The DGA of Shiotob

Archived: 2026-04-06 15:47:51 UTC

The Shiotob malware family steals user credentials - most notably information related to banking. The malware injects itself into legitimate processes, for instances `explorer.exe`. To contact its C&C servers Shiotob uses a Domain Generation Algorithm (DGA), for example:

```
 02:31:53  HTTP connection, method: GET, URL: http://www.google.com/
02:31:53  HTTPS connection, method: POST, URL: https://wtipubctwiekhir.net/gnu/
02:31:53  HTTPS connection, method: POST, URL: https://rwmu35avqo12tqc.com/gnu/
02:31:58  HTTPS connection, method: POST, URL: https://rskb5bsfhm2fk5h.net/gnu/
02:32:03  HTTPS connection, method: POST, URL: https://rbp9pprrxgflut9.com/gnu/
02:32:08  HTTPS connection, method: POST, URL: https://zzxeyzgy45yy2a.net/gnu/
02:32:13  HTTPS connection, method: POST, URL: https://e3oa4wglvd21xa.com/gnu/
02:36:12  HTTP connection, method: GET, URL: http://www.gstatic.com/generate_204
02:36:12  HTTP connection, method: GET, URL: http://www.gstatic.com/generate_204
02:37:18  HTTPS connection, method: POST, URL: https://mqmq1hvmtxzjv.net/gnu/
02:37:23  HTTPS connection, method: POST, URL: https://pd4o4wu24vimn.com/gnu/
02:37:28  HTTPS connection, method: POST, URL: https://tlmrzvpbpsqsb.net/gnu/
02:37:33  HTTPS connection, method: POST, URL: https://pbmnz59uzndpo.com/gnu/
02:37:38  HTTPS connection, method: POST, URL: https://x2lxslqz3wztw.net/gnu/
```

Shiotob uses some anti-sandbox techniques, e.g., a 10 minute wait time before contacting the first domain. For this reason, most online sandbox reports don’t list the DGA domains. In this blog post I show how the domain names are generated.

Seed

Shiotob is **seeded with a hardcoded url**. The url is stored in a larger structure with other data. Apart from the seed url, this struct also contains a time stamp relevant to the DGA. I’m not entirely sure when the timestampis taken, but it probably refers to when the malware is first run. The two values are at offset 9 and 0x164 respectively:

```
 global_struct

+009h   seed_domain
...
+164h   install_time?
```

For example, this is the seed url inside one of the samples;

```
 001CF5D0 global_data     dd 0D104D840h
001CF5D0
001CF5D4 byte_1CF5D4     db 132
001CF5D4
001CF5D5                 db 88, 2, 0
001CF5D8                 db    0
001CF5D9 seed_domain     db 'wtipubctwiekhir.net/gnu/',0
```

I looked a 8 different samples of Shiotob and found three different url seeds:

wtipubctwiekhir.net/gnu/

.	**Sample 1**
sha256	9a386b68c5548b7971b24b58d02abe33fbb4c96ea54b268db7f96a67c81b9c21
md5	1762a640449c489f5f4460898e5fea8e
.	**Sample 2**
sha256	56d30a9aaf45e76d6f9e47ec118eedbddff70cce6f9c147f9f0f1efaf882c51c
md5	242ce522d7b0d26c408b875b5b6ce371
.	**Sample 3**
sha256	f4a7ee8bdc46cc59029ed893899feb8e18a381d3bc6e9b450bf2cb49e15f0956
md5	9a5628f51621466fa0cc8483a4312e12

The first 10 domains for this seed are:

- wtipubctwiekhir.net
- rwmu35avqo12tqc.com
- rskb5bsfhm2fk5h.net
- rbp9pprrxgflut9.com
- zzxeyzgy45yy2a.net
- e3oa4wglvd21xa.com
- mqmq1hvmtxzjv.net
- pd4o4wu24vimn.com
- tlmrzvpbpsqsb.net
- pbmnz59uzndpo.com

n9oonpgabxe31.net/gnu/

.	**Sample 4**
sha256	5bac5fa46973b75acc51f0706303d8ac4e1ee7f829d0e3f8cd7e31a3bb28d9a1
md5	8d360a487c32c7e47d989b061942bf40
.	**Sample 5**
sha256	cc5006de8daaa10d28e75c39bdc5228cf8494b4d6e4b968c10d3d499b90c59ce
md5	d0fcc2f1bfe8b0d5d2433cb4598b00fb

