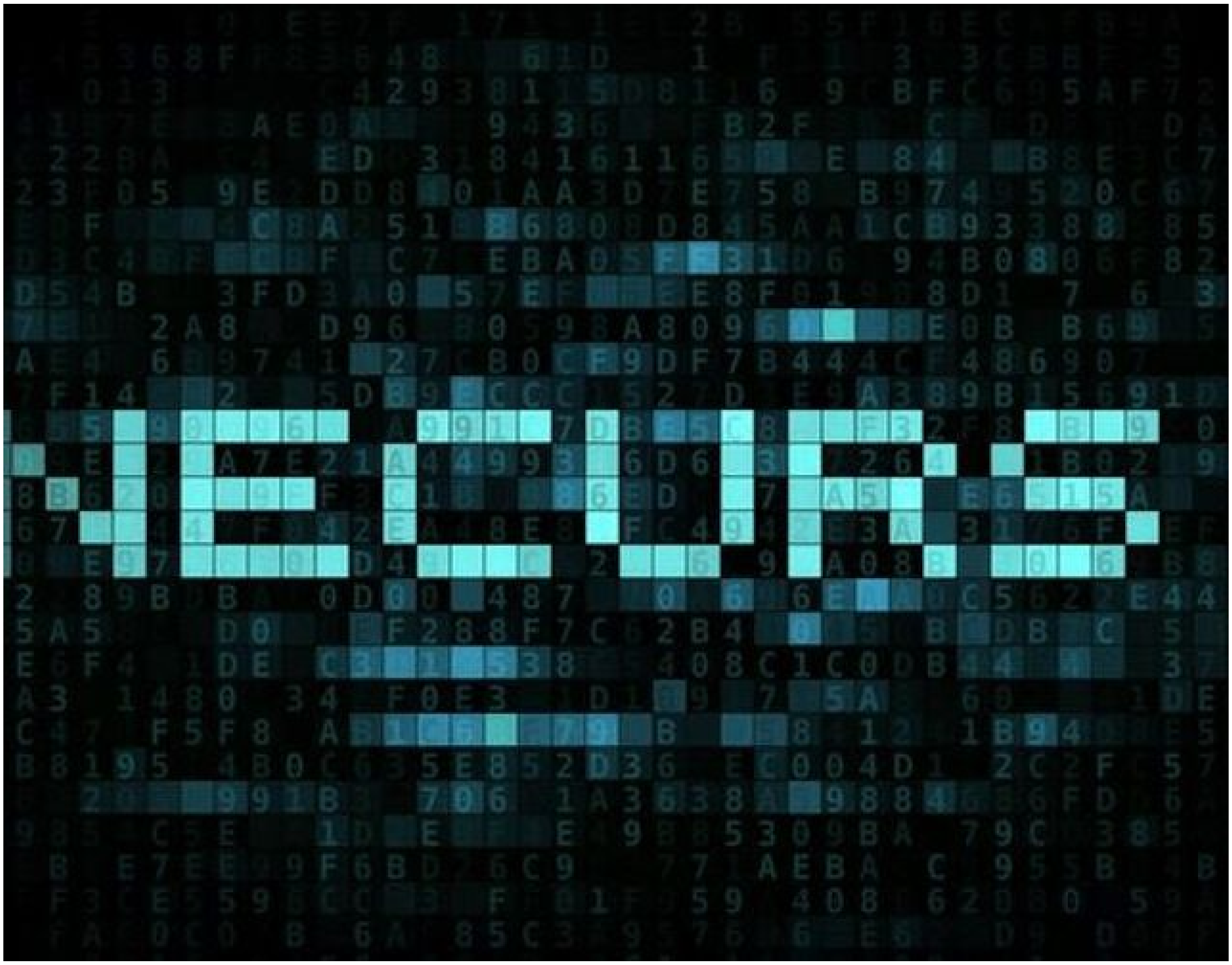
# The DGAs of Necurs

bin.re/blog/the-dgas-of-necurs/




Necurs is a malware that opens a backdoor on infected systems, see NECURS: The Malware That Breaks Your Security. A broad analysis of the malware can be found in the three part series The Curse of Necurs by Peter Ferrie.

This post focuses exclusively on the network traffic of Necurs, in particular the used domains. Necurs features three different sets of hostnames that serve different purposes. Although all three sets reek of domain generation algorithm (DGA), only the first and last set are actually generated algorithmically. The following example traffic shows the different stages of the callback attempts with number 3, 4 and 6 marking the three malicious domains batches:

```
1.----  13:09:39.635819000  facebook.com

    +-  13:09:39.638039000  0.pool.ntp.org
2.--|   13:09:39.998526000  1.pool.ntp.org
    +-  13:09:40.020627000  2.pool.ntp.org

    +-  13:09:40.027471000  sxotmrxwhddr.com
    |   13:09:40.028494000  btysiiquuc.com
3.--|   13:09:40.032591000  kfncxvayakmb.com
    +-  13:09:40.033539000  vmslcvvocseu.com

    +-  13:09:40.688199000  qcmbartuop.bit
    |   13:09:41.699491000  qcmbartuop.bit
4.--|   13:09:42.716128000  qcmbartuop.bit
    |   (... 16 times total)
    +-  13:16:32.364351000  qcmbartuop.bit

5.----  13:16:49.187559000  facebook.com

    +-  13:16:49.516477000  boymlujtgp.nu
    |   13:16:49.520651000  ybynentfsjvmsgtktcoog.im
    |   13:16:49.527701000  oiijxplrnmvgskxwaye.ru
6.--|   13:16:49.529669000  imgirmyddbsniuh.pw
    |   13:16:49.834913000  ultrttvbvjaanrj.jp
    |   (... 2048 total)
    +-  13:18:03.607552000  porgtemsbycy.ki
```

The following list gives a brief summary of the source and purpose of the domains. See the respective section for an in-depth analysis of the three DGA sets.

