

## The DGA of Pykspa - "you skype version is old"

Archived: 2026-04-05 16:45:12 UTC

Pykspa (also known as *Pykse*, *Skyper* or *SkypeBot*) is a worm that spreads via Skype, see [“Take a Deep Breath: a Stealthy, Resilient and Cost-Effective Botnet Using Skype” by Antonio Nappa et al.](#) and [“Recognising Botnets in Organisations” by Barry Weymes](#). The malware has a hardcoded list of chat messages which it sends to contacts of the infected Skype user, trying to lure them into clicking on links that install Pykspa on their computer.

Examples of the chat messages from my sample are:

you skype version is old *Pykspa*

I saw you last week. I would like to speak with you *Pykspa*

i lost my job..

i am idiot..

i want to die.. *Pykspa*

[This file](#) lists all chat messages in English. All messages are translated, albeit poorly, to the following languages: *German, Russian, Ukranian, Romanian, Danish, Polish, Italian, Latvian, French, Slovak, Lithuanian, Spanish, Norwegian, Estonian, Swedish, Czech.*

Since at least October 2013, Pykspa comes with a Domain Generation Algorithm (DGA) to contact its Command and Control (C&C) servers. Here is an example of the traffic generated on February 16th, 2015:

```
+--- whatismyip.everdot.org
| www.whatismyip.ca
(1)+ whatismyipaddress.com
| www.showmyipaddress.com
+--- www.whatismyip.com
(2)----> www.google.com
+--> ejtuxsflbknn.net
| haqwrnonlaj.info <--+
| jlbdfghi.net <-----+
| hiyumk.net <-----+
| ezrmsrnmzddx.net <--+
(3)+ okqwmayiseaq.com <--+
| lchsxgzmwg.info <--+---- (4)
+--> wrthooba.info |
| wbtkdnfr.info <-----+
| bodlmkjckx.net <----+
+--> asgwkyaioy.com |
| xcixrwcyesou.net <--+
| ofebrbnbmvf.a.info <--+
```

```
+--> nazihzsljevt.net |  
      bkmothb.net <-----+
```

The IP lookup domains **(1)** are used to determine the IP and location of the host, probably to select the right language for the chat messages. The single call to Google **(2)** is used to determine the current date and time. After that follow two sets of interleaved DGA domains. The set **(3)** likely contains the C&C target, while **(4)** is probably just added as noise.

Both **(3)** and **(4)** use the same DGA, but with different seeds. This blog post shows the DGA of Pykspa, lists the time dependent seeds, and links to some samples on malwr.com that match the DGA and seeds. I analysed [this sample from malwr.com](#), more samples that use Pykspa’s DGA are listed in Section [Samples on Malwr.com](#):

MD5

```
6da71b4317dd664544903cbc872308d7
```

SHA1

```
e373766587b78c4ee7cef9d7a7735b3a52cf41bb
```

SHA256

```
16cf97e7237828c37c796a8f9e81451f7fc301da7fe2ff66d97ce5b44c7dfb42
```

Size

```
1284KB, 1314816 Bytes
```

Compile Timestamp

```
2006-12-09 09:18:24 UTC
```

*(Changes 2015-03-11: Also included discussion of the second set of domains.)*

## Some of the Preliminary Steps

### Anti-VM

Most of the recent Pykspa samples use VM detection. If the sample feels like it is running inside a VM, it immediately shuts down the machine:

```
0040DAE6 shutdown:  
0040DAE6          call    shutdown1  
0040DAEB          call    shutdown2  
0040DAF0  
0040DAF0 loc_40DAF0:  
0040DAF0          call    detect_vm  
0040DAF5          test   al, al  
0040DAF7          jnz   short shutdown
```

The VM detection is based on the exotic “Visual Property Container Extender” assembly call:

```
0040D408 vpcont 7, 0Bh
```

My VM was unmasked by this call; I therefore patched the call to the VM detection routine with `xor eax, eax` :

```
0040DAF0 loc_40DAF0:
0040DAF0          xor     eax, eax
0040DAF2          nop
0040DAF3          nop
0040DAF4          nop
0040DAF5          test   al, al
0040DAF7          jnz    short shutdown
```

## Host-IP

The first network traffic that the sample generates are calls to various IP lookup sites. Pykspa probably uses the information from these sites to geolocate the infected client and choose the appropriate language in Skype chats.

