The DGA of Banjori

Archived: 2026-04-05 21:08:08 UTC

This post analyses the domain generation algorithm (DGA) of the banking trojan *Banjori*, also known as *MultiBanker 2* or *BankPatch*/*BackPatcher*. The DGA was active mostly between April and November of 2013 (at least thats when I found most seeds). Two blog posts on kleissner.org [1] [2] document many interesting properties of the malware, including some aspects of the DGA.

Although the malware family and its DGA might no longer be relevant, I still liked the malware for two reasons:

1. The malware's disassembly is very clean and simple to reverse, many parts were probably written in assembly.
2. The DGA uses a multivariate recurrence relation with some interesting properties.

I reversed this fairly recent sample from malwr.com, shared by the uploader for anyone to download.

The DGA

The callback loop for the examined Banjori sample in IDA's graph view looks as follows:

The malware tries to post various data with subroutine `connects` (offset 2B113). The target domain of the POST is stored in a global variable which does not appear in the above graph view. The first such domain is hard-coded. In case the domain name can be resolved, the malware POSTs data to the CnC-server and reads its response (not shown). Valid responses end with `"chek"`, see offset 2B129. Only if Banjori receives such a response from the CnC server will it consider the callback a success and return. In all other cases, the malware enters a `check_connectivity_loop` (offset 2B14B):

```
 000282CB check_connectivity_loop proc near
000282CB
000282CB                push    offset aWww_google_de ; "www.google.de"
000282D0                call    ds:gethostbyname ; ws2_32.gethostbyname
000282D6                test    eax, eax
000282D8                jnz     short locret_282E7
000282DA                push    5000
000282DF                call    ds:Sleep        ; kernel32.Sleep
000282E5                jmp     short check_connectivity_loop
000282E7 ; ---------------------------------------------------------------
000282E7
000282E7 locret_282E7:
000282E7                retn
000282E7 check_connectivity_loop endp
```

This loop tries to resolve `www.google.de` every 5 seconds until it succeeds — indicating the host has internet connectivity.

Every failed POST is retried a second time for the same domain (see offsets 2B0EE and 2B146). After the second failed attempt, Banjori calls `generate_and_save_next_domain`. This routine generates a new domain based on the previous:

```
 000281BD push    ds:pDomain
000281C3 call    next_domain
```

The routine `next_domain` is:

```
 00028242 next_domain     proc near
00028242
00028242
00028242 prevdomain      = dword ptr  8
00028242
00028242                 push    ebp
00028243                 mov     ebp, esp
00028245                 push    edi
00028246                 mov     edi, [ebp+prevdomain]
00028249                 cmp     dword ptr [edi], 'ptth'
0002824F                 jnz     short loc_28254
00028251                 add     edi, 7
00028254
00028254 loc_28254:
00028254                 movzx   eax, byte ptr [edi]
00028257                 movzx   ecx, byte ptr [edi+3]
0002825B                 add     eax, ecx
0002825D                 push    eax
0002825E                 call    map_to_lowercase_letter
00028263                 mov     [edi], al
00028265                 movzx   eax, byte ptr [edi+1]
00028269                 movzx   ecx, byte ptr [edi]
0002826C                 movzx   edx, byte ptr [edi+1]
00028270                 add     eax, ecx
00028272                 add     eax, edx
00028274                 push    eax
00028275                 call    map_to_lowercase_letter
0002827A                 mov     [edi+1], al
0002827D                 movzx   eax, byte ptr [edi+2]
00028281                 movzx   ecx, byte ptr [edi]
00028284                 add     eax, ecx
00028286                 dec     eax
00028287                 push    eax
00028288                 call    map_to_lowercase_letter
0002828D                 mov     [edi+2], al
00028290                 movzx   eax, byte ptr [edi+3]
00028294                 movzx   ecx, byte ptr [edi+1]
00028298                 movzx   edx, byte ptr [edi+2]
0002829C                 add     eax, ecx
0002829E                 add     eax, edx
000282A0                 push    eax
000282A1                 call    map_to_lowercase_letter
000282A6                 mov     [edi+3], al
000282A9                 pop     edi
000282AA                 leave
```

```
000282AB                retn    4
000282AB next_domain    endp
```

