Adobe Flash Update

Communication and DGA

Qadars locates the command and control center by generating a list of 200 domains using a combination of a time seed and some constants. On February 1st, Qadars started using a new seed value **0xE1F1**, replacing the previous seed, **0xE1F2.**

Qadars Domain Generation 1.png Qadars Domain Generation 2.png

Domain Generation

Initially, two information packets are generated and concatenated. They consist of a chunk of information serialized in the following format: *botid, version , operation type, etc.*

This information is packed together and fed to another subroutine which generates a MD5 hash of a 9-byte random string. This string will be used as an AES-128 encryption key which is then appended in the beginning of the encoded packet for command and control traffic decoding.

Information is serialized in each entry in the following format:

```
struct InfoStructEntry
{
unsigned int len;
unsigned char Buffer[len];
}
```

The response is encrypted using AES-128 and the first 16 bytes consist of the MD5 hash of the command and control buffer. This hash is used for verification before processing.

```
Struct c2packet
{
BYTE MD5Hash[16];
BYTE []AESEncryptedBuffer;
}
```

After decryption, the base packet consists of metadata information which is used to determine the parameters and type of block to be processed. Multiple entries consist of either modules, updates, or a web inject file which is APLIB compressed.

Qadars Base Packet.png

Yara rule

The following Yara rule can be used to identify this Qadars variant:

```
rule Qadars
 {
     strings:
         $dga_function = { 69 C9 93 B1 39 3E BE F1 E1 00 00 2B F1 81 E6 FF FF FF 7F
B8 56 55 55 55 F7 EE 8B C2 C1 E8 1F 03 C2 8D 04 40 }
      condition:
         $dga_function
 }
```