The first 10 domains for this seed are:

- n9oonpgabxe31.net
- q9mqi2au2d5sv.com
- e4zm4yxpnikf2.net
- sen4i12uzyixx.com
- lr1eve4qog1m2.net
- 2oidwapmv2cwp.com
- 5ge4f3gzlywq1.net
- skwskzyp2ktoc.com
- qtiixgafexkgze1.net
- xtjqjmjt344l22w.com

dogcurbctw.com/gnu/

.	**Sample 6**
sha256	fb8f084cc84b6a6abb98228717f10381c74ed55c75f812d5220bec8ad8bc2181
md5	072cce981cecdc97239b30a8479ac067
.	**Sample 7**
sha256	0853c0615e0e486df96c85bf21957411f5fcf6d863b7753fede9774fc5faa5fc
md5	0d846cd79ff72c4e74391baa1b181fae
.	**Sample 8**
sha256	b765e11bd9d44364b284c8420babb400b9140303885e66aca89864efe04ffa34
md5	329b2157a7a0b956a2949a29d152a82a

The first 10 domains for this seed are:

- dogcurbctw.com

- 9g2rdi9uga.net
- vhklwvvon1.com
- bevgfijycd.net
- 5g1xxzjvohrb5.com
- xg5mmhrtbog5b.net
- s2i9eecchnsvh.com
- ka9rik1aqu5li.net
- 5hx2xw4yb52kr.com
- uvwpywhvji3.net

Callback Routine

The seed domain and the timestamp are passed to the `dga_callback` routine that will try to contact one of its C&C servers:

```
 001C4103 mov     edi, offset global_data
...
001C456A lea     edx, [edi+global_struct.seed_domain]
001C456D mov     eax, [edi+168h]
001C4573 and     eax, ds:c_128
001C4579 mov     ecx, dword ptr [edi+global_struct.install_time]
001C457F call    dga_callback
```

The callback routine consists of an infinite loop that tries to POST data to algorithmically generated domains. The loop terminates only when a successful POST is made, including a valid response from the C&C peer.

