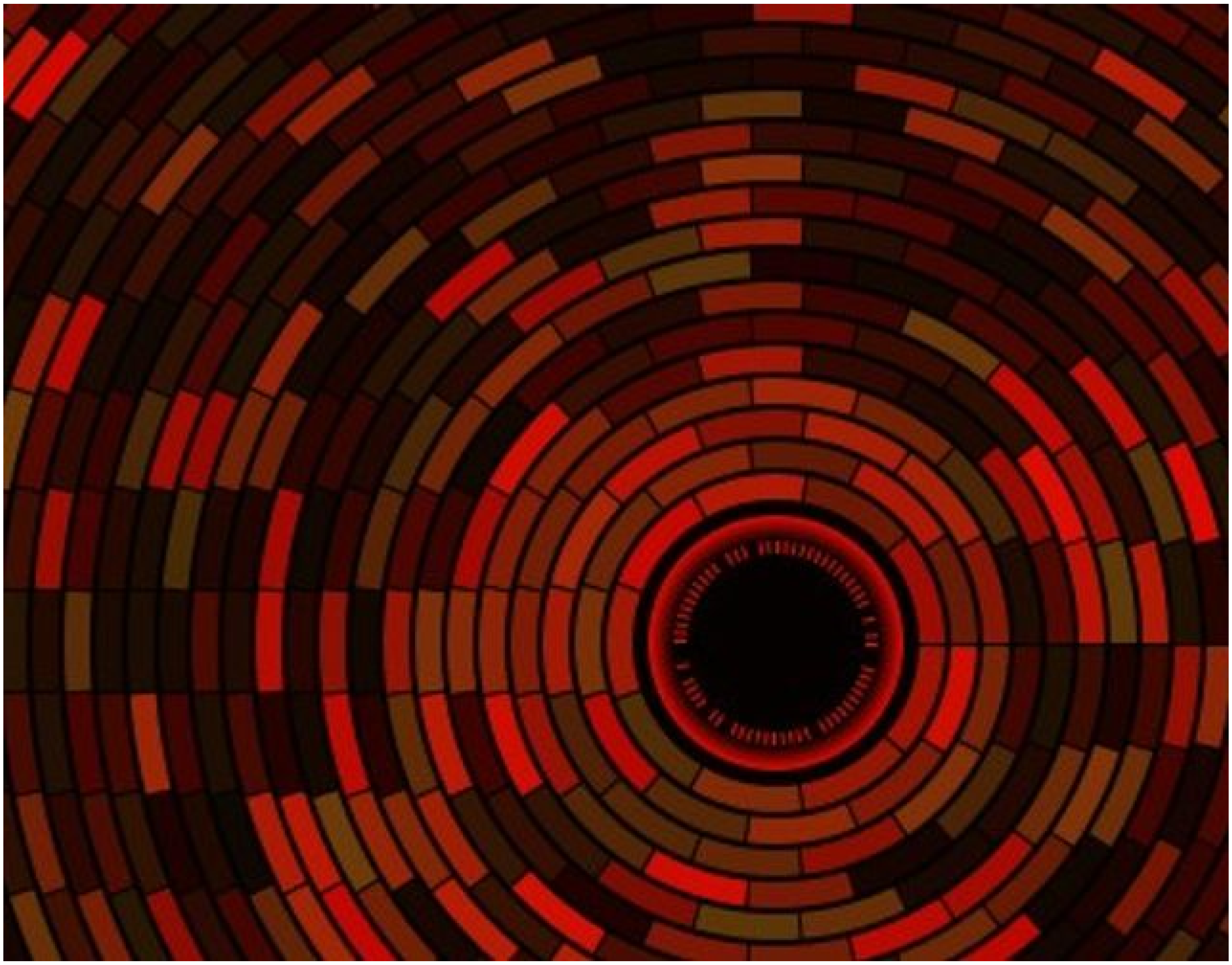
Ranbyus's DGA, Revisited

bin.re/blog/ranbyuss-dga-revisited/



A second version of the Domain Generation Algorithm

> **Edit Dec. 8th, 2015**: I found two additional samples. One of them uses a different tld ordering and an additional operation on the hardcoded seed. I left the original text as is and put the changes in as edits. **Edit Jan. 25, 2016:** found another seed: 0x572473BB
> **Edit Mar. 2, 2016:** found another seed: 0x17794CF1
> **Edit Apr. 7, 2016:** found another seed: 0x7CB7966E

In May I wrote about the Domain Generation Algorithm (DGA) of the banking trojan *Ranbyus*. This week I stumbled on some new Ranbyus samples that use a significant modification of the DGA. For simplicity's sake I call the DGA from the previous post the *May DGA*, and the DGA in this post the *September DGA*. However, I can't tell if the chronology is correct; the DGA in this post might just as well be an earlier or concurrent version of the DGA reported in May.

The domains of the September version at first glance look like the ones from May. The second level domains consist of the letters a-y; the top level domains are the same and they appear in the same order, i.e., *.in* → *.me* → *.cc* → *.su* → *.tw* → *.net* → *.com* → *.pw*. (**Edit**: this newer sample uses the same TLDs in a different order: *.in* → *.net* → *.org* → *.com* → *.me* → *.su* → *.tw* → *.cc* → *.pw*)

For example, these are the first few domains from this report:

- rftkbenepisfitgdj.in
- xiqmvbjmhmhvmgcmi.me
- wxdunehygeonndttn.cc
- sbghxfgtslfpppqiu.su
- upixinckripequtam.tw
- oamxeavybfwlhqhob.net
- jkkugptcygpwxkjkw.com
- cvorpvaacmkfacelm.pw
- vptafodmeuaxopjbs.in
- eycagukbeduvmjnpx.me

The most striking difference to the May version is the increased length of the second level domains: the May version has 14 letters, while the September version uses 17 letters. As it turns out, the September DGA uses a vastly different algorithm to generate the second level domains.

The DGA

Seeding and Samples

Like the May DGA, the new DGA is seeded with the current date. It also produces 40 fresh domains (almost) every day. In addition to the new domains, the DGA will revisit the domains of up 30 days into the past.

Apart from the current date, the DGA is seeded with a hard-coded magic number, which allows for separate sets of domains. So far, the DGArchive collected seven different seeds for the DGA from May. For the new variant, I found seven seeds so far. **Edit Dec. 8th, 2015**: Some samples, e.g., b625b87a9dfdc345d226e913f9f95d77 and d8c247f95b2784419ffc14c8df8efc07, actually reverse the seed before applying it:

```
0009CAB9 | E8 80F8FEFF      | CALL 0008C33E
0009CABE | 8B8C24 8C00000( | MOV ECX,DWORD PTR SS:[ESP+8C]  | hardcoded seed
0009CAC5 | 0FAFC6           | IMUL EAX,ESI
0009CAC8 | F7D1             | NOT ECX
0009CACA | 6A 00            | PUSH 0
0009CACC | 33C1             | XOR EAX,ECX
```

The following table lists the seed after negation so I could leave the reimplementation as is. The *negated* column shows the original seed *before* the NOT-operation.

MD5	seed	negated
eb35f453b87a2f430f53da4dafb2c968	0F0D5BFA	no
b82bfd9f649e08185a4100ab555ee9b9	F2C72B14	no
72a367560582ccd51be6f2284d92c946	0F0D5BFA	no
293cb29f3009503bebb3f9a4d4362537	F2C72B14	no
b7e7c7b77abbc89922806f4bf42fb30e	AE8714BE	no
b625b87a9dfdc345d226e913f9f95d77	CE7F8514	yes (~31807AEB)
ad9f06a74114dfee3e52d63b6b97ce54	F2C72B14	no
821c05d5c949a9b03ba21973ef9072a1	F2C72B14	no
d8c247f95b2784419ffc14c8df8efc07	572473BB	yes (~A8DB8C44)
1d4edada362f6a289b156d94bff26f41	17794CF1	yes (~E886B30E)
c6665471f52a0a7aba50edf8fc9cc886a	C0E32524	yes (~3F1CDADB)
d9393e7afcae648aa742ecaeefd36e07	7CB7966E	yes (~83486991)

The way the current date influences the domains is different. The May DGA uses the year, month and day directly as variables to generate the letters of the second level domain. The September version condenses the date and the hard-coded magic number into a single 32bit value:

$$ X_0 = (\text{year} \cdot \text{month} \cdot \text{day}) \oplus \text{seed} $$

Consequently, all dates that have the same product of year, month and day will generate the same domains. For example, the domains from Januar 24 will be revisited six times the same year: February 12, March 8, April 6, June 4, August 3, and December 12. From a sinkholing perspective, it makes sense to pick a domain from this set.

Python Implementation

The DGA differs in the way the second level characters are picked. While the May version used a custom algorithm to determine the characters, the September edition relies on a pseudo random number generator (PRNG). The PRNG is of the LCG (linear congruential generator) family with common multiplier and increment:

You also find this code, along with a reimplementation of the other Ranbyus version, on my Github).

```
"""
    The DGA of Ranbyus as described here:
        https://bin.re/blog/ranbyuss-dga-revisited/

    Known Seeds are:
        - 0F0D5BFA
        - F2C72B14
        - AE8714BE
        - CE7F8514  (= ~ 31807AEB)
        - 572473BB (= ~ A8DB8C44)
        - 17794CF1 (= ~ E886B30E)
        - C0E32524 (= ~ 3F1CDADB)
"""

import argparse
from datetime import datetime

def to_little_array(val):
    a = 4*[0]
    for i in range(4):
        a[i] = (val & 0xFF)
        val >>= 8
    return a

def pcg_random(r):
    alpha = 0x5851F42D4C957F2D
    inc = 0x14057B7EF767814F

    step1 = alpha*r + inc
    step2 = alpha*step1 + inc
    step3 = alpha*step2 + inc

    tmp = (step3 >> 24) & 0xFFFFFF00 | (step3  & 0xFFFFFFFF) >> 24
    a = (tmp ^ step2) & 0x000FFFFF ^ step2
    b = (step2 >> 32)
    c = (step1 & 0xFFF00000)  | ((step3 >> 32) & 0xFFFFFFFF) >> 12
    d = (step1 >> 32) & 0xFFFFFFFF

    data = 32*[None]
    data[0:4] = to_little_array(a)
    data[4:8] = to_little_array(b)
    data[8:12] = to_little_array(c)
    data[12:16] = to_little_array(d)
    return step3 & 0xFFFFFFFFFFFFFFFF, data

def dga(year, month, day, seed):
    x = (day*month*year) ^ seed
    tld_index = day
    for _ in range(40):
        random = 32*[None]
        x, random[0:16] = pcg_random(x)
        x, random[16:32] = pcg_random(x)
```

```
        domain = ""
        for i in range(17):
            domain += chr(random[i] % 25 + ord('a'))
        if seed == 0xCE7F8514:
            tlds =  ["in", "net", "org", "com", "me", "su", "tw", "cc", "pw"]
        else:
            tlds =  ["in", "me", "cc", "su", "tw", "net", "com", "pw", "org"]
        domain += '.' + tlds[tld_index % (len(tlds) - 1)]
        tld_index += 1
        yield domain

if __name__=="__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--date", help="date for which to generate domains")
    parser.add_argument("-s", "--seed", help="seed as hex string",
default="0F0D5BFA")
    args = parser.parse_args()
    if args.date:
        d = datetime.strptime(args.date, "%Y-%m-%d")
    else:
        d = datetime.now()
    for domain in dga(d.year, d.month, d.day, int(args.seed, 16)):
        print(domain)
```

Please note that the above Python script only generates the 40 domains of the current day. Like the May version, Ranbyus can also revisit older domains up to 30 days into the past. So to get the full set of domains for any given day, you need to run the script for 31 different days.

Properties

Almost all characteristics of the Ranbyus September DGA are the same as for the May version. The only difference is the increased length of the second level domains:

property	value
seed	magic number and current date
granularity	1 day, with a 31 day sliding window
domains per seed and day	40
domains per sliding window	1240
sequence	sequential
wait time between domains	500 ms
top level domains	.in, .me, .cc, .su, .tw, .net, .com, .pw

property	value
second level characters	lower case letters except ‘z’
second level domain length	**17 letters** (May version: 14 letters)

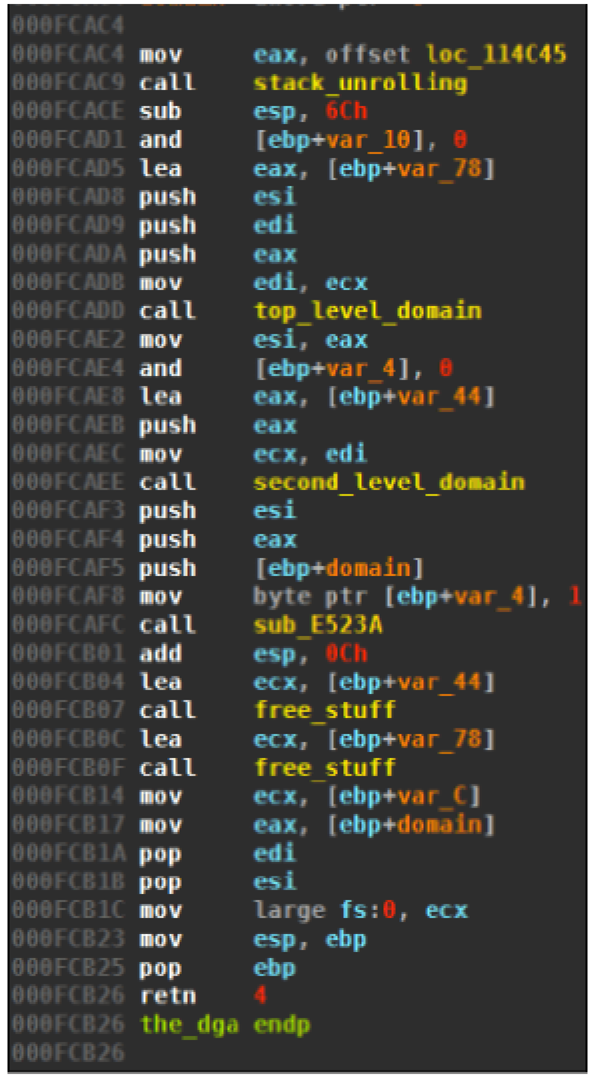
Appendix - Reversing the DGA

Similarities with May version

The new samples share most of the DGA code with the May version. The following graph views show the callback loop from May (left) and September (right):

The basic structure of the DGA itself is also equal:

```
0009C848 the_dga proc near
0009C848 mov     eax, offset loc_B4696
0009C84D call    stack_unrolling
0009C852 sub     esp, 6Ch
0009C855 and     dword ptr [ebp-10h], 0
0009C859 lea     eax, [ebp-78h]
0009C85C push    esi
0009C85D push    edi
0009C85E push    eax
0009C85F mov     edi, ecx
0009C861 call    top_level_domain
0009C866 mov     esi, eax
0009C868 and     dword ptr [ebp-4], 0
0009C86C lea     eax, [ebp-44h]
0009C86F push    eax
0009C870 mov     ecx, edi
0009C872 call    second_level_domain
0009C877 push    esi
0009C878 push    eax
0009C879 push    dword ptr [ebp+8]
0009C87C mov     byte ptr [ebp-4], 1
0009C880 call    sub_852F4
0009C885 add     esp, 0Ch
0009C888 lea     ecx, [ebp-44h]
0009C88B call    free_stuff
0009C890 lea     ecx, [ebp-78h]
0009C893 call    free_stuff
0009C898 mov     ecx, [ebp-0Ch]
0009C89B mov     eax, [ebp+8]
0009C89E pop     edi
0009C89F pop     esi
0009C8A0 mov     large fs:0, ecx
0009C8A7 mov     esp, ebp
0009C8A9 pop     ebp
0009C8AA retn    4
0009C8AA the_dga endp
0009C8AA
```



```
000FCAC4
000FCAC4 mov     eax, offset loc_114C45
000FCAC9 call    stack_unrolling
000FCACE sub     esp, 6Ch
000FCAD1 and     [ebp+var_10], 0
000FCAD5 lea     eax, [ebp+var_78]
000FCAD8 push    esi
000FCAD9 push    edi
000FCADA push    eax
000FCADB mov     edi, ecx
000FCADD call    top_level_domain
000FCAE2 mov     esi, eax
000FCAE4 and     [ebp+var_4], 0
000FCAE8 lea     eax, [ebp+var_44]
000FCAEB push    eax
000FCAEC mov     ecx, edi
000FCAEE call    second_level_domain
000FCAF3 push    esi
000FCAF4 push    eax
000FCAF5 push    [ebp+domain]
000FCAF8 mov     byte ptr [ebp+var_4], 1
000FCAFC call    sub_E523A
000FCB01 add     esp, 0Ch
000FCB04 lea     ecx, [ebp+var_44]
000FCB07 call    free_stuff
000FCB0C lea     ecx, [ebp+var_78]
000FCB0F call    free_stuff
000FCB14 mov     ecx, [ebp+var_C]
000FCB17 mov     eax, [ebp+domain]
000FCB1A pop     edi
000FCB1B pop     esi
000FCB1C mov     large fs:0, ecx
000FCB23 mov     esp, ebp
000FCB25 pop     ebp
000FCB26 retn    4
000FCB26 the_dga endp
000FCB26
```