The routine `map_to_lowercase_letter` is:

```
 000282AE map_to_lowercase_letter proc near
000282AE
000282AE
000282AE charsum         = dword ptr  8
000282AE
000282AE                push    ebp
000282AF                mov     ebp, esp
000282B1                mov     eax, [ebp+charsum]
000282B4                sub     eax, 'a'
000282B7                mov     ecx, 26
000282BC
000282BC loc_282BC:
000282BC                cmp     eax, ecx
000282BE                jb      short loc_282C4
000282C0                sub     eax, ecx
000282C2                jmp     short loc_282BC
000282C4 ; ---------------------------------------------------------
000282C4
000282C4 loc_282C4:
000282C4                add     eax, 'a'
000282C7                leave
000282C8                retn    4
000282C8 map_to_lowercase_letter endp
```

The following Python script implements these two routines to calculate 10 domains of the DGA:

```
 def map_to_lowercase_letter(s):
    return ord('a') + ((s - ord('a')) % 26)

def next_domain(domain):
    dl = [ord(x) for x in list(domain)]
    dl[0] = map_to_lowercase_letter(dl[0] + dl[3])
    dl[1] = map_to_lowercase_letter(dl[0] + 2*dl[1])
    dl[2] = map_to_lowercase_letter(dl[0] + dl[2] - 1)
    dl[3] = map_to_lowercase_letter(dl[1] + dl[2] + dl[3])
    return ''.join([chr(x) for x in dl])

seed = 'earnestnessbiophysicalohax.com' # 15372 equal to 0 (seed = 0)
domain = seed
for i in range(10):
```

```
    print(domain)
    domain = next_domain(domain)
```

The DGA only changes the first four letters of the seed domain, leaving the rest of the domain as is:

```
 $ python dga.py
earnestnessbiophysicalohax.com
kwtoestnessbiophysicalohax.com
rvcxestnessbiophysicalohax.com
hjbtestnessbiophysicalohax.com
txmoestnessbiophysicalohax.com
agekestnessbiophysicalohax.com
dbzwestnessbiophysicalohax.com
sgjxestnessbiophysicalohax.com
igjyestnessbiophysicalohax.com
zxahestnessbiophysicalohax.com
```

The current domain is saved in registry. This allows Banjori to pick up where it left off after a reboot:

```
 000281C8 push    ds:pDomain
000281CE call    ds:lstrlenA     ; kernel32.lstrlenA
000281D4 inc     eax
000281D5 push    eax
000281D6 push    ds:pDomain
000281DC push    1
000281DE push    ebx
000281DF push    offset subkey   ; "tst"
000281E4 push    ds:hkey         ; HKCU
000281EA call    ds:RegSetValueExA ; ADVAPI32.RegSetValueExA
```

Some Properties of the Recurrence Relation

Let $d = (d_0, d_1, d_2, d_3)$ denote the first four letters of the domains subject to change, without the offset `"a" = 97`. For example, `abcz` is represented by (0,1,2,25). Also let $d^{(i)}$ denote the first four letters of the i th domain. Then `next_domain` implements the following 4-indexed recurrence relation of order 1:

$$
\begin{aligned}
d_0^{(i+1)} &= d_0^{(i)} + d_3^{(i)} \mod 26 \\
d_1^{(i+1)} &= d_0^{(i)} + 2 \cdot d_1^{(i)} + d_3^{(i)} \mod 26 \\
d_2^{(i+1)} &= d_0^{(i)} + d_2^{(i)} + d_3^{(i)} - 1 \mod 26 \\
d_3^{(i+1)} &= 2 \cdot d_1^{(j)} + 2 \cdot d_1^{(j)} + d_2^{(j)} + 3 \cdot d_3^{(j)} - 1 \mod 26
\end{aligned}
$$

Since d is in $Z_{26} \times Z_{26} \times Z_{26} \times Z_{26}$ the sequence will inevitably repeat itself. Once the sequence reaches a previously visited word, all domains in between form a cycle. I call all sequences of four letter words that not part of a cycle *tail*. The recurrence relation used for this DGA has an interesting property: d is part of a tail if and only if $d_0 + d_1 \equiv 1 \mod 2$. Furthermore, because

$$
\begin{aligned}
d_0^{(i+1)} + d_1^{(i+1)} &= d_0^{(i)} + d_3^{(i)} + d_0^{(i)} + 2 \cdot d_1^{(i)} + d_3^{(i)} \\
&= 2 \cdot d_0^{(i)} + 2 \cdot d_1^{(i)} + 2 \cdot d_3^{(i)} \\
&\equiv 0 \mod 2
\end{aligned}
$$

all tails have length 1. It can also been shown that every non-tail word has one tail precursor. From these properties, and an exhaustive enumeration of all words, one can deduce the following properties:

- If a domain starts with a four letter tail word, it must be the seed domain.
- All four letter tail words have no precursor.
- All four letter words of a cycle have two precursor: one tail word and one word that is part of the cycle.
- Half of the four letter words are tails, the other half part of a cycle
- There are 32 different cycles with different lengths (see below).