## Current Time

Next, Pykspa enters this code snippet:

These lines first randomly choose one of the following 13 common website (stored at `test_domains` , see offset 40C214):

- `www.google.com`
- `www.facebook.com`
- `www.myspace.com`
- `www.youtube.com`
- `www.yahoo.com`

```
- www.youtube.com
- www.wikipedia.org
- www.blogger.com
- www.adobe.com
- www.bbc.co.uk
- www.imdb.com
- www.baidu.com
- www.ebay.com
```

The malware then calls the subroutine `connectivity_check` which makes a HTTP GET request for the selected domain , e.g.,

```
GET / HTTP/1.1
Host: www.blogger.com
Accept: */*
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; ->
-> rv:1.9.1.3) Gecko/20090824 Firefox/3.5.3
Connection: close
```

Pykspa extracts the `Date` field of the HTTP response's header:

The string is then parsed to get the current date and time. For example, these lines convert “Feb” to the month number 2:

```
0040FC6F loc_40FC6F:
0040FC6F          push    offset aFeb
0040FC74          push    edi
0040FC75          call   esi ; lstrcmpiA
0040FC77          test   eax, eax
0040FC79          jnz    short loc_40FC84
0040FC7B          mov    [ebp+74h+month], 2
0040FC7F          jmp    loc_40FD3C
```

For instance, if the response from `www.blogger.com` is:

```
HTTP/1.1 302 Moved Temporarily
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts/ ->
    -> bin/answer.py?hl=en&answer=151657 for more info."
Content-Type: text/html; charset=UTF-8
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Date: Mon, 09 Mar 2015 11:03:14 GMT
...
```

Then the date is `2015-03-09 11:03:14`. The routine `connectivity_check` returns this date in unix timestamp format in `eax`, or NULL if the date extraction failed. If a date is returned, it is saved in variable `today` at offset `40C224`. If the date extraction failed, for example because internet went down, the snippet sleep 3 seconds at `40C22B` and retries with a different domain for at most 20 times.

After the current time is stored in `today`, the malware starts a stopwatch in line `40c247` which allows it to estimate the current time using only calls to `GetTickCount` (see Section [Seed](#)).

## Callback Loop

As shown in the beginning of the post, Pykspa generates two sets of interleaved DGA domains. These domains are indistinguishable due to using the same algorithm. However, the two sets use a different seed and are generated under different circumstances.

In the following I first show which seed is used when and how the seeds are changed. Next, I show how the initial seed is determined based on the current time. I conclude the section by showing how the DGA actually generates domains based on the current seed.

## When are DGA Calls Made

The callback loop iterates from 0 to 15999:

```
00406B7A xor     dga1_nr, dga1_nr
(...)
00406B82 mov     [ebp+index], dga1_nr
00406B85 next_index:
(...)

00406C64          mov     eax, 16000
(...)
00406C7A          inc     [ebp+index]
00406C7D          cmp     [ebp+index], eax
00406C80          jl     next_index
```

The above loop is wrapped in an infinite loop, which sleeps 1 second before starting the callback loop over:

```
.text:00406C86 push    one_second    ; dwMilliseconds
.text:00406C8C call    ds:Sleep
.text:00406C92 jmp     restart_all
```

To summarize, these are the two loops in pseudo code:

```
WHILE True DO
    // Initialize seeds
    FOR index = 0 TO 15999 DO
        // DGA calls
    END FOR
    sleep 1 second
END WHILE
```