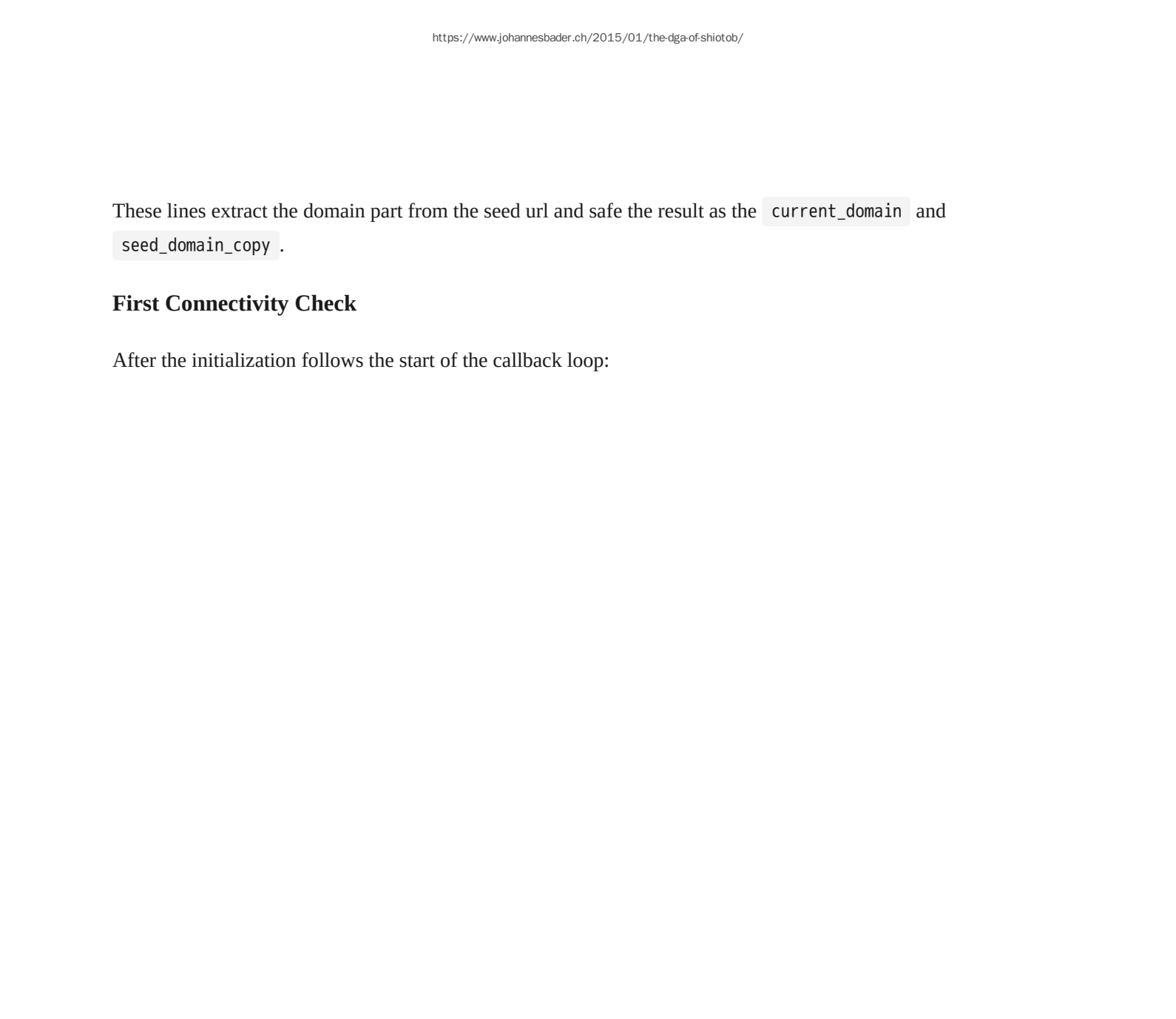
Initialization

The start of `dga_callback` looks like that:

These lines extract the domain part from the seed url and safe the result as the `current_domain` and `seed_domain_copy`.

First Connectivity Check

After the initialization follows the start of the callback loop:



click to enlarge

The routine uses a loop counter in register `ebx` that starts at 1. If the loop counter is at 1 or if it reaches a multiple of 50, then the callback routine performs a connectivity check by contacting *www.google.com*. If Google is unreachable Shiotob sleeps for 10 minutes and retries. Once Google can be reached, Shiotob also tries to contact the seed domain; if it can successfully POST to the seed domain and if the response is as expected, the routine returns:

```
IF ebx == 1 OR ebx % 50 == 0 DO   % do for 1, 50, 100, ...
   WHILE check_connectivity('www.google.com') fails DO
       sleep(10 Minutes)
   contact(seed_domain) % seed domain
   IF "POST" was succesful THEN
       RETURN
```

If contacting the seed domain fails, or if the connectivity check is skipped, then Shiotob calls the DGA routine `the_dga` to generate a new domain. I discuss this routine later in the blog post.

Frequency

Back at the outer callback routine, the domain generated by `the_dga` is turned into an url `https://{domain}/gnu/`, which is then contacted. If the POST is successful, the callback returns. Otherwise, the routine sleeps for 5 seconds less the time it took for the POST:

Second Connectivity Check

Next, if the loop counter is 5 then Shiotob pauses for 5 minutes and enters another connectivity check loop:

Number of Generated Domains

The number of generated domains is time dependent. After every loop, the current time is taken:

The time passed since the start of the malware is then compared to different time spans. For instance, if the uptime is between 3 and 6 days, then the loop counter is reset to 1 after 500 iterations:

When the loop counter is reset to 1, the current domain is also reset to the seed domain. This way the callback routine will generate domains forever - always resetting to the seed domain.

The number of domains increases with time. In total there are four different time spans:

time since malware start	nr of domains
up to 3 days	251

> = 3 days, < 6 days | 501 = 6 days, < 10 days | 1001 more than 10 days | 2001

Summary of the Callback Loop

To summarize, this is how the callback routine iterates over the domains:

```
 domain = seed_domain % initialize with seed domain
ebx = 1
WHILE TRUE:
    IF ebx == 1 OR ebx % 50 == 0 DO   % do for 1, 50, 100, ...
        WHILE check_connectivity('www.google.com') fails DO
            sleep(10 minutes)
        contact(seed_domain)
        IF "POST" was succesful THEN
            RETURN

    domain = the_dga(domain)  % get next domain
    contact(domain)
    IF "POST" was succesful THEN
        RETURN
    sleep(5 seconds)
    IF ebx == 5 and less than 10 minutes passed DO
        sleep(5 minutes)
        WHILE check_connectivity('www.google.com') fails DO
            sleep(10 minutes)

    IF time_passed < 3 days THEN
        iterations = 250
    ELSE IF time_passed < 6 days THEN
        iterations = 500
    ELSE IF time_passed < 6 days THEN
        iterations = 1000
    ELSE
        iterations = 2000

    IF ebx == iterations THEN
        ebx = 1
        domain = seed_domain
```

For example, this is the traffic at the beginning:

1. `02:31:53 HTTP connection, method: GET, URL: http://www.google.com/` : This is part of the connectivity check, since `ebx` = 1
2. `02:31:53 HTTPS connection, method: POST, URL: https://wtipubctwiekhir.net/gnu/` : This is the call to the seed domain, since `ebx` = 1
3. `02:31:53 HTTPS connection, method: POST, URL: https://rwmu35avqo12tqc.com/gnu/` : This is the call after `the_dga` , `ebx` = 1. After this call, the routine sleeps 5 seconds.
4. `02:31:58 HTTPS connection, method: POST, URL: https://rskb5bsfhm2fk5h.net/gnu/` : This is the call after `the_dga` for `ebx` = 2. After this call, the routine sleeps 5 seconds.
5. `02:32:03 HTTPS connection, method: POST, URL: https://rbp9pprrxgflut9.com/gnu/` : This is the call after `the_dga` for `ebx` = 3. After this call, the routine sleeps 5 seconds.
6. `02:32:08 HTTPS connection, method: POST, URL: https://zzxeyzgy45yy2a.net/gnu/:` This is the call after `the_dga` for `ebx` = 4. After this call, the routine sleeps 5 seconds.
7. `02:32:13 HTTPS connection, method: POST, URL: https://e3oa4wglvd21xa.com/gnu/` : This is the call after `the_dga` for `ebx` = 5. After this call, the routine sleeps 5 seconds, and another 5 minutes because `ebx` = 5.
8. `02:36:12 HTTP connection, method: GET, URL: http://www.gstatic.com/generate_204` : This is the connectivity check call for `ebx` = 5.

Next, I'll analyse the heart of the DGA in `the_dga` . This routine is responsible for generating new domains.

The DGA: Fresh Domains by Recurrence Relation

The subroutine `the_dga` gets the current domain passed to in `eax` . Based on the current domain, the algorithm then generates a new domain and returns it.

Disassembly

This is the disassembly of the entire `the_dga` routine:

```
 CODE:001BB508 ; =============== S U B R O U T I N E =======================================
CODE:001BB508
CODE:001BB508 ; Attributes: bp-based frame
CODE:001BB508
CODE:001BB508 the_dga         proc near               ; CODE XREF: sub_40B744+E4p
CODE:001BB508
CODE:001BB508 next_domain     = byte ptr -4Dh
CODE:001BB508 domain[2]       = byte ptr -4Bh
CODE:001BB508 sum_of_chars    = dword ptr -0Ch
CODE:001BB508 var_6           = byte ptr -6
CODE:001BB508 strlen_prev_domain= byte ptr -5
CODE:001BB508 curr_domain     = byte ptr -4
CODE:001BB508
CODE:001BB508                 push    ebp
CODE:001BB509                 mov     ebp, esp
CODE:001BB50B                 add     esp, 0FFFFFFB0h
```

```
CODE:001BB50E                push    ebx
CODE:001BB50F                push    esi
CODE:001BB510                push    edi
CODE:001BB511                mov     [ebp+prev_domain], eax
CODE:001BB514                lea     eax, [ebp+next_domain]
CODE:001BB517                mov     edx, 65
CODE:001BB51C                call    zero_out_ecx_bytes_of_eax ; clean 65 characters of domain
CODE:001BB521                lea     eax, [ebp+next_domain]
CODE:001BB524                mov     edx, dword ptr [ebp+curr_domain]
CODE:001BB527                call    copy_edx_to_eax
CODE:001BB52C                lea     eax, [ebp+next_domain]
CODE:001BB52F                call    strlen
CODE:001BB534                sub     al, 4           ; strlen(tld + .)
CODE:001BB536                mov     [ebp+strlen_prev_domain], al
CODE:001BB539                xor     eax, eax
CODE:001BB53B                mov     [ebp+sum_of_chars], eax
CODE:001BB53E                xor     edx, edx
CODE:001BB540                mov     dl, [ebp+strlen_prev_domain]
CODE:001BB543                test    edx, edx
CODE:001BB545                jl      short loc_40B556
CODE:001BB547                inc     edx
CODE:001BB548                lea     eax, [ebp+next_domain]
CODE:001BB54B
CODE:001BB54B loc_40B54B:                            ; CODE XREF: dga+4Cj
CODE:001BB54B                xor     ecx, ecx
CODE:001BB54D                mov     cl, [eax]
CODE:001BB54F                add     [ebp+sum_of_chars], ecx
CODE:001BB552                inc     eax
CODE:001BB553                dec     edx
CODE:001BB554                jnz     short loc_40B54B
CODE:001BB556
CODE:001BB556 loc_40B556:                            ; CODE XREF: dga+3Dj
CODE:001BB556                xor     edx, edx
CODE:001BB558
CODE:001BB558 loc_40B558:                            ; CODE XREF: dga+7Aj
CODE:001BB558                xor     eax, eax
CODE:001BB55A                lea     esi, [ebp+next_domain]
CODE:001BB55D
CODE:001BB55D loc_40B55D:                            ; CODE XREF: dga+74j
CODE:001BB55D                lea     edi, [eax+edx]
CODE:001BB560                cmp     edi, 41h
CODE:001BB563                jge     short loc_40B577
CODE:001BB565                mov     cl, [ebp+strlen_prev_domain]
CODE:001BB568                imul    cx, [esi]
CODE:001BB56C                xor     cl, [ebp+edi+next_domain]
CODE:001BB570                xor     cl, byte ptr [ebp+sum_of_chars]
CODE:001BB573                mov     [ebp+edi+next_domain], cl
```

```
CODE:001BB577
CODE:001BB577 loc_40B577:                             ; CODE XREF: dga+5Bj
CODE:001BB577                 inc     eax
CODE:001BB578                 inc     esi
CODE:001BB579                 cmp     eax, 66
CODE:001BB57C                 jnz     short loc_40B55D
CODE:001BB57E                 inc     edx
CODE:001BB57F                 cmp     edx, 66
CODE:001BB582                 jnz     short loc_40B558
CODE:001BB584                 mov     cl, [ebp+domain[2]]
CODE:001BB587                 imul    cx, word ptr [ebp+strlen_prev_domain]
CODE:001BB58C                 xor     cl, [ebp+next_domain]
CODE:001BB58F                 xor     eax, eax
CODE:001BB591                 mov     al, cl
CODE:001BB593                 shr     eax, 4
CODE:001BB596                 mov     ecx, eax
CODE:001BB598                 cmp     cl, 10
CODE:001BB59B                 jnb     short loc_40B5A0
CODE:001BB59D                 mov     cl, [ebp+strlen_prev_domain]
CODE:001BB5A0
CODE:001BB5A0 loc_40B5A0:                             ; CODE XREF: dga+93j
CODE:001BB5A0                 xor     edx, edx
CODE:001BB5A2                 mov     dl, cl
CODE:001BB5A4                 test    edx, edx
CODE:001BB5A6                 jl      short loc_40B5BF
CODE:001BB5A8                 inc     edx
CODE:001BB5A9                 lea     eax, [ebp+next_domain]
CODE:001BB5AC
CODE:001BB5AC loc_40B5AC:                             ; CODE XREF: dga+B5j
CODE:001BB5AC                 xor     ebx, ebx
CODE:001BB5AE                 mov     bl, [eax]
CODE:001BB5B0                 shr     ebx, 3
CODE:001BB5B3                 mov     bl, byte ptr ds:aQwertyuiopasdfghjklzxcvb[ebx]
CODE:001BB5B9                 mov     [eax], bl
CODE:001BB5BB                 inc     eax
CODE:001BB5BC                 dec     edx
CODE:001BB5BD                 jnz     short loc_40B5AC
CODE:001BB5BF
CODE:001BB5BF loc_40B5BF:                             ; CODE XREF: dga+9Ej
CODE:001BB5BF                 xor     eax, eax
CODE:001BB5C1                 mov     al, cl
CODE:001BB5C3                 mov     [ebp+eax+next_domain], 0
CODE:001BB5C8                 mov     edx, offset a_net ; ".net"
CODE:001BB5CD                 mov     eax, dword ptr [ebp+curr_domain]
CODE:001BB5D0                 call    sub_401198
CODE:001BB5D5                 test    eax, eax
CODE:001BB5D7                 jz      short loc_40B5EC
```

```
CODE:001BB5D9                 push    offset a_com    ; ".com"
CODE:001BB5DE                 lea     eax, [ebp+next_domain]
CODE:001BB5E1                 push    eax
CODE:001BB5E2                 call    concat
CODE:001BB5E7                 add     esp, 8
CODE:001BB5EA                 jmp     short loc_40B5FD
CODE:001BB5EC ; ---------------------------------------------------------------------------
CODE:001BB5EC
CODE:001BB5EC loc_40B5EC:                             ; CODE XREF: dga+CFj
CODE:001BB5EC                 push    offset a_net    ; ".net"
CODE:001BB5F1                 lea     eax, [ebp+next_domain]
CODE:001BB5F4                 push    eax
CODE:001BB5F5                 call    concat
CODE:001BB5FA                 add     esp, 8
CODE:001BB5FD
CODE:001BB5FD loc_40B5FD:                             ; CODE XREF: dga+E2j
CODE:001BB5FD                 lea     edx, [ebp+next_domain]
CODE:001BB600                 mov     eax, dword ptr [ebp+curr_domain]
CODE:001BB603                 call    copy_edx_to_eax
CODE:001BB608                 pop     edi
CODE:001BB609                 pop     esi
CODE:001BB60A                 pop     ebx
CODE:001BB60B                 mov     esp, ebp
CODE:001BB60D                 pop     ebp
CODE:001BB60E                 retn
CODE:001BB60E the_dga         endp
CODE:001BB60E
CODE:001BB60E ; ---------------------------------------------------------------------------
```

The string `aQwertyuiopasdfghjklzxcvb` has the value “qwertyuiopasdfghjklzxcvbnm123945678”.

Decompilation to Python

Decompiled to Python, the DGA still look rather messy:

```
def get_next_domain(domain):
   qwerty = 'qwertyuiopasdfghjklzxcvbnm123945678'

   def sum_of_characters(domain):
       return sum([ord(d) for d in domain[:-3]])

   sof = sum_of_characters(domain)
   ascii_codes = [ord(d) for d in domain] + 100*[0]
   old_hostname_length = len(domain) - 4
   for i in range(0, 66):
       for j in range(0, 66):
```

```
            edi = j + i
            if edi < 65:
                p = (old_hostname_length * ascii_codes[j])
                cl = p ^ ascii_codes[edi] ^ sof
                ascii_codes[edi] = cl & 0xFF

    """
        calculate the new hostname length
        max: 255/16 = 15
        min: 10
    """
    cx = ((ascii_codes[2]*old_hostname_length) ^ ascii_codes[0]) & 0xFF
    hostname_length = int(cx/16) # at most 15
    if hostname_length < 10:
        hostname_length = old_hostname_length

    """
        generate hostname
    """
    for i in range(hostname_length):
        index = int(ascii_codes[i]/8) # max 31 --> last 3 chars of qwerty unreachable
        bl = ord(qwerty[index])
        ascii_codes[i] = bl

    hostname = ''.join([chr(a) for a in ascii_codes[:hostname_length]])

    """
        append .net or .com (alternating)
    """
    tld = '.com' if domain.endswith('.net') else '.net'
    domain = hostname + tld

    return domain
```

The recurrence relation generates a pseudo random array of 66 chars seeded with the previous domain (nested `for`-loop). Next, the DGA determines the length of the new hostname based on the random data. The determined length will be between 0 and 15 characters; if the picked length is smaller than 10 though, Shiotob will use the length of the previous hostname instead. This means that the domain length will only changes 37.5% of the time. The DGA then determines the new hostname based on the random array using 32 characters from a hard coded string. Finally, the top level domain is picked, alternating between `.net` and `.com`.

From this we can see the following properties of the DGA:

- the top level domains alternate between `.net` and `.com`
- the length of the domain is 10 to 15 characters (excluding the tld), with succeeding domains likely (62.5%) to be of same length.

- the domain consists of all lower case letters and the digits 123459
- the length of the domain and the letters are approximately uniformly distributed.

Summary

The following table summarizes the properties of the DGA

property	**value**
seed	static url, for example: wtipubctwiekhir.net/gnu/
domains per seed	unlimited
domain order	static: seed, 1, 2, 3, …, 50, seed, 51, …
used domains	251, 501, 1001 or 2001, depends on passed time
wait time between domains	normally 5 seconds, 0 seconds after connectivity check, 5:05 after the first 6 DGA domains.
top level domains	*.net* and *.com*, alternating
second level characters	all letters and the digits 123459 (uniformly distributed)
second level domain length	10 to 15 (approx. uniformly distributed)

You find a script to generate the domains on GitHub Gist.