The following graphic shows part of a 15372 word cycle with its tail words:

The length of the 32 cycles varies:

- 13 cycles have length 15372, covering 87% of all four letter words.
- 13 cycles have length 2196, covering and additional 12% of words. 99.95% of all four letter words end up in either 2196 or 15372 length cycles.
- 2 cycles have length 42.
- 1 cycles has length 7
- 2 cycles have length 6
- 1 four letter – the word “igih” – is a fixed point; the recurrence relation will not change this word.

The analysis of the recurrence relation show two interesting facts:

- The DGA will not work for seed domains starting with “igih”. Of course the probability of such a choice is virtually zero.
- Half of the seed domains will be tails, meaning they are never revisited once the initial connection fails. Registering one of the cycle domains (e.g., the domain right after the seed domain) might therefore be a better option as these domains will be revisited eventually - although most cycles lengths are awfully long.

Seeds in the Wild

As mentioned above, Banjori is seeded with hardcoded domains. One can find Banjori samples by searching the various online sandboxes for the two mutexe `UpdateJbr32` and `JbrDelMutex`. The following tables lists the samples that I found, including the seed domain. The table also lists whether or not the seed domain is a tail domain (meaning it will never be revisited), and the length of the cycle (how many different domains the seed will produce).

seed	date	md5	tail	cycle length
antisemitismgavenuteq.com	2013-04-24	538da019729597b176e5495aa5412e83	yes	15372
bandepictom.com	2013-05-05	5592456E82F60D2222C9F2BCE5444DE5	yes	15372
buckbyplaywobb.com	2013-05-13	f9d02df23531cff89b0d054b30f98421	yes	15372
telemachuslazaroqok.com	2013-05-14	bc69a956b147c99f6d316f8cea435915	yes	15372
texanfoulilp.com	2013-07-01	36a9c28031d07b82973f7c9eec3b995c	yes	15372
clearasildeafeninguvuc.com	2013-07-05	1e081e503668347c81bbba7642bef609	yes	15372
marisagabardinedazyx.com	2013-07-12	c2c980ea81547c4b8de34adf829ccc26	no	15372
pickfordlinnetavox.com	2013-07-12	4e76a7ba69d1b6891db95add7b29225e	yes	15372
snapplefrostbitecycz.com	2013-08-06	abb80f23028c49d753e7c93a801444d8	yes	2196
filtererwyatanb.com	2013-08-12	eff48dae5e91845c2414f0a4f91a1518	yes	15372
antwancorml.com	2013-08-26	5dda3983ac7cebd3190942ee47a13e50	yes	15372
stravinskycattederifg.com	2013-08-30	eaeb5a9d8d955831c443d4a6f9e179fd	yes	15372
forepartbulkyf.com	2013-09-01	080b3f46356493aeb7ec38e30acbe4f5	yes	15372
fundamentalistfanchonut.com	2013-09-15	40827866594cc26f12bda252939141f6	yes	15372
criterionirkutskagl.com	2013-10-28	8e1d326b687fc4aacc6914e16652c288	yes	2196

seed	date	md5	tail	cycle length
criminalcentricem.com	2013-11-04	a03971bff15ec6782ae25182f4533b92	yes	15372
babysatformalisticirekb.com	2014-11-08	b9fb8ae5e3985980175e74cf5deaa6fb	yes	15372
earnestnessbiophysicalohax.com	2015-02-05	f555132e0b7984318b965f984785d360	no	15372

The blog posts on kleissner.org [1] [2] list additional seeds.

DGA Characteristics

The following table summarizes the properties of the DGA:

property	value
seed	domain, time independent
domains per seed	at most 15373, at least 2196 (in some very rare cases less, see the analysis on the recurrence relation)
tested domains	all
sequence	one after another, potentially restarting with first or second domain (see discussion on tail vs body domains). Last checked domain state saved in registry.
wait time between domains	none as long as DNS query for `www.google.de` succeeds
top and second level domain	same as seed, `.com` for all observed seeds
second level characters	first four characters are letters, remaining character could be anything
second level domain length	same as seed domain

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.*