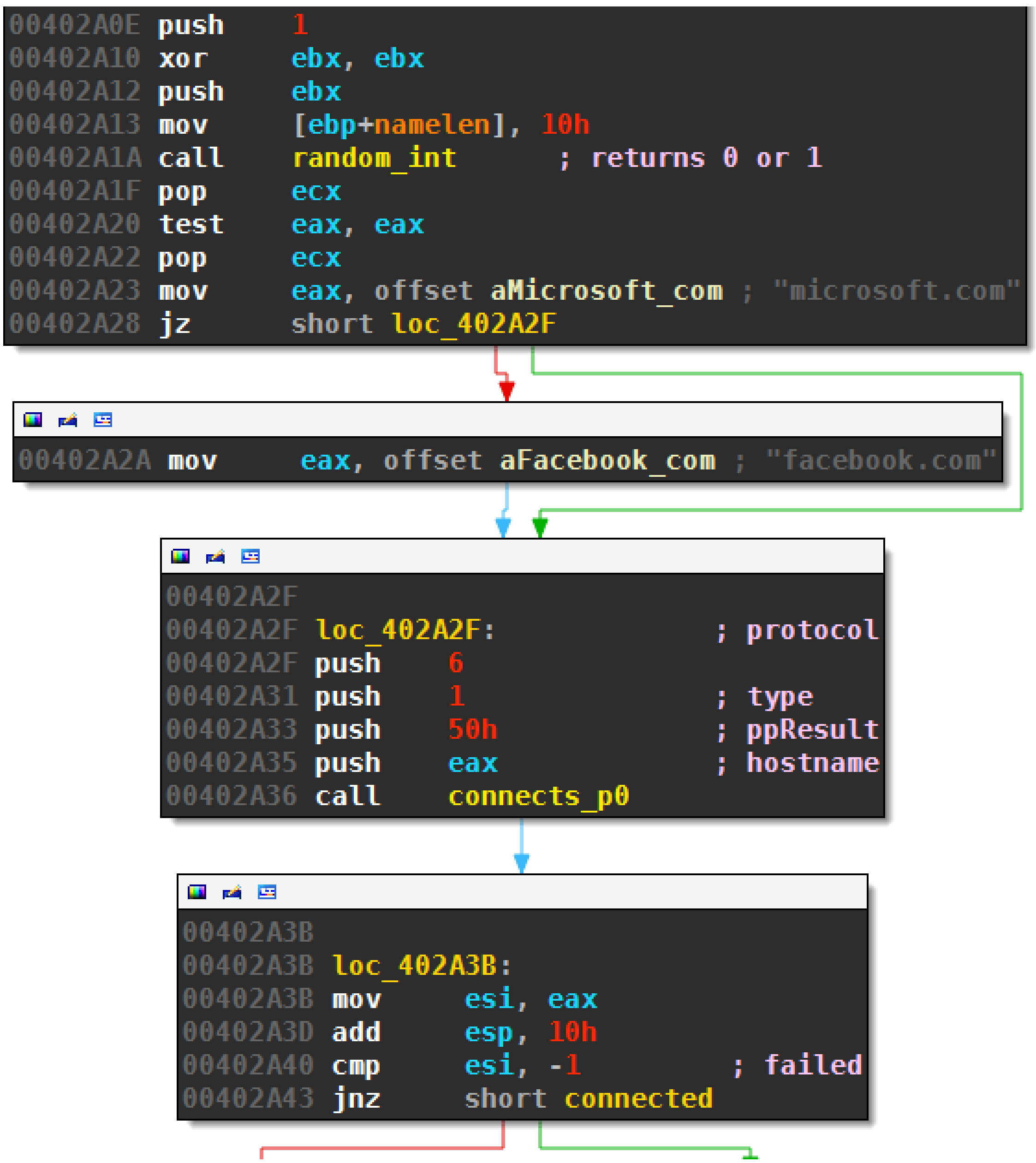
1. Necurs starts by checking internet connectivity by resolving *facebook.com* or *microsoft.com*.
2. The malware then contacts three NTP pools to get the accurate date and time.
3. Next, four DGA domains are generated by the **first dga** with top level domain *.com*. The domains are unpredictable and can therefore not be sinkholed or used to identify Necurs samples when taken in isolation. The purpose of the domains is to detect simulated internet in lab environments
4. If the lab detection test passes, Necurs will try sixteen domains from a hard-coded list of pseudo-DGA domains, see **second dga**. Necurs will reuse domains if the list contains less than 16 domains. The analysed sample, for example, only has one hard-coded domain which will therefore be repeated 16 times. Necurs sleeps between 1 and 20 seconds after each failed connection attempt, which mounts up to about 5 minutes before the malware moves on to the next stage. The hard-coded domains are probably the main C&C domains.
5. Another connectivity check to *facebook.com* or *microsoft.com* is made before Necurs resorts to the last DGA.

6. The **third dga** finally checks up to 2048 different domains. One special feature is the large list of 43 different top level domains, some of which are quite exotic. The DGA is time-dependent — the domains change every four days. The DGA is probably a backup in case the set of hard-coded domains no longer work.

## The First DGA

### Connectivity Checks

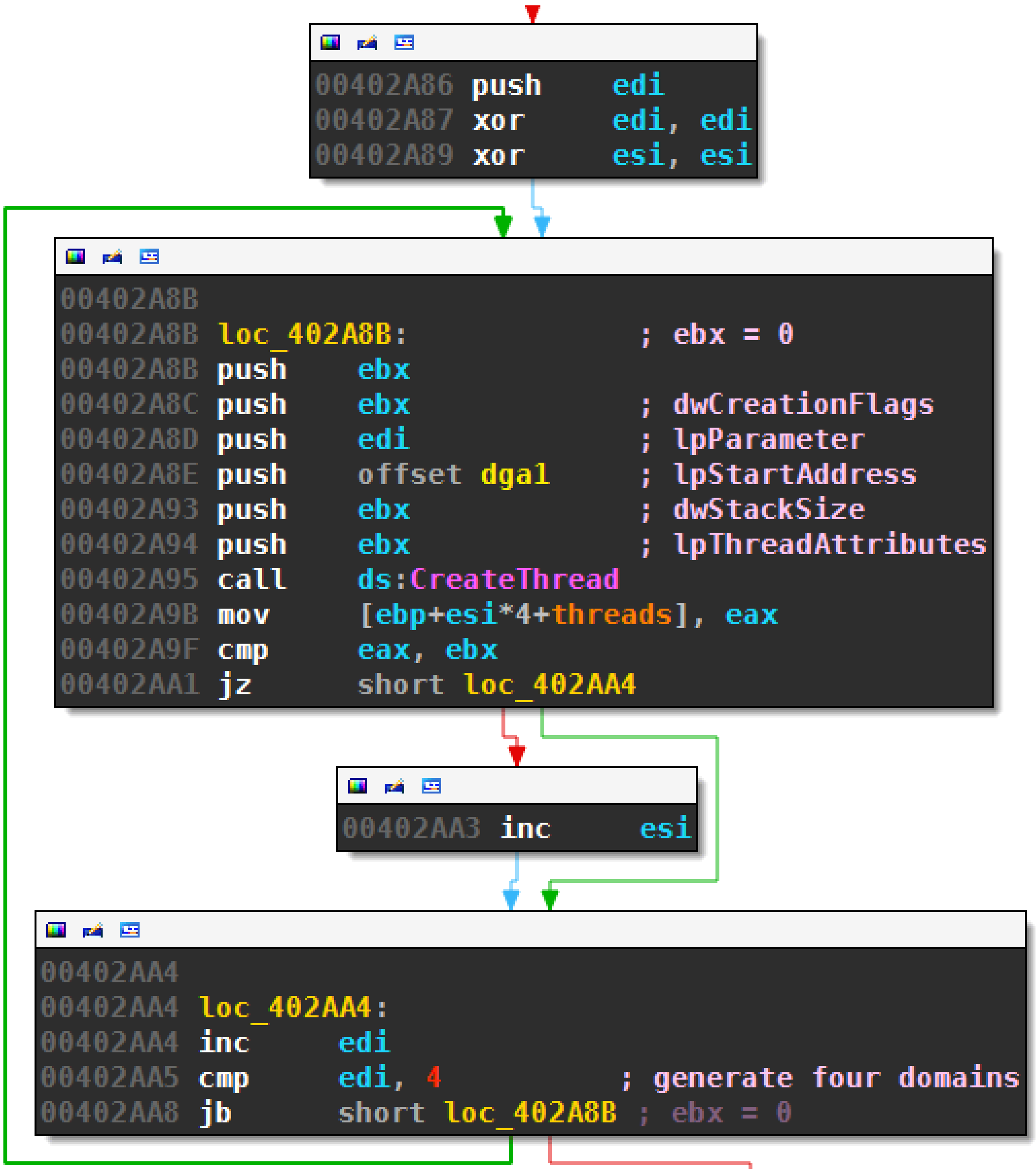
Necurs makes a connectivity check to either *facebook.com* or *microsoft.com* before the first set of DGA domains are tested:



The routine `random_int(0,1)` returns 0 or 1 (the implementation of `random_int` is discussed below). The malware aborts the callback attempt if it can't resolve and contact the domain of Facebook or Microsoft as the case may be.

## DGA Caller

Necurs then launches four threads that attempt to resolve domains generated by the first DGA:




There is no delay between creating the threads, the four DNS queries happen at around the same time.

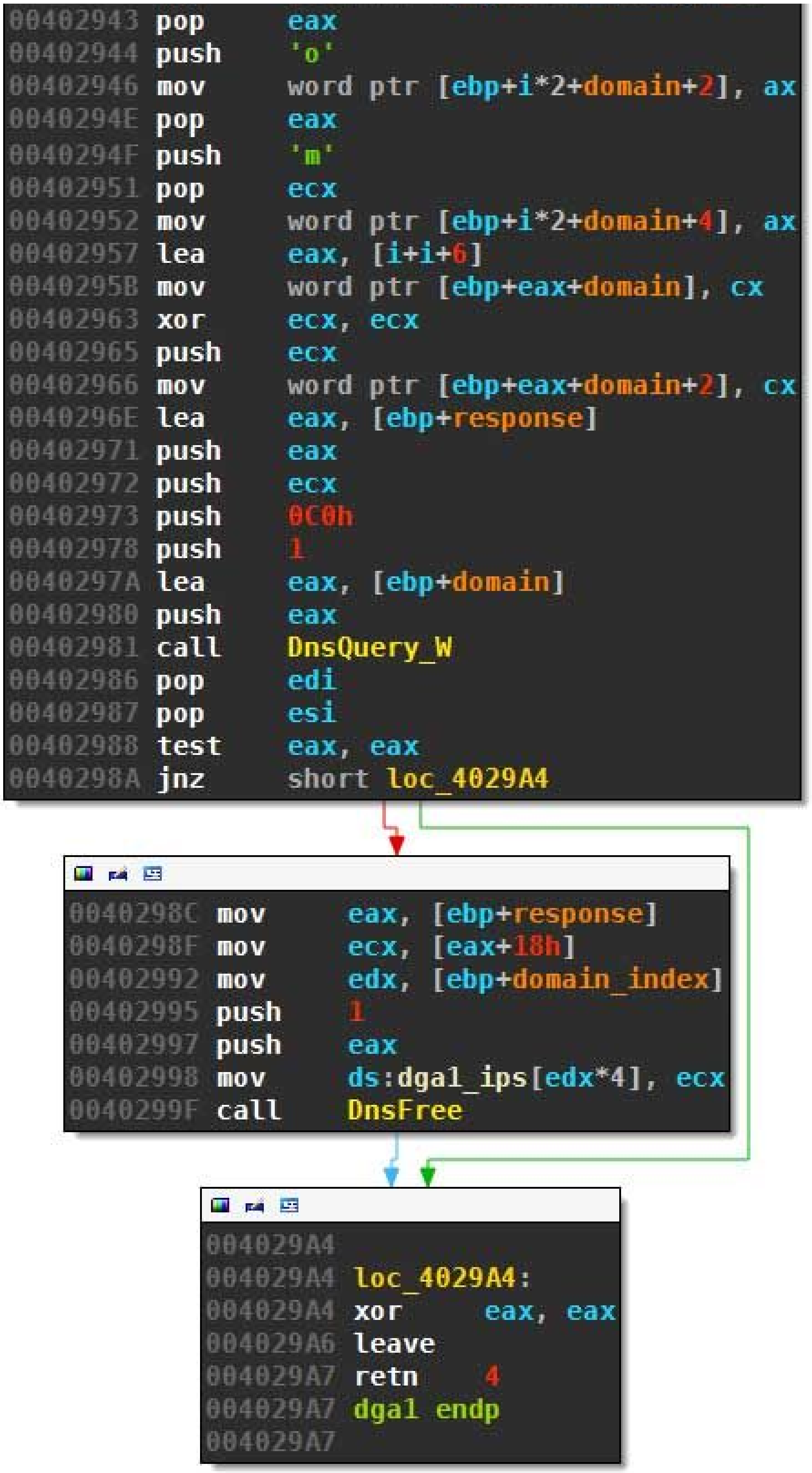
## The Heart of the First DGA

Let's see how the domains are generated by `dga1`:

```
00402900
00402900
00402900 ; Attributes: bp-based frame
00402900
00402900 ; DWORD __stdcall dga1(LPVOID domain_index)
00402900 dga1 proc near
00402900
00402900 domain= dword ptr -84h
00402900 response= dword ptr -4
00402900 domain_index= dword ptr  8
00402900
00402900 domain_len = edi
00402900 push    ebp
00402901 mov     ebp, esp
00402903 sub     esp, 84h
00402909 push    esi
0040290A push    domain_len
0040290B push    15              ; max domain length
0040290D push    10              ; min domain length
0040290F i = esi
0040290F xor     i, i
00402911 call    random_int
00402916 domain_len = edi
00402916 mov     domain_len, eax
00402918 pop     ecx
00402919 pop     ecx
0040291A test    domain_len, domain_len
0040291C jz      short loc_402936
```

```
0040291E
0040291E loc_40291E:
0040291E push    'z'
00402920 push    'a'
00402922 call    random_int
00402927 mov     word ptr [ebp+i*2+domain], ax
0040292F inc     i
00402930 pop     ecx
00402931 pop     ecx
00402932 cmp     i, domain_len
00402934 jb      short loc_40291E
```

```
00402936
00402936 loc_402936:
00402936 push    '.'
00402938 pop     eax
00402939 push    'c'
0040293B mov     word ptr [ebp+i*2+domain], ax
```




This very simple algorithm first randomly determines the length of the second level domain to be between 10 and 15 characters. It then builds the domain by picking uniformly at random from all lowercase letters and appending the hard-coded top level domain *.com*. The DGA concludes with querying the resulting domain; the resolved IP, if there is any, is appended to the array `dga1_ips`.

## Pseudorandom Number Generator

The connectivity check, as well as `dga1`, use `random_int` to generate random integers. The disassembly of this routine is:


This is just a mapping of the return value of `random_mwc` to the desired range:

```
function random_int(lower, upper)
    if lower > upper then
        return 0
    else
        return lower + random_mwc % (upper - lower + 1)
```

The routine `random_mwc` is an almost standard implementation of a lag-3 multiply-with-carry pseudorandom number generator invented by George Marsaglia:

```
00404869
00404869
00404869
00404869 random_mwc proc near
00404869 rdtsc
0040486B mov     ecx, eax
0040486D mov     eax, ds:rng1
00404872 push    esi
00404873 mov     esi, edx
00404875 mov     edx, 916905990
0040487A mul     edx
0040487C add     ecx, eax
0040487E mov     eax, ds:rng2
00404883 adc     esi, edx
00404885 xor     edx, edx
00404887 add     ecx, ds:c
0040488D mov     ds:rng1, eax
00404892 mov     eax, ds:rng3
00404897 adc     esi, edx
00404899 mov     ds:rng2, eax
0040489E mov     ds:c, esi
004048A4 mov     ds:rng3, ecx
004048AA mov     eax, ecx
004048AC pop     esi
004048AD retn
004048AD random_mwc endp
004048AD
```

The only difference is the addition of an `rdtsc` summand, which will add the current time stamp counter to all generated numbers (not just the initial seed). The initial seed values and initial carry are:

```
0040F048 rng3            dq  77465321
0040F04C c               dd     13579
0040F050 rng2            dd 362436069
0040F054 rng1            dd 123456789
```

The choice of these values is pretty common. Necurs never changed these values in its long existence according to the aforementioned article “the curse of Necurs”. This is the PRNG `random_mwc` in pseudo-code is:

```
b = 2^(32)
a = 916905990
// Initial seeds and carry
rng1 = 123456789
rng2 = 362436069
rng3 = 77465321
c = 13579

function random_mwc()
    t = a*rng1 + rdtsc() + c
    rng1 = rng2;
    rng2 = rng3;
    rng3 = t;
    c = (t // b) % b
    return v1;
```

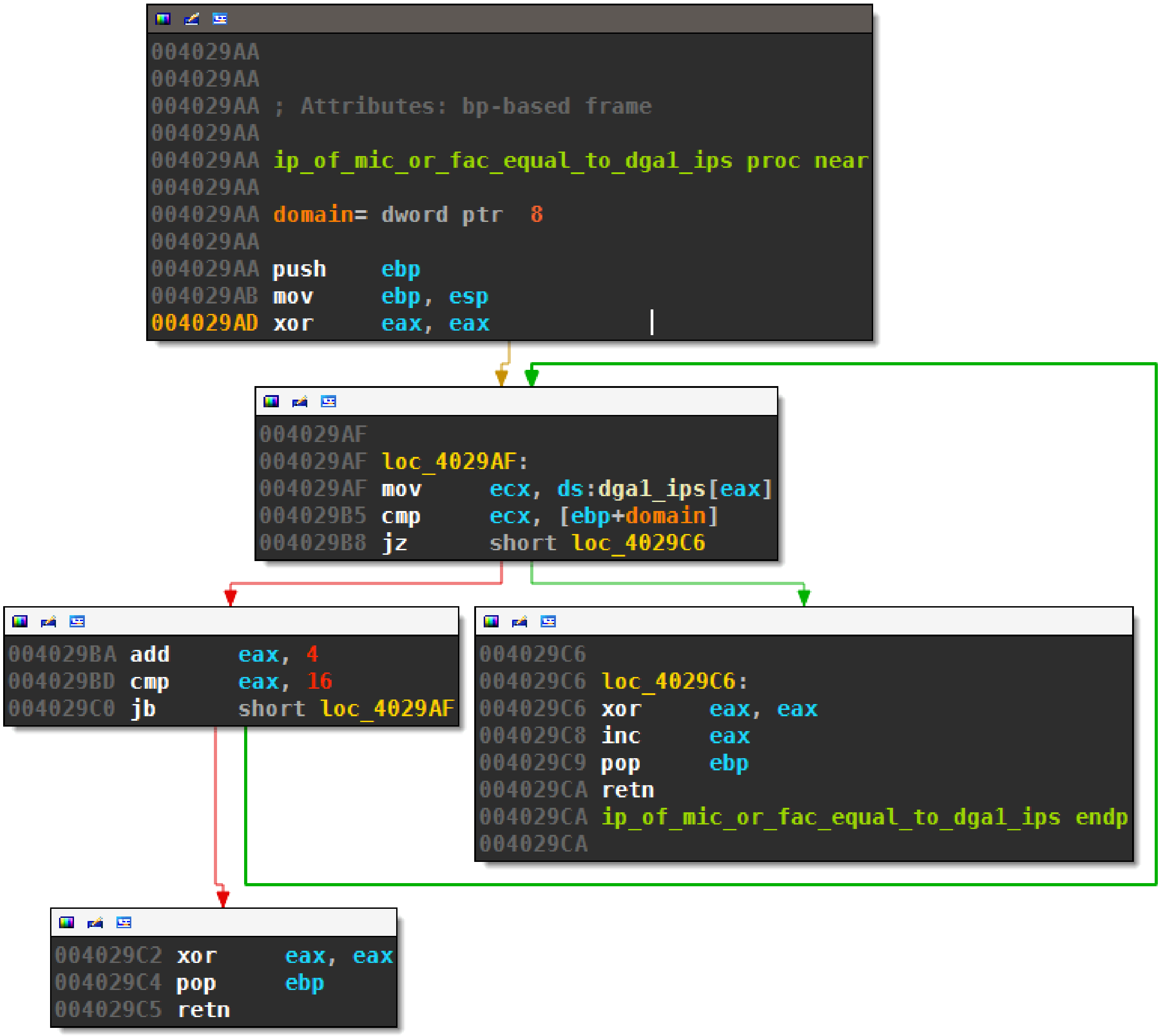
The addition of `rdtsc` makes this random number generator, and therefore also the first domain generation algorithm, virtually impossible to predict.

## Comparison with Facebook's or Microsoft's IP

Although the domains of `dga1` are unusable as callback targets, they still serve a purpose: If one of the DGA IPs matches the one of Facebook or Microsoft (resolved during connectivity checking), Necurs will abort the callback attempt:

```
00402ACE
00402ACE loc_402ACE:                 ;
00402ACE push    dword ptr [ebp+var_3C.sa_data+2] ; ip response of connectivity check
00402ACE                             ; to microsoft.com or facebook.com
00402AD1 call    ip_of_mic_or_fac_equal_to_dga1_ips
00402AD6 pop     ecx
00402AD7 pop     edi
00402AD8 test    eax, eax
00402ADA jnz     fail
```

The excessively long named routine `ip_of_mic_or_fac_equal_to_dga1_ips` compares the IP of Microsoft or Facebook to the IPs of `dga1` stored in array `dga1_ips`:



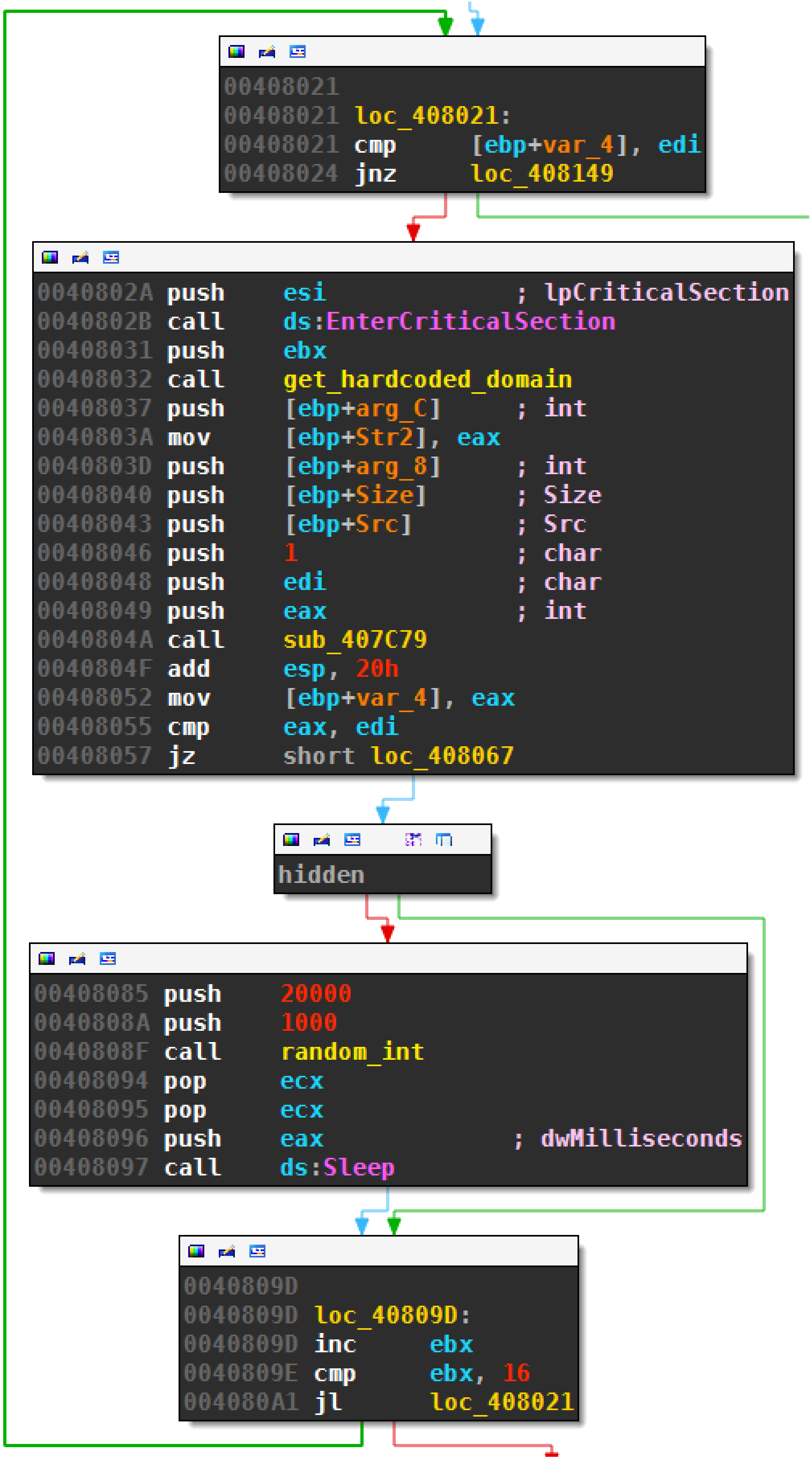

I suspect that Necurs tries to detect simulated internet services this way. Simulated Internet will typically return the same IP for all requested domains. The 4 DGA domains, however, are unpredictable and therefore non-existent. And even if the victim's ISP uses DNS hijacking, the returned IPs will not match Facebook's or Microsoft's IP.

## Summary

The first DGA of Necurs generates four *.com* domains of 10 to 15 lower case letters. It tries to resolve all four domains in parallel. The modified MWC-PRNG makes the domains unusable as callback targets; instead they are probably used to detect simulated internet connections in a lab environment.

## The Second DGA

Necurs will continue without break with the second set of DGA domains in case the lab detection passed. The second set of DGA-like domains, for example *qcmbartuop.bit*, are not generated algorithmically but hard-coded. The following loop fetches 16 of those hard-coded domains and tries to contact them:


00408021
00408021 loc_408021:
00408021 cmp [ebp+var_4], edi
00408024 jnz loc_408149
0040802A push esi ; lpCriticalSection
0040802B call ds:EnterCriticalSection
00408031 push ebx
00408032 call get_hardcoded_domain
00408037 push [ebp+arg_C] ; int
0040803A mov [ebp+Str2], eax
0040803D push [ebp+arg_8] ; int
00408040 push [ebp+Size] ; Size
00408043 push [ebp+Src] ; Src
00408046 push 1 ; char
00408048 push edi ; char
00408049 push eax ; int
0040804A call sub_407C79
0040804F add esp, 20h
00408052 mov [ebp+var_4], eax
00408055 cmp eax, edi
00408057 jz short loc_408067
hidden
00408085 push 20000
0040808A push 1000
0040808F call random_int
00408094 pop ecx
00408095 pop ecx
00408096 push eax ; dwMilliseconds
00408097 call ds:Sleep
0040809D
0040809D loc_40809D:
0040809D inc ebx
0040809E cmp ebx, 16
004080A1 jl loc_408021

Necurs sleeps 1 to 20 seconds after each failed attempt, which sums up to a average total sleep time of about 5 minutes for all 16 attempts.

The relevant snippet of `get_hard-coded_domain` is:

```
00407539 mov     eax, ds:domain_index
0040753E xor     edx, edx
00407540 div     ds:total_nr_of_domains
00407546 inc     ds:domain_index
0040754C mov     eax, ds:hardcoded_domains[edx*4]
00407553 pop     ebp
00407554 retn
```

This returns the hard-coded domains one after another, starting over with the first domain if necessary.

From what I could gather from other reports, the hard-coded domains seem to mostly use the special *.bit* pseudo top level domain served by the namecoin project. The domains are probably Necurs` main C&C targets.

## The Third DGA

If the second set of domains also fails to produce a working C&C server, Necurs tries one last set of domains.

### Sequence Numbers

The third DGA starts by creating an array of 2048 sequence numbers 0 to 2047; these sequence numbers are then randomly permutated:

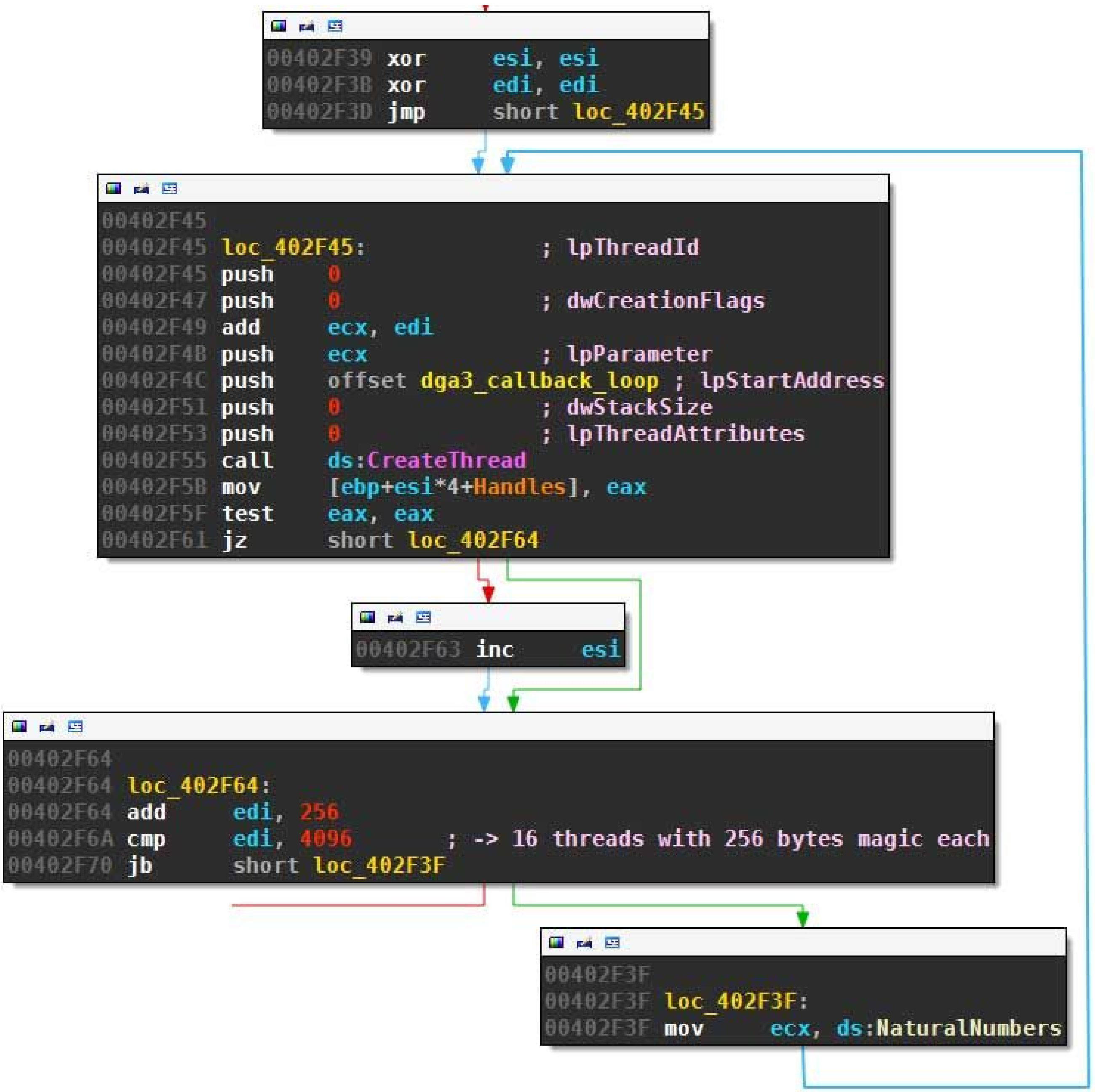

The permutation routine randomizes the sequence numbers in place with the following algorithm:

```
n = A.length
for i = 1 to n
    swap A[i] with A[Random(1,n)]
```

Although this randomizes the sequence numbers, it does not do it uniformly. A better implementation would call `Random(i,n)` instead of `Random(1,n)`, see for instance exercise 5.3-3 on page 129 of "Introduction to Algorithms", 3rd edition. Nevertheless, the sequence numbers are still random and unpredictable because of the call to `random_mwc` which includes the current tick count see section above.

## DGA Caller

After the sequence numbers have been randomized, Necurs starts up 16 threads, each with 128 of the sequence numbers (referenced by `ecx`, indexed by `edi`):




Each thread will call the routine to generates the domains for all 128 sequence numbers it got assigned to (unless a callback is successful beforehand):

```
00402DCD mov     eax, [ebp+sequence_numbers]
00402DD0 movzx   eax, word ptr [eax+index*2]
00402DD4 push    eax             ; sequence number
00402DD5 lea     edi, [ebp+hostname]
00402DD8 call    dga3
00402DDD pop     ecx
00402DDE cmp     eax, 4
00402DE1 jbe     short loc_402E19
```

```
00402EB5
00402EB5 loc_402EB5:
00402EB5 inc     index
00402EB6 cmp     index, 128       ; how many domains per thread
00402EBC jb      loc_402DC0
```

## The DGA

At the heart of the third DGA we find this long, but easy to understand algorithm:

```
00402BA1
00402BA1
00402BA1 ; Attributes: bp-based frame
00402BA1
00402BA1 dga3 proc near
00402BA1
00402BA1 tlds= byte ptr -70h
00402BA1 SystemTime= _SYSTEMTIME ptr -10h
00402BA1 sequence_number_and_domain_len= dword ptr  8
00402BA1
00402BA1 push    ebp
00402BA2 mov     ebp, esp
00402BA4 sub     esp, 70h
00402BA7 push    esi
00402BA8 xor     esi, esi
00402BAA cmp     ds:magic_number, esi ; is zero for the first call
00402BB0 jnz     short loc_402BCC
```

```
00402BB2 push    2
00402BB4 push    esi
00402BB5 push    1363720480
00402BBA push    1208126542
00402BBF call    find_in_linked_list
00402BC4 add     esp, 10h
00402BC7 mov     ds:magic_number, eax
```

```
00402BCC
00402BCC loc_402BCC:
00402BCC lea     eax, [ebp+SystemTime]
00402BCF push    ebx
00402BD0 push    eax             ; lpSystemTime
00402BD1 call    get_system_time
```

```
00402BD6 movzx   eax, [ebp+SystemTime.wYear]
00402BDA cdq
00402BDB push    edx
00402BDC push    eax
00402BDD call    pseudo_random
00402BE2 mov     ecx, eax
00402BE4 movzx   eax, [ebp+SystemTime.wMonth]
00402BE8 mov     ebx, edx
00402BEA cdq
00402BEB add     ecx, eax
00402BED adc     ebx, edx
00402BEF add     ecx, 0AAAAh
00402BF5 adc     ebx, esi
00402BF7 push    ebx
00402BF8 push    ecx
00402BF9 call    pseudo_random
00402BFE mov     ecx, eax
00402C00 mov     ax, [ebp+SystemTime.wDay]
00402C04 shr     ax, 2
00402C08 movzx   eax, ax
00402C0B mov     ebx, edx
00402C0D cdq
00402C0E add     ecx, eax
00402C10 adc     ebx, edx
00402C12 push    ebx
00402C13 push    ecx
00402C14 call    pseudo_random
00402C19 xor     ecx, ecx
00402C1B add     eax, [ebp+sequence_number_and_domain_len]
00402C1E adc     edx, ecx
00402C20 push    edx
00402C21 push    eax
00402C22 call    pseudo_random
00402C27 xor     ebx, ebx
00402C29 add     eax, ds:magic_number
00402C2F adc     edx, ebx
00402C31 push    edx
00402C32 push    eax
00402C33 call    pseudo_random
00402C38 add     esp, 2Ch
00402C3B push    0
00402C3D push    0Fh
00402C3F mov     ebx, eax
00402C41 push    edx
00402C42 push    ebx
00402C43 mov     dword ptr [ebp+SystemTime.wSecond], edx
00402C46 call    mod64
00402C4B add     eax, 7
00402C4E mov     [ebp+sequence_number_and_domain_len], eax
00402C51 jz      short loc_402C9D
```

```
00402C53
00402C53 sec_lvl_char:
00402C53 xor     ecx, ecx
00402C55 mov     eax, esi
00402C57 add     eax, ebx
00402C59 adc     ecx, dword ptr [ebp+SystemTime.wSecond]
00402C5C push    ecx
00402C5D push    eax
00402C5E call    pseudo_random
00402C63 pop     ecx
00402C64 pop     ecx
00402C65 push    0
00402C67 push    19h
00402C69 mov     ebx, eax
00402C6B push    edx
00402C6C push    ebx
```

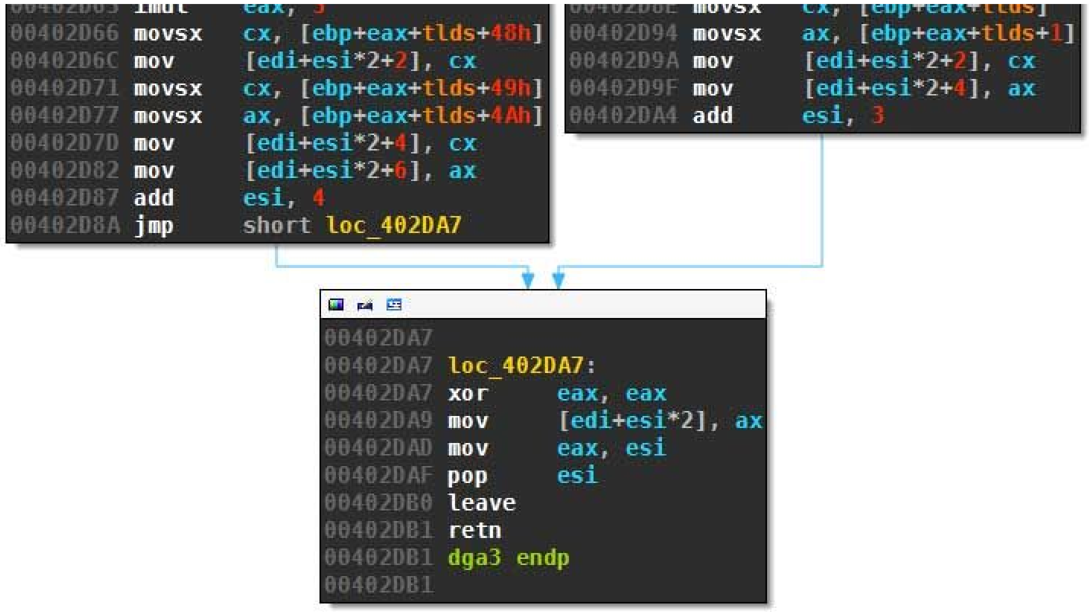
```
00402C6D mov     dword ptr [ebp+SystemTime.wSecond], edx
00402C70 call    mod64
00402C75 add     ax, 61h
00402C79 mov     [edi+esi*2], ax
00402C7D mov     eax, dword ptr [ebp+SystemTime.wSecond]
00402C80 add     ebx, 0ABBEDFh
00402C86 adc     eax, 0
00402C89 push    eax
00402C8A push    ebx
00402C8B call    pseudo_random
00402C90 inc     esi
00402C91 pop     ecx
00402C92 pop     ecx
00402C93 mov     ebx, eax
00402C95 mov     dword ptr [ebp+SystemTime.wSecond], edx
00402C98 cmp     esi, [ebp+sequence_number_and_domain_len]
00402C9B jb      short sec_lvl_char
```

```
00402C9D
00402C9D loc_402C9D:
00402C9D push    2Eh
00402C9F pop     eax
00402CA0 push    0
00402CA2 push    2Bh
00402CA4 push    dword ptr [ebp+SystemTime.wSecond]
00402CA7 mov     [edi+esi*2], ax
00402CAB push    ebx
00402CAC mov     dword ptr [ebp+tlds], 'nijt'
00402CB3 mov     dword ptr [ebp+tlds+4], 'wtpj'
00402CBA mov     dword ptr [ebp+tlds+8], 'mcca'
00402CC1 mov     dword ptr [ebp+tlds+0Ch], 'nmal'
00402CC8 mov     dword ptr [ebp+tlds+10h], 'hsos'
00402CCF mov     dword ptr [ebp+tlds+14h], 'uncs'
00402CD6 mov     dword ptr [ebp+tlds+18h], 'umfn'
00402CDD mov     dword ptr [ebp+tlds+1Ch], 'xmsm'
00402CE4 mov     dword ptr [ebp+tlds+20h], 'miik'
00402CEB mov     dword ptr [ebp+tlds+24h], 'ccxc'
00402CF2 mov     dword ptr [ebp+tlds+28h], 'zbvt'
00402CF9 mov     dword ptr [ebp+tlds+2Ch], 'ueem'
00402D00 mov     dword ptr [ebp+tlds+30h], 'ured'
00402D07 mov     dword ptr [ebp+tlds+34h], 'usoc'
00402D0E mov     dword ptr [ebp+tlds+38h], 'zkwp'
00402D15 mov     dword ptr [ebp+tlds+3Ch], 'suxs'
00402D1C mov     dword ptr [ebp+tlds+40h], 'rigu'
00402D23 mov     dword ptr [ebp+tlds+44h], 'agot'
00402D2A mov     dword ptr [ebp+tlds+48h], 'nmoc'
00402D31 mov     dword ptr [ebp+tlds+4Ch], 'rote'
00402D38 mov     dword ptr [ebp+tlds+50h], 'zibg'
00402D3F mov     dword ptr [ebp+tlds+54h], 'pxxx'
00402D46 mov     dword ptr [ebp+tlds+58h], 'ibor'
00402D4D mov     [ebp+tlds+5Ch], 't'
00402D51 call    mod64
00402D56 pop     ebx
00402D57 test    edx, edx
00402D59 jnz     short three_letter_tld
```

```
00402D5B cmp     eax, 36
00402D5E jb      short two_letter_tld
```

```
00402D60
00402D60 three_letter_tld:
00402D60 add     eax, 0FFFFFFDCh
```

```
00402D8C
00402D8C two_letter_tld:
00402D8C add     eax, eax
```




The first call to the routine fetches a hard-coded magic number from a rather complicated linked list and saves it as `magic_number`. For my sample, the hard-coded seed was 9.

Necurs then determines a length between 7 and 21 letters with multiple calls to `pseudo_random`, using the current date, the sequence number and the magic number as seeds. This will lead to a new set of domains every four days:

```
n = pseudo_random(date.year)
n = pseudo_random(n + date.month + 43690)
n = pseudo_random(n + (date.day>>2))
n = pseudo_random(n + sequence_nr)
n = pseudo_random(n + magic_nr)
domain_length = mod64(n, 15) + 7
```

Next, Necurs picks the characters of the second level domain from from ‘a’ to ‘y’ (‘z’ is unreachable like for Ramnit):

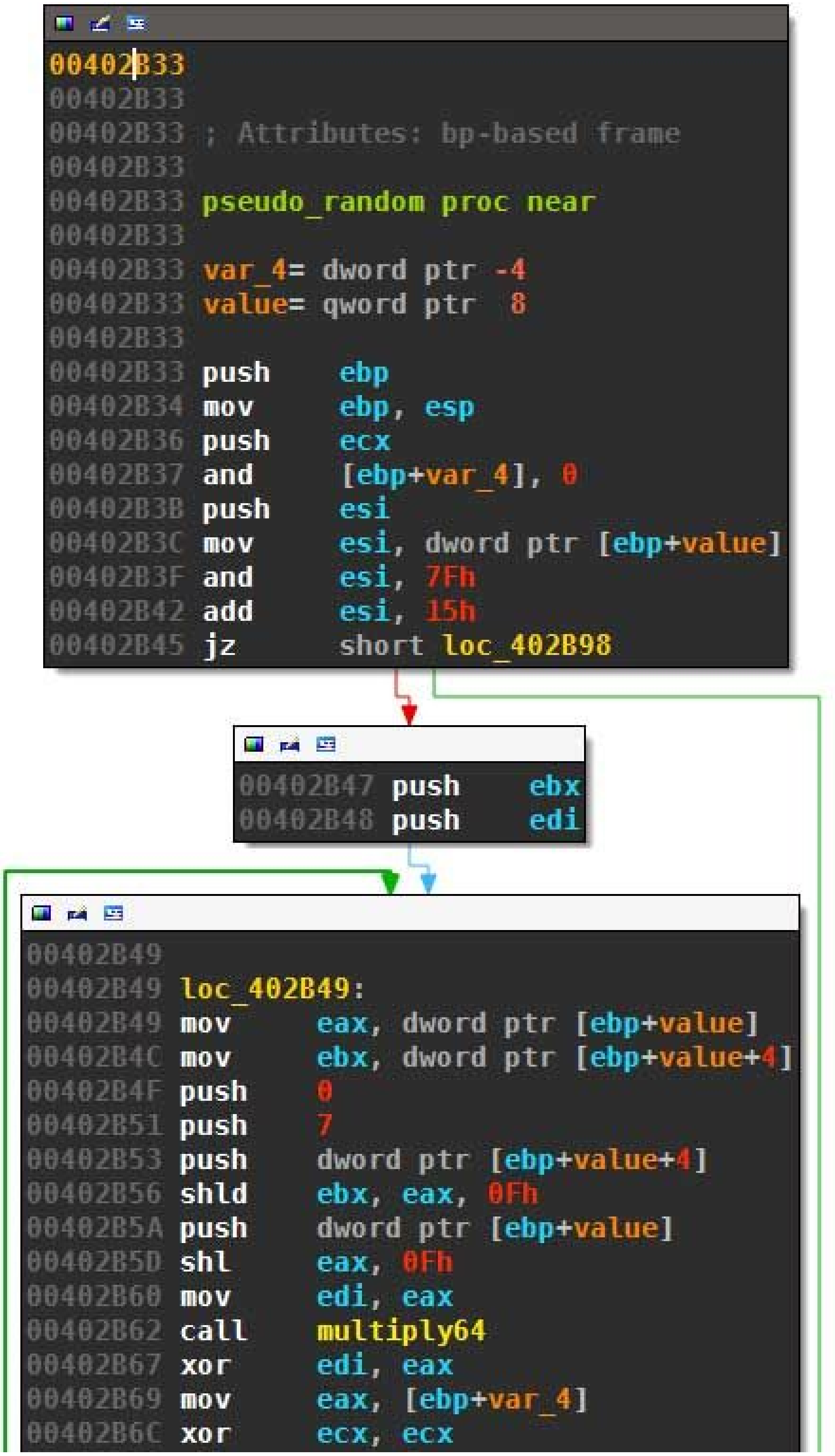
```
domain = ""
for i in range(domain_length):
    n = pseudo_random(n+i)
    ch = mod64(n, 25) + ord('a')
    domain += chr(ch)
    n += 0xABBEDF
    n = pseudo_random(n)
```

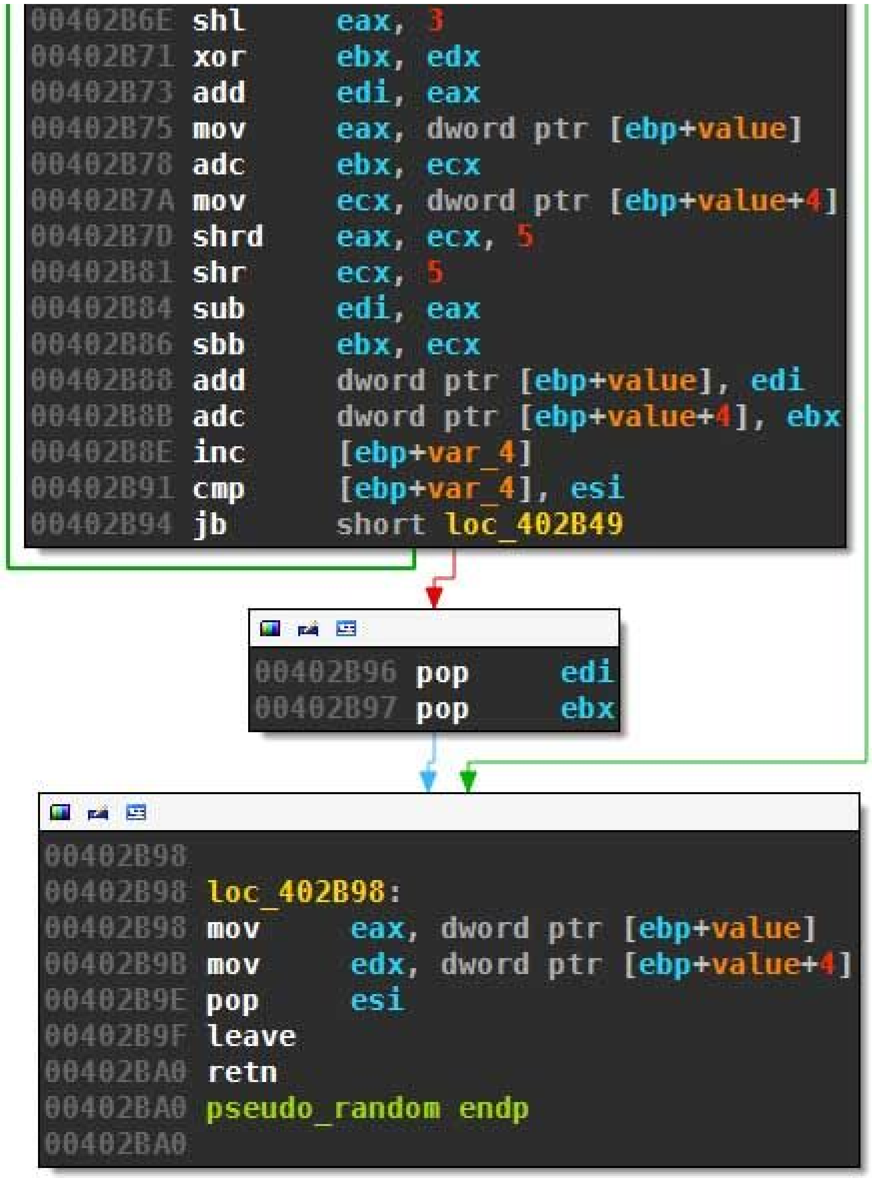
Finally, one of 43 top level domains is chosen to finish the domain name:

```
tlds = ['tj','in','jp','tw','ac','cm','la','mn','so','sh','sc','nu','nf','mu',
'ms','mx','ki','im','cx','cc','tv','bz','me','eu','de','ru','co','su','pw',
'kz','sx','us','ug','ir','to','ga','com','net','org','biz','xxx','pro','bit']

tld = tlds[mod64(n, 43)]
domain += '.' + tld
return domain
```

All picks are randomized with calls to `pseudo_random` which is:






This routine calculates:

```
def pseudo_random(value):
    loops = (value & 0x7F) + 21
    for index in range(loops):
        value += ((value*7) ^ (value << 15)) + 8*index - (value >> 5)
        value &= ((1 << 64) - 1)

    return value
```

## Summary

The third DGA, including the random number generator, boils down to this routine:

```python
import argparse
from datetime import datetime

def generate_necurs_domain(sequence_nr, magic_nr, date):
    def pseudo_random(value):
        loops = (value & 0x7F) + 21
        for index in range(loops):
            value += ((value*7) ^ (value << 15)) + 8*index - (value >> 5)
            value &= ((1 << 64) - 1)

        return value

    def mod64(nr1, nr2):
        return nr1 % nr2

    n = pseudo_random(date.year)
    n = pseudo_random(n + date.month + 43690)
    n = pseudo_random(n + (date.day>>2))
    n = pseudo_random(n + sequence_nr)
    n = pseudo_random(n + magic_nr)
    domain_length = mod64(n, 15) + 7

    domain = ""
    for i in range(domain_length):
        n = pseudo_random(n+i)
        ch = mod64(n, 25) + ord('a')
        domain += chr(ch)
        n += 0xABBEDF
        n = pseudo_random(n)

    tlds = ['tj','in','jp','tw','ac','cm','la','mn','so','sh','sc','nu','nf','mu',
    'ms','mx','ki','im','cx','cc','tv','bz','me','eu','de','ru','co','su','pw',
    'kz','sx','us','ug','ir','to','ga','com','net','org','biz','xxx','pro','bit']

    tld = tlds[mod64(n, 43)]
    domain += '.' + tld
    return domain

if __name__=="__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--date", help="as YYYY-mm-dd")
    args = parser.parse_args()
    date_str = args.date
    if date_str:
        date = datetime.strptime(date_str, "%Y-%m-%d")
    else:
        date = datetime.now()

    for sequence_nr in range(2048):
        print(generate_necurs_domain(sequence_nr, 9, date))
```

The characteristics of this DGA are:

| property | value |
| --- | --- |
| seed | magic number and current date (changing domains every four days) |
| domains per seed | 2048 |
| sequence | randomized (unpredictable, albeit not uniformly random) |
| wait time between domains | none, 16 parallel threads |
| top level domain | 43 different tld, picked randomly |
| second level characters | lower case letters except ‘z’ |
| second level domain length | 7 to 21 letters |

The DGA likely serves as a fallback in case the hard-coded domains fail.

## Archived Comments

***Note:*** *I removed the Disqus integration in an effort to cut down on bloat. The following comments were retrieved with the export functionality of Disqus. If you have comments, please reach out to me by Twitter or email.*

Anon Feb 25, 2015 14:50:36 UTC
Excellent clear analysis. Thanks for the code at the end too, it makes finding evidence of Necurs much easier.