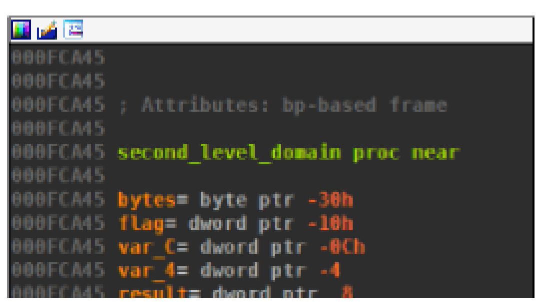
Most other DGA-related functions stayed the same too, in particular:

- The routine to determine the top level domain *top_level_domain*, i.e., the domains will have the same top level domains in the same order as the DGA from May.
- The routines to determine and handle the current date.
- The data structures to configure the DGA.

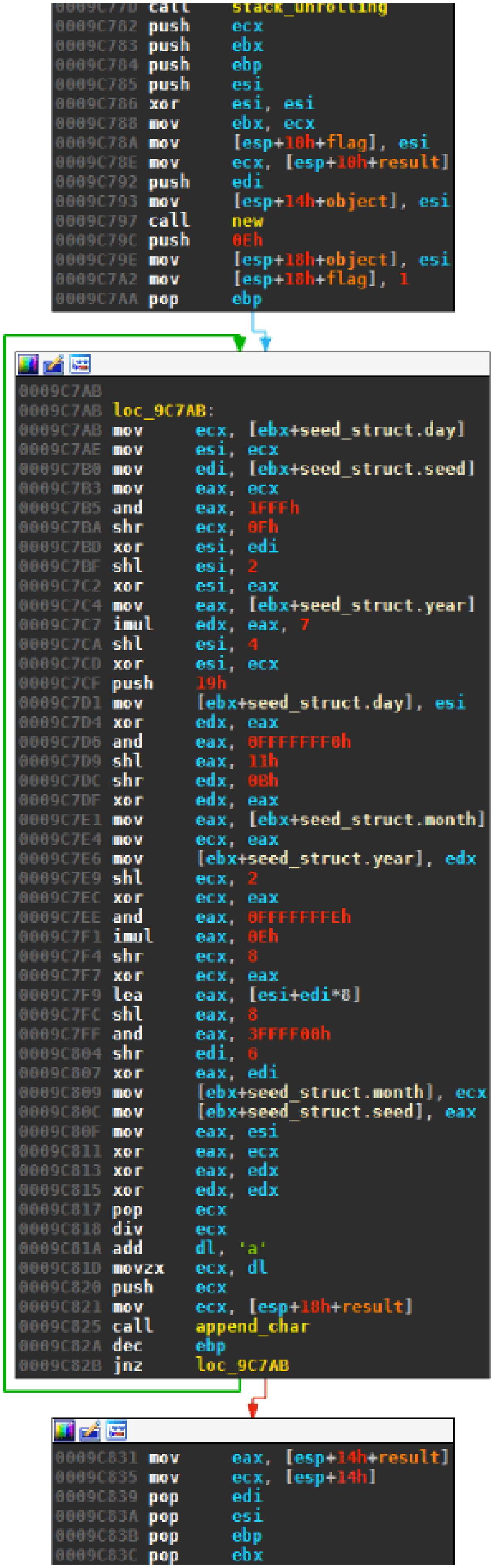
Differences to May version

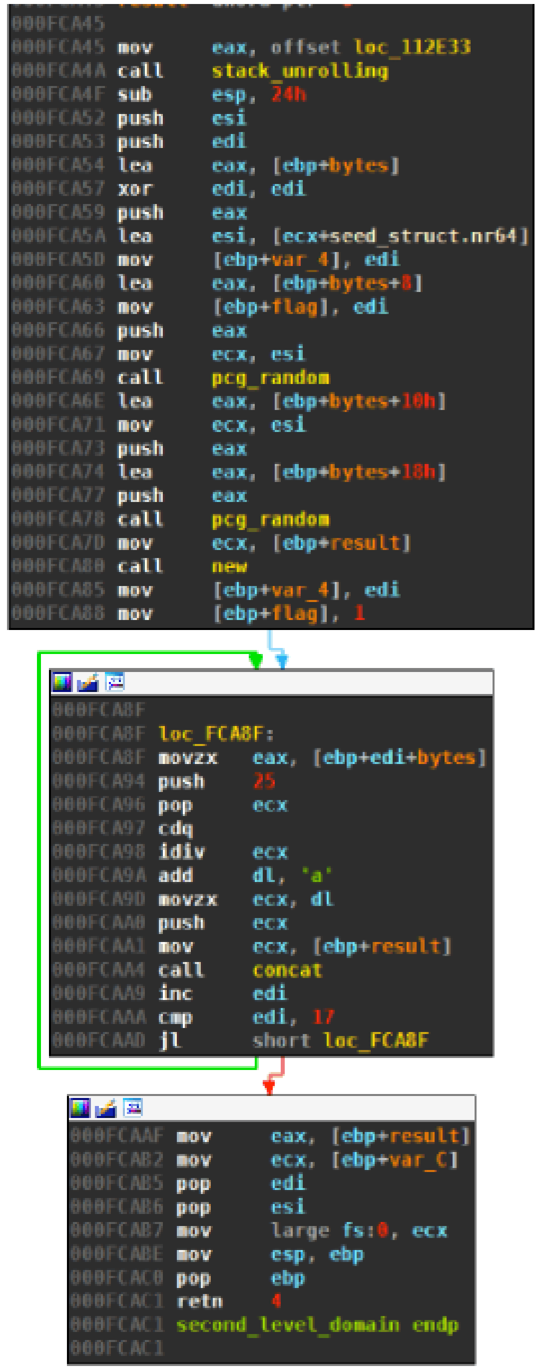
The main difference between the two DGAs is the routine to generate the second level domains:

```
0009C778
0009C778
0009C778
0009C778 second_level_domain proc near
0009C778
0009C778 flag= dword ptr -4
0009C778 object= dword ptr  8
0009C778 result= dword ptr  14h
0009C778
0009C778 mov     eax, offset loc_B4976
```



```
000FCA45
000FCA45
000FCA45 ; Attributes: bp-based frame
000FCA45
000FCA45 second_level_domain proc near
000FCA45
000FCA45 bytes= byte ptr -30h
000FCA45 flag= dword ptr -10h
000FCA45 var_C= dword ptr -0Ch
000FCA45 var_4= dword ptr -4
```


0009C782 push ecx
0009C783 push ebx
0009C784 push ebp
0009C785 push esi
0009C786 xor esi, esi
0009C788 mov ebx, ecx
0009C78A mov [esp+10h+flag], esi
0009C78E mov ecx, [esp+10h+result]
0009C792 push edi
0009C793 mov [esp+14h+object], esi
0009C797 call new
0009C79C push 0Eh
0009C79E mov [esp+18h+object], esi
0009C7A2 mov [esp+18h+flag], 1
0009C7AA pop ebp
0009C7AB
0009C7AB loc_9C7AB:
0009C7AB mov ecx, [ebx+seed_struct.day]
0009C7AE mov esi, ecx
0009C7B0 mov edi, [ebx+seed_struct.seed]
0009C7B3 mov eax, ecx
0009C7B5 and eax, 1FFFh
0009C7BA shr ecx, 0Fh
0009C7BD xor esi, edi
0009C7BF shl esi, 2
0009C7C2 xor esi, eax
0009C7C4 mov eax, [ebx+seed_struct.year]
0009C7C7 imul edx, eax, 7
0009C7CA shl esi, 4
0009C7CD xor esi, ecx
0009C7CF push 19h
0009C7D1 mov [ebx+seed_struct.day], esi
0009C7D4 xor edx, eax
0009C7D6 and eax, 0FFFFFFF0h
0009C7D9 shl eax, 11h
0009C7DC shr edx, 0Bh
0009C7DF xor edx, eax
0009C7E1 mov eax, [ebx+seed_struct.month]
0009C7E4 mov ecx, eax
0009C7E6 mov [ebx+seed_struct.year], edx
0009C7E9 shl ecx, 2
0009C7EC xor ecx, eax
0009C7EE and eax, 0FFFFFFFEh
0009C7F1 imul eax, 0Eh
0009C7F4 shr ecx, 8
0009C7F7 xor ecx, eax
0009C7F9 lea eax, [esi+edi*8]
0009C7FC shl eax, 8
0009C7FF and eax, 3FFFF00h
0009C804 shr edi, 6
0009C807 xor eax, edi
0009C809 mov [ebx+seed_struct.month], ecx
0009C80C mov [ebx+seed_struct.seed], eax
0009C80F mov eax, esi
0009C811 xor eax, ecx
0009C813 xor eax, edx
0009C815 xor edx, edx
0009C817 pop ecx
0009C818 div ecx
0009C81A add dl, 'a'
0009C81D movzx ecx, dl
0009C820 push ecx
0009C821 mov ecx, [esp+18h+result]
0009C825 call append_char
0009C82A dec ebp
0009C82B jnz loc_9C7AB
0009C831 mov eax, [esp+14h+result]
0009C835 mov ecx, [esp+14h]
0009C839 pop edi
0009C83A pop esi
0009C83B pop ebp
0009C83C pop ebx


000FCA45
000FCA45 mov eax, offset loc_112E33
000FCA4A call stack_unrolling
000FCA4F sub esp, 24h
000FCA52 push esi
000FCA53 push edi
000FCA54 lea eax, [ebp+bytes]
000FCA57 xor edi, edi
000FCA59 push eax
000FCA5A lea esi, [ecx+seed_struct.nr64]
000FCA5D mov [ebp+var_4], edi
000FCA60 lea eax, [ebp+bytes+8]
000FCA63 mov [ebp+flag], edi
000FCA66 push eax
000FCA67 mov ecx, esi
000FCA69 call pcg_random
000FCA6E lea eax, [ebp+bytes+10h]
000FCA71 mov ecx, esi
000FCA73 push eax
000FCA74 lea eax, [ebp+bytes+18h]
000FCA77 push eax
000FCA78 call pcg_random
000FCA7D mov ecx, [ebp+result]
000FCA80 call new
000FCA85 mov [ebp+var_4], edi
000FCA88 mov [ebp+flag], 1
000FCA8F
000FCA8F loc_FCA8F:
000FCA8F movzx eax, [ebp+edi+bytes]
000FCA94 push 25
000FCA96 pop ecx
000FCA97 cdq
000FCA98 idiv ecx
000FCA9A add dl, 'a'
000FCA9D movzx ecx, dl
000FCAA0 push ecx
000FCAA1 mov ecx, [ebp+result]
000FCAA4 call concat
000FCAA9 inc edi
000FCAAA cmp edi, 17
000FCAAD jl short loc_FCA8F
000FCAAF mov eax, [ebp+result]
000FCAB2 mov ecx, [ebp+var_C]
000FCAB5 pop edi
000FCAB6 pop esi
000FCAB7 mov large fs:0, ecx
000FCABE mov esp, ebp
000FCAC0 pop ebp
000FCAC1 retn 4
000FCAC1 second_level_domain endp
000FCAC1

```
0009C83D mov     large fs:0, ecx
0009C844 leave
0009C845 retn    4
0009C845 second_level_domain endp
0009C845
```

The May DGA (on the left) uses a custom algorithm inside the loop body to produce a pseudo random number. The September version on the right first generates 32 bytes of random data using the *pcg_random* routine, and then simply accesses this data inside the loop body. Both version take the resulting pseudo random number modulo 25 to get letters from *a* to *y*.

The pseudo random number generator is based on 64bit numbers, which make the routine a little hard to read:

```
000F361A ; _QWORD *__thiscall pcg_random(_QWORD *this, int res1, int res2)
000F361A pcg_random proc near
000F361A
000F361A low_second= dword ptr -10h
000F361A ii= dword ptr -0Ch
000F361A jj= dword ptr -8
000F361A this= dword ptr -4
000F361A res1= dword ptr  4
000F361A res2= dword ptr  8
000F361A
000F361A sub     esp, 10h
000F361D push    ebx
000F361E push    ebp
000F361F push    esi
000F3620 push    edi
000F3621 mov     eax, ecx
000F3623 mov     edi, 4C957F2Dh
000F3628 push    5851F42Dh
000F362D push    edi
000F362E mov     [esp+28h+this], eax
000F3632 push    dword ptr [eax+4]
000F3635 push    dword ptr [eax]
000F3637 call    multiply
000F363C mov     ebp, eax
000F363E mov     ebx, 0F767814Fh
000F3643 push    5851F42Dh
000F3648 add     ebp, ebx
000F364A mov     eax, edx
000F364C mov     esi, 14057B7Eh
000F3651 adc     eax, esi
000F3653 push    edi
000F3654 push    eax
000F3655 push    ebp
000F3656 mov     [esp+30h+low_second], eax
000F365A call    multiply
000F365F add     eax, ebx
000F3661 mov     ecx, edx
000F3663 push    5851F42Dh
000F3668 adc     ecx, esi
000F366A mov     [esp+24h+ii], eax
000F366E push    edi
000F366F push    ecx
000F3670 push    eax
000F3671 mov     [esp+30h+jj], ecx
000F3675 call    multiply
000F367A mov     edi, eax
```

```
000F367C mov     ebx, edx
000F367E mov     eax, [esp+20h+res1]
000F3682 add     edi, 0F767814Fh
000F3688 mov     edx, edi
000F368A adc     ebx, esi
000F368C xor     ecx, ecx
000F368E or      ecx, [esp+20h+low_second]
000F3692 mov     esi, ebx
000F3694 mov     [eax+4], ecx
000F3697 and     ebp, 0FFF00000h
000F369D shr     esi, 0Ch
000F36A0 xor     ecx, ecx
000F36A2 or      esi, ebp
000F36A4 mov     [eax], esi
000F36A6 mov     eax, ebx
000F36A8 shrd    edx, eax, 18h
000F36AC mov     eax, [esp+20h+res2]
000F36B0 xor     edx, [esp+20h+ii]
000F36B4 and     edx, 0FFFFFh
000F36BA xor     edx, [esp+20h+ii]
000F36BE xor     ecx, [esp+20h+jj]
000F36C2 mov     [eax], edx
000F36C4 mov     [eax+4], ecx
000F36C7 mov     eax, [esp+20h+this]
000F36CB mov     [eax], edi
000F36CD pop     edi
000F36CE pop     esi
000F36CF pop     ebp
000F36D0 mov     [eax+4], ebx
000F36D3 pop     ebx
000F36D4 add     esp, 10h
000F36D7 retn    8
000F36D7 pcg_random endp
```

At the core of the above routine is the following linear congruential generator:

$$ X_{n+1} = (6364136223846793005\cdot X_n + 1442695040888963407) \text{ mod } 2^{64} $$

The initial value X_0 is set to the product of *year*, *month*, and *day*, XORed with the hardcoded seed:

```
000FC832 mov     ecx, [esp+70h+datetime]
000FC839 push    esi
000FC83A call    get_day
000FC83F mov     ecx, [esp+74h+datetime]
000FC846 mov     esi, eax
000FC848 call    get_month
000FC84D mov     ecx, [esp+74h+datetime]
000FC854 imul    esi, eax
000FC857 call    get_year
000FC85C imul    eax, esi
000FC85F lea     ecx, [ebx+seed_struct.X]
000FC862 push    0
000FC864 xor     eax, [esp+78h+seed]
000FC86B push    eax
000FC86C call    copy64_0
```