**First Seed - The Useful DGA calls:** Whenever the `index` is divisible by 80, the DGA is called based on the first seed:

```
00406B85      mov     eax, [ebp+index]
00406B88      push   80
00406B8A      cdq
00406B8B      pop    ecx
00406B8C      idiv  ecx
00406B8E      test  edx, edx
00406B90      jnz   short loc_406BF9
00406B92      mov   ecx, [ebp+seed1]
00406B95      mov   eax, ecx
00406B97      lea  ebx, [dga1_nr+1]
00406B9A      div  ebx
00406B9C      lea  eax, [ecx+edx+1]
00406BA0      push  eax          ; seed
00406BA1      mov  [ebp+seed1], eax
(DGA related lines)
00406BE6      mov  dga1_nr, ebx
```

These lines decompile to:

```
IF index % 80 == 0 THEN
    s = seed1 % (dga1_nr + 1)
    seed1 = seed1 + s + 1
    // DGA call
    dga1_nr = dga1_nr + 1
END IF
```

The `dga1_nr` starts at 0 (see offset `00406B7A` ). The initial value of the seed will be discussed later on. Because the index runs from 0 to 15999, there are 200 different domains generated by `seed1` .

**Second Seed - The Noisy DGA calls:** The callback loop can make second DGA calls independent of the above DGA calls. These second DGA calls are based on an independent second seed. This seed is changed every iteration, but DGA calls are only made in about 5% of all iterations:

```
.text:00406BF9 loc_406BF9:
.text:00406BF9 mov     ecx, [ebp+seed2]
.text:00406BFC mov     ebx, [ebp+dga2_nr]
.text:00406BFF mov     eax, ecx
.text:00406C01 xor     edx, edx
.text:00406C03 inc     ebx
.text:00406C04 div     ebx
.text:00406C06 lea    eax, [ecx+edx+1]
.text:00406C0A mov     [ebp+seed2], eax
.text:00406C0D call   _rand
.text:00406C12 push   20
.text:00406C14 cdq
.text:00406C15 pop    ecx
.text:00406C16 idiv   ecx
.text:00406C18 test   edx, edx
.text:00406C1A jnz    short loc_4
```

These line decompile to this pseudo code:

```
s = seed2 % (dga2_nr + 1)
seed2 = seed2 + s + 1
IF rand() % 20 == 0 THEN
    // DGA call
END IF
dga2_nr = dga2_nr + 1
```

Notice that the `seed2` is changed regardless of whether the `seed2` is actually used to generate a new domain. Therefore, there exist 16000 different domains from `seed2`, of which only about 5% or 800 domains are actually generated and used.

The `rand()` call used to determine if a domain is generated or not is based on the current tick count:

```
.text:00406ABE call   ds:GetTickCount
.text:00406AC4 push   eax
.text:00406AC5 call   set_seed
```

This makes the `rand()` function unpredictable. Because each of the domains from the second seed only has a 5% chance of being used, I assume these domains are meant merely to produce noise and not be registered as actual

C&C servers.

## Seed

The initial seeds for both DGA sets are generated almost the same way. First, the current time is determined:

```
00406ACA call    get_timestamp
```

This routine uses the unix timestamp in `today`, which was determined during the [connectivity check](#), and adds the number of seconds that passed since then:

```
.text:00413F2B get_timestamp proc near
.text:00413F2B call    ds:GetTickCount
.text:00413F31 sub     eax, tick_count_after_connectivity_check
.text:00413F37 xor     edx, edx
.text:00413F39 mov     ecx, 1000
.text:00413F3E div     ecx
.text:00413F40 add     eax, today
.text:00413F46 retn
.text:00413F46 get_timestamp
```

This gives an estimate of the current time as unix timestamp. This value — in `eax` — is then divided:

```
.text:00406ACF xor     edx, edx
.text:00406AD1 mov     ecx, 1728000    ; 20 days ...
.text:00406AD6 div     ecx
.text:00406AD8 mov     [ebp+time_divided_by_20days], eax
```

The divisor is the only difference in creating the first and second seed:

- For the first set of domains the divisor is 1728000. This is the number of seconds in **20 days**.
- For the second set of domains the divisor is 86400. This is the number of seconds in **1 day**.

The seed is based on the resulting quotient, and will therefore change once every 20 days for the useful first set of domains, and daily for the noisy second set of domains. The seed initialization continues by creating a 64 bytes long ASCII hex string:

```
.text:00406ADB lea     eax, [ebp+hash_string]
.text:00406AE1 push    eax          ; void *
.text:00406AE2 push    4
.text:00406AE4 pop     edi
.text:00406AE5 lea     eax, [ebp+time_divided_by_20days]
.text:00406AE8 push    edi          ; int
```

```
.text:00406AE9 push    eax            ; int  
.text:00406AEA call    create_hash_string
```

The routine `create_hash_string` is complicated and I didn't reverse engineer the code. It probably is a hash function that returns 256 bytes encoded as a hex string. Pykspa then takes a 4 character substring of this hex string. The start of the substring is determined by taking the time quotient modulo 50:

```
.text:00406AEF mov     eax, [ebp+time_divided_by_20days]  
.text:00406AF2 push   edi            ; size_t  
.text:00406AF3 push   50  
.text:00406AF5 pop    ecx  
.text:00406AF6 xor    edx, edx  
.text:00406AF8 div    ecx            ; offset at most 49  
.text:00406AFA lea   eax, [ebp+edx+hash_string]  
.text:00406B01 push   eax            ; substring
```

For instance, on March 10th, 2015 at 8 pm the `hash_string` for the first set of domains is:

```
b8b8ae799a67ccc2046e97b6935341680e0e830a8920ec393aa8fe12860b9b09
```

The unix timestamp is 1426017600. Divided by twenty days, this becomes 825, which is 25 modulo 50. The malware will therefore use the 4 characters starting at offset 25, i.e., "3534".

These four hex characters and the entire 64 character hash are then passed to routine `calc_seed` that will determine the seed:

```
.text:00406B0B lea   eax, [ebp+hash_string]  
.text:00406B11 push  eax            ; hash  
.text:00406B12 lea   eax, [ebp+seed]  
.text:00406B15 push  edi            ; 4  
.text:00406B16 push  eax            ; target  
.text:00406B17 call  calc_seed  
.text:00406B1C mov   eax, [ebp+seed]  
.text:00406B1F mov   [ebp+original_seed], eax
```

Again, this routine is quite complicated. It probably does some sort of decryption of the substring based on the hash. I couldn't identify the algorithm, and didn't have the time to reverse the code from scratch. I opted instead to let the code calculate the seeds for me using a small debugger script. The procedure was as follows:

1. Attach a debugger to the malware while inside the callback loop.
2. Set a first breakpoint at offset `0x00406ACF`, this is where the current time has just been stored in `eax`.
3. Set a second breakpoint at offset `0x00406B1F`, this is after the first seed has been stored in `eax`.
4. Iterate over all desired timestamps. For each timestamp do the following: (a) Jump to the beginning of the callback routine; (b) Run to the first breakpoint and change `eax` to the desired timestamp; (c) Run to the

second breakpoint, get the seed from `eax` and log the result.

I implemented these steps in the following Immunity Debugger script:

```
import immlib
import time
from datetime import datetime

def main(args):
    imm = immlib.Debugger()
    filename = "seeds.txt"

    with open(filename, "w") as w:
        w.write("first time;last time;seed\n")

    # addresses
    start_of_callback = 0x00406A7C
    after_get_timestamp = 0x00406ACF
    result = 0x00406B1F

    # time values
    twenty_days = 3600*24*20
    timestamp = time.mktime((2008,1,1,0,0,0,1,1,-1))
    timestamp = int(((timestamp//twenty_days)*twenty_days))
    end_timestamp = time.mktime((2016,1,1,0,0,0,4,1,-1))

    # setting breakpoints
    imm.log("setting breakpoints ...")
    imm.setBreakpoint(after_get_timestamp)
    imm.setBreakpoint(result)

    while timestamp < end_timestamp:
        first_time = datetime.fromtimestamp(timestamp).strftime("%Y-%m-%d %H:%M:%S")
        last_time = datetime.fromtimestamp(timestamp + twenty_days - 1).\
            strftime("%Y-%m-%d %H:%M:%S")
        imm.log("getting seed for {}".format(first_time))
        imm.setReg('EIP', start_of_callback)
        imm.run()
        imm.setReg('EAX', timestamp)
        imm.run()
        seed = imm.getRegs()['EAX']
        with open(filename, "a") as a:
            a.write("{};{};{:x}\n".format(first_time, last_time, seed))
        timestamp += twenty_days
    return "done extracting seeds"
```

To generate the seeds of the second set of domains, simply change the breakpoints to 0x406B27 and 0x406B7C respectively.

The following table lists all seeds of the useful domains for the years 2014 and 2015, including the first five domains that the DGA — shown in the [following Section](#) — produces: The time periods are in UTC.

period	seed	first domains
2013-12-21 - 2014-01-10	4ae6a802	yabbpifwawe.info, eszgwhn.net, fgskxmbql.org, ycautv.net, dlhcdwfif.info
2014-01-10 - 2014-01-30	3896367b	jkdcmitej.com, ccnfrsfeht.net, wwlodun.info, ivdjhcwvwnla.info, clvcgwpx.net
2014-01-30 - 2014-02-19	6a2c8eb2	nbmcfivnj.info, wkjywepmjoqx.net, zkkerwhef.com, owvydqimfvl.net, gwiogocc.com
2014-02-19 - 2014-03-11	f96bf2a5	bvxbdfc.org, msmartytyd.info, ncnalaomgmw.com, xvanur.net, yookeocesq.com
2014-03-11 - 2014-03-31	fcef457e	zfbjofqorjhm.info, fjzcpepv.net, msseaycyaek.com, zstalto.net, xmfzbtyaewxb.net
2014-03-31 - 2014-04-20	dd370744	eugcltd.net, bplvoxgfbx.info, maccwskcmsq.com, zwfelqr.net, omizdirtly.info
2014-04-20 - 2014-05-10	a1c5400f	vcsvuay.com, emhebepmd.net, ufakhnixqiwr.info, lfsuyhcrlx.info, ymesmsyiymqo.com
2014-05-10 - 2014-05-30	9d0d436c	wcsjaj.net, hmdfbqzor.info, xigkzlxymfen.com, nfinek.net, yicqumsi.org
2014-05-30 - 2014-06-19	7bb24638	wjfbhnjedf.net, tltpcr.info, sbfebsuys.net, zrghcumanzne.info, jilmfepwgrc.com
2014-06-19 - 2014-07-09	a6b042c	lbjafzzofdx.net, ntilnij.info, vcjdxwbxx.com, cyfwjtjicnep.net, assoka.org
2014-07-09 - 2014-07-29	f34c4850	usyytuo.net, hnhdxtxbvd.info, wmsuiuckkagu.com, rosktbi.net, adgzqqstzn.info
2014-07-29 - 2014-08-18	f2509f0b	quyeek.com, rprumlqy.net, uuocymsegeog.com, rrrkxsp.net, vbkubbxralgs.net
2014-08-18 - 2014-09-07	f0978d0e	mgojzac.info, tkcapzfbrj.net, uscoccqakswe.org, jcrwiwvmq.net, kexlgqb.net

period	seed	first domains
2014-09-07 - 2014-09-27	44dd4e5f	bewbwrj.com, amoudctuy.net, cciksy.com, masvzhr1.net, lkirpo.net
2014-09-27 - 2014-10-17	aac6d7f6	lrvhlp.info, ojxczyten.net, zlyqrojqqhiw.info, yjvmfxydt.info, drdgnmro.info
2014-10-17 - 2014-11-06	f26296b1	nemnhtdcpag.org, rdfml.info, oaiwwgaa.com, mjzknmuztv.net, wzhpoo.info
2014-11-06 - 2014-11-26	c950cbfe	vehbkr.info, zfvszoiql.net, lcqgvikjtg.org, vbthttzj.net, kiusqwmsye.org
2014-11-26 - 2014-12-16	41de3ee2	eiiuvczrb.info, uhorab.net, uqocygsskk.com, ltzzjconbftl.net, fsoelqbsd.com
2014-12-16 - 2015-01-05	f085a446	bmnudlx.info, dsyxxfvtyy.net, bshtdy.info, toefqwaaeim.info, rkskpjb.net
2015-01-05 - 2015-01-25	ffc222d4	xwwglbhkjsl.net, tscdrkt.info, pvldmlteif.org, hldvfrca.net, bsvurea.org
2015-01-25 - 2015-02-14	f3fc72bb	titjxqdwcmd.com, yglkyz.net, qoumyuce.org, rwhnxqtqftkb.net, lhheddtsy.com
2015-02-14 - 2015-03-06	2ff654d0	ejtuxsflbknn.net, wrthooba.info, asgwkyaioy.com, nazihzsljevt.net, tdxbpku.org
2015-03-06 - 2015-03-26	54d64257	nlpfgnhcb.com, horfdqdtia.net, oayeww.org, jmydflbsel.net, dkvyhta.com
2015-03-26 - 2015-04-15	ec343337	rwtyvfrddfk.com, imwgaj.net, mkyaeeye.org, xkvjbgdknfbt.net, teejnil.org
2015-04-15 - 2015-05-05	ccd10407	aaqeemgo.com, gilzxjidxj.net, ywumyoiuuigi.org, uvfpbbsyv.net, tbvtngvxocp.org
2015-05-05 - 2015-05-25	91f7e71c	ytaxprtas.net, hstrdnpcoznh.info, nqpmsct.com, xulciqvvd.net, ggwjhuxqtklf.info
2015-05-25 - 2015-06-14	5cea1d7a	veliren.info, xqzupldhuo.net, ooiomymggqom.org, ucovoaxwi.net, ywmuyucg.org
2015-06-14 - 2015-07-04	b6fcd7ef	hwumvsr.com, gipqptued.net, gickck.com, lobmdjpw.net, nedqtxt.org

period	seed	first domains
2015-07-04 - 2015-07-24	7c562f77	uoakekgy.com, ysbcaebxnt.net, oucmkcyaskua.org, agvsjmdoz.net, omgsgs.com
2015-07-24 - 2015-08-13	4936d2db	wgascuec.com, goxolemcot.net, gyykawsgmeki.org, uiffdqxsf.net, gauaam.com
2015-08-13 - 2015-09-02	b8b2c9e1	uammskmq.org, jqplflktas.info, rybwtr.net, uyznvxlof.info, gakcmqiw.com
2015-09-02 - 2015-09-22	31b275c5	mcuomg.org, qdzziyj.info, cyykckimuy.com, tufbdpbtbxr.net, hofzvmf.org
2015-09-22 - 2015-10-12	a13852	hrlefwwvqqpp.info, dthawyxm.net, aucoakiaicoe.com, voqzaaq.net, ehibnxwshs.info
2015-10-12 - 2015-11-01	6f4f5a0	uauyxamffw.net, mhfira.info, ugfuystwx.net, ffxxnbesitxz.info, hvhkadugb.org
2015-11-01 - 2015-11-21	90864d97	egmsooqueq.com, ckcdyqewadl.net, bwrduhv.info, xkywrr.info, oyaaweoymgm.com
2015-11-21 - 2015-12-11	225a3e31	rgdyrur.org, yfthagxeb.info, rwccjihweor.com, xdhgnb.net, egymosgwqiue.org
2015-12-11 - 2015-12-31	f400ac9c	kzdwohdoo.net, joemwm.info, mkuqmkccgo.org, lqyxzej.net, mydsxhtrli.info
2015-12-31 - 2016-01-20	1f5c0eed	lkwokkrehon.org, cwcrlh.info, hcgktauh.net, dgarcqijrdjc.info, zufrmm.com

You can find more seeds for the useful domains, as well as the seeds for the noisy domains, in the download in the [DGA download](#).

## The DGA

Finally, this section shows how the domains are generated based on the current seed. First, the length of the domains is determined and passed to the dga subroutine `get_sld` (= get second level domain):

```
.text:00406BA4 push    7
.text:00406BA6 pop     ecx
.text:00406BA7 add     eax, dga1_nr
.text:00406BA9 xor     edx, edx
.text:00406BAB div     ecx
.text:00406BAD lea    eax, [ebp+domain]
.text:00406BB0 add     edx, 6
```

```
.text:00406BB3 push    edx            ; length
.text:00406BB4 push    eax            ; domain
.text:00406BB5 call   get_sld
```

This snippet boils down to:

```
length = (seed1 + dga1_nr) % 7 + 6
domain = get_sld(length, seed)
```

The DGA routine to generate the second level domain is quite long, you can [see the full disassembly here](#). The code boils down to this Python snippet:

```
def get_sld(length, seed):
    domain = ""
    modulo = 541 * length + 4
    a = length * length
    for i in range(length):
        index = (a + (seed*((seed % 5) + (seed % 123456) +
            i*((seed & 1) + (seed % 4567))) & 0xFFFFFFFF)) % 26
        a += length;
        a &= 0xFFFFFFFF
        domain += chr(ord('a') + index)
        seed += (((7837632 * seed * length) & 0xFFFFFFFF) + 82344) % modulo;
    return domain
```

Because the seed is passed *by value*, the assignment in the second to last line won't change `seed1` or `seed2`.

Next, the top level domain is chosen randomly from an array of top level domains:

```
.text:00406BBA      add     esp, 0Ch
.text:00406BBD      push   offset a_      ; "."
.text:00406BC2      lea   eax, [ebp+hostname]
.text:00406BC5      push   eax            ; lpString1
.text:00406BC6      call  esi ; lstrcatA
.text:00406BC8      mov   eax, [ebp+asdf]
.text:00406BCB      and   eax, 3
.text:00406BCE      imul  eax, 7
.text:00406BD1      add   eax, offset tld ; "com"
.text:00406BD6      push   eax            ; lpString2
.text:00406BD7      lea   eax, [ebp+hostname]
.text:00406BDA      push   eax            ; lpString1
.text:00406BDB      call  esi ; lstrcatA
```

With the `tld` array:

```
.data:0042E048 tld          db 'com',0          ; DATA XREF: sub_406A7C+1550
.data:0042E048              ; sub_406A7C+1D20
.data:0042E04C              db 0
.data:0042E04D              db 0
.data:0042E04E              db 0
.data:0042E04F              db 'net',0
.data:0042E053              db 0
.data:0042E054              db 0
.data:0042E055              db 0
.data:0042E056              db 'org',0
.data:0042E05A              db 0
.data:0042E05B              db 0
.data:0042E05C              db 0
.data:0042E05D              db 'info',0
.data:0042E062              db 0
.data:0042E063              db 0
.data:0042E064              db 'cc',0
.data:0042E067              db 0
.data:0042E068              db 0
```

So the top level domain is picked according to:

```
tlds = ['com', 'net', 'org', 'info', 'cc']
top_level_domain = tlds[(seed1 & 3)]
```

Note that only the first four top level domains are reachable, “cc” can’t be picked.

## Python Code and Summary

The Python code [in this ZIP download](#) generates the useful domain for any date between 2008 and 2020. It can also generate the noisy domains for the period March 2013 to March 2020. The script uses the list of seeds stored in `dga1_seeds.json` and `dga2_seeds.json`. For instance, to get the four DGA domains from set (3) shown in the introduction:

```
$ python dga.py -d 2015-02-16 -n 4
ejtuxsflbknn.net
wrthooba.info
asgwkyaioy.com
nazihzsljevt.net
```

To confirm that the first noisy domain from (4), i.e., *haqwrnonlaj.info* is covered by the second seed (the date is offset by two days, maybe because of timezone differences?):

```
$ python dga.py -d 2015-02-18 -n 1000 -s 2 | grep -n haqwrnonlaj.info
8:haqwrnonlaj.info
```

The following table summarizes the properties of the DGA:

property	useful domains (first seed)	noisy domains (second seed)
seed	changes every 20 days	changes daily
domains per seed	200	800
tested domains	all	5% per round
sequence	one after another	skipping over 95% of domains
wait time between domains	none	same
top level domain	.com, .net, .org and .info, picked uniformly at random	same
second level characters	lower case letters, picked uniformly at random	same
second level domain length	6 to 12 characters	same

## Samples on Malwr.com

The DGA was probably first used in **October 2013**. The first reference to a DGA domain on Google are for the domain “mczvyzye.net”, which was active from October 3rd to 22nd, 2013, and is listed in [in this analysis](#). The following table lists samples on malwr.com that match the DGA and seeding procedure:

md5	analysis date	seed
<a href="#">467a8f934ba9ea9b438a2c89c9f18c1b</a>	21 Feb. 2014	0xf96bf2a5L
<a href="#">f081f266f1800f6192aa662c9cd15da1</a>	21 Feb. 2014	0xf96bf2a5L
<a href="#">45c750a60992e1f0c433713fbff50734</a>	21 Feb. 2014	0xf96bf2a5L
<a href="#">04d5ee33b95c4ec35ee5295fedf155a9</a>	21 Feb. 2014	0xf96bf2a5L
<a href="#">02a4634b2ad3800f2a8a285933f03946</a>	29 May. 2014	0x9d0d436cL
<a href="#">30ea7bfec0a9a5fa1a94265cba336116</a>	04 Jun. 2014	0x7bb24638

md5	analysis date	seed
<a href="#">0f223c93889d60de3eeec997a17a5121</a>	18 Jun. 2014	0x7bb24638
<a href="#">1c04b0440ca8bf2f9e4fdae6783e480</a>	18 Jun. 2014	0x7bb24638
<a href="#">01d57201b405cc2eec8688ceac6861f6</a>	27 Jun. 2014	0xa6b042c
<a href="#">21bbf78287b44331941e99f124326156</a>	03 Jul. 2014	0xa6b042c
<a href="#">02b9fba52d81bc77f92dd6cbdae2eae1</a>	03 Jul. 2014	0xa6b042c
<a href="#">10aa6b8873010c2158342d2f96f11fc1</a>	03 Jul. 2014	0xa6b042c
<a href="#">16a39f6d50deef6614e2e8cabdc56671</a>	03 Jul. 2014	0xa6b042c
<a href="#">28cbcc88b68bba309fab8229dfb3621</a>	04 Jul. 2014	0xa6b042c
<a href="#">d4fdea06373888645c693ed2c91b9853</a>	08 Jul. 2014	0xa6b042c
<a href="#">f95f4809d1e239da71935728ae588c05</a>	08 Jul. 2014	0xa6b042c
<a href="#">57231e30070e9d500ffa25774809c6b1</a>	13 Jul. 2014	0xf34c4850L
<a href="#">77ce85d6fc611cc533fcc5c235fb71af</a>	20 Jul. 2014	0xf34c4850L
<a href="#">83e4b0ca5ebfa9de4adbc58009629d61</a>	27 Dec. 2014	0xf085a446L
<a href="#">c85ddcad47577b294b13ce3c8f0134bb</a>	14 Jan. 2015	0xffc222d4L
<a href="#">6da71b4317dd664544903cbc872308d7</a>	25 Jan. 2015	0xf3fc72bbL
<a href="#">1667b27c7ca9662330ad15a9ab34dff</a>	16 Feb. 2015	0x2ff654d0
<a href="#">cb07607586d35e8a5832c0149f6c244c</a>	09 Mar. 2015	0x54d64257
<a href="#">1667b27c7ca9662330ad15a9ab34dff</a>	09 Mar. 2015 <sup>R</sup>	0x54d64257
<a href="#">cb07607586d35e8a5832c0149f6c244c</a>	09 Mar. 2015 <sup>R</sup>	0x54d64257

<sup>R</sup>: Reanalysis

Most samples on Malwr were analysed in June and July of 2014, but there are also some recent samples.

---

Source: <https://bin.re/blog/the-dga-of-pykspa/>