

# A Look Back at BazarLoader's DGA

---

 [malwarebookreports.com/a-look-back-at-bazarloaders-dga/](https://malwarebookreports.com/a-look-back-at-bazarloaders-dga/)

muzi

August 6, 2022

I was recently asked a question about DGA and I was unsatisfied with my explanation, so I wanted to write a quick post on DGA, what it is, and how it works. I learned a lot going through this exercise and I hope you enjoy it.

## What is DGA?

---

A Domain Generation Algorithm (DGA) is a technique used by malware authors to generate new domain names for malware command and control. Typically malware will contain a configuration which will house any number of things, including the Command and Control (C2) domains/IPs. While these configurations are typically encrypted within the binary, malware analysts and reverse engineers can often extract these C2s through sandboxes or configuration extractors. This makes it fairly easy, if not trivial, to extract these C2s and put in network blocks. To combat this, malware authors use DGAs to generate domains over time, allowing for a sometimes infinite stream of C2s. This allows for increased persistence if C2 infrastructure is taken down and makes it more difficult to block network traffic.

## How Does a DGA Work?

---

DGAs generate domains over time according to the particular algorithm written in the malware. DGAs often produce their own distinct patterns in the domains they generate. Some DGAs may generate domains by combining multiple words or numbers from a hardcoded dictionary included in the malware. Others will use a seed value to generate a more random looking domain name. Once the domain name has been created, the DGA will then add a top-level domain (TLD), such as .com or .net, to finalize the DGA C2. Because a DGA is capable of producing infinite domains, threat actors do not need to register every domain potentially generated. The threat actor holds the algorithm and therefore can identify when a domain would be generated according to the routine and can register that domain as needed, such as a situation where previous C2 infrastructure is taken down and communication to the implant is lost.

## Defend Against DGAs

---

DGAs can pose a difficult problem to the blue team. Extracting configurations and putting in network blocks is often times trivial, but DGAs present the challenge of preventing virtually infinite combinations of C2 addresses. DGAs embedded in malware can be reverse engineered, the DGA emulated and network blocks put into place, but that task is time consuming and network blocks will quickly get out of hand. The main advantage of DGAs in the modern era is longer lasting infrastructure, as host-based firewalls and EDR products

make network containment of endpoints significantly easier. Threat Researchers and law enforcement often work together to reverse engineer the C2 network protocols and C2 DGA in order to register future domains. In some cases, commands can be sent to the botnet to uninstall the malware, remove critical files, etc. to takedown particular botnets.

## BazarLoader Domain Generation Algorithm

---

A while back I wrote a blog post about BazarLoader, which had been quite prevalent in those days. In more recent times, BazarLoader has all but disappeared in favor of the newer BumbleBee malware. In that blog post, I briefly highlighted the existence of the domain generation algorithm included with BazarLoader, but I did not dive into how it works. Let's take a look at the algorithm and see if we can replicate the algorithm in Python.

BazarLoader's DGA

```

RIP → 000000180007028 48:89c24 08 mov qword ptr ss:[rsp+8],rbx
00000018000702d 48:897424 10 mov qword ptr ss:[rsp+10],rsi
000000180007032 48:897c24 18 mov qword ptr ss:[rsp+18],rdi
000000180007037 55 push rbp
000000180007038 41:56 push r14
00000018000703a 41:57 push r15
00000018000703c 48:8bec mov rbp,rsp
00000018000703f 48:83ec 50 sub rsp,50
000000180007043 48:8bd0 40 mov rbx,qword ptr ss:[rbp+40]
000000180007047 48:8bfa ba 20000000 mov rdi,rdx
00000018000704a 4d:8bf8 mov edx,20
00000018000704f 8bf1 mov esi,ecx
000000180007052 4d:8bf1 mov r14,r9
000000180007054 48:8bcc mov rcx,rbx
000000180007057 44:8d42 e1 lea r8d,qword ptr ds:[rdx-1F]
00000018000705e 48:89ff 40 call 180001000
000000180007063 48:8843 20 mov rax,qword ptr ds:[rbx+20]
000000180007067 45:33db xor r11d,r11d
00000018000706a 48:85c0 test rax,rax
00000018000706d 74 1a je 180007089
00000018000706f 884b 10 mov ecx,dword ptr ds:[rbx+10]
000000180007072 4c:895b 18 mov qword ptr ds:[rbx+18],r11
000000180007076 48:85c0 jmp 180007085
000000180007078 74 0c test rax,rax
00000018000707d 4fc9 je 180007089
00000018000707f 44:8818 dec ecx
000000180007082 48:fcc0 mov byte ptr ds:[rax],r11b
000000180007085 85c9 inc rax
000000180007087 4c:8b4f 20 test ecx,ecx
000000180007089 41:88cb mov r9,qword ptr ds:[rdi+20]
00000018000708d 49:88c1 mov ecx,r11d
000000180007090 4d:85c9 mov rax,r9
000000180007093 0f84 d2010000 test r9,r9
000000180007096 45:3819 je 18000726E
00000018000709c 0f84 c9010000 cmp byte ptr ds:[r9],r11b
00000018000709f 48:fcc0 je 18000726E
0000001800070a5 4fc1 inc rax
0000001800070a8 44:3818 inc ecx
0000001800070aa 75 f6 cmp byte ptr ds:[rax],r11b
0000001800070ad 83f9 06 jne 1800070A3
0000001800070af 44:8bd6 cmp ecx,0
0000001800070b2 0f85 b6010000 mov r10d,esi
0000001800070b8 88 a8901468 mov eax,681490AB
0000001800070bb f7e6 mul esi
0000001800070c0 44:2bd2 sub r10d,edx
0000001800070c2 41:d1ea shr r10d,1
0000001800070c5 44:03d2 add r10d,edx
0000001800070c8 41:clea 08 shr r10d,8
0000001800070cb 41:clea 08

qword ptr [rsp+8]=(000000000090F540)=000000000000FEAE28
rbx=000000000000FEAE28

00000000180007028 <DGA_Algorithm>

```

Address	Hex	ASCII
000000000090F63C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000000000090F64C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000000000090F65C	67 65 6E 65 72 61 74 65 20 45 6d 65 72 63 6e 69	generate Emercoi
000000000090F66C	6E 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00	n.....
000000000090F67C	00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00	.....
000000000090F68C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....:.....
000000000090F69C	01 00 00 00 C0 84 84 00 00 00 00 00 00 00 00 00	.....A.....
000000000090F6AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....if.....
000000000090F6BC	00 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00	.....if.....0a
000000000090F6CC	01 00 00 00 01 00 00 00 00 00 00 00 30 F8 90 00	.....if.....0a
000000000090F6DC	00 00 00 00 00 00 00 00 00 00 00 23 00 00 00 00	.....#.....
000000000090F6EC	00 00 00 00 01 00 00 00 00 00 00 23 00 00 00 00	.....#.....
000000000090F6FC	00 00 00 00 00 00 00 00 00 00 00 F0 01 E0 00	.....@K.....
000000000090F70C	00 00 00 00 40 6F BE 00 00 00 00 00 03 00 00 00	.....@K.....
000000000090F71C	00 00 00 00 30 F8 90 00 00 00 00 84 36 01 80	.....0s.....6..
000000000090F72C	01 00 00 00 F4 DA 3D 00 00 00 00 00 00 00 00 00 00	.....0u.....
000000000090F73C	00 00 00 00 AA FE 00 00 00 00 00 02 00 00 00 00 00	.....0p.....
000000000090F74C	00 00 00 00 48 F9 90 00 00 00 00 05 00 00 00 00 00 00	.....0p.....
000000000090F75C	00 00 00 00 28 31 E2 76 00 00 00 00 00 00 00 00 00 00	.....(1av.....
000000000090F76C	00 00 00 00 28 31 E2 76 00 00 00 00 00 00 00 00 00 00	.....(1av.....
.....?.....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Figure 1: Generate Emercoin – BazarLoader’s DGA Algorithm

## Algorithm Breakdown

Before I break this down, I want to give credit where credit is due. I’m not experienced when it comes to reversing DGAs and working through this algorithm was tough. I came across [this blog post](#) and the author does an incredible job breaking down the algorithm. I heavily relied on it as a reference as I debugged and learned how the algorithm works. I highly recommend reading that post, and others by the author, for an in-depth understanding of BazarLoader’s DGA and how it has evolved over time. With that said, let’s dive into the algorithm.

1. BazarLoader first separates the 26 letter alphabet into two characters classes containing 25 characters total. (j is omitted)
  - 6 vowels **aeiouy**
  - 19 consonants **bcd fghklmnpqrstvwxz**
2. The two sets are then combined into  $2 * 6 * 19$  ordered pairs that contain one vowel and one consonant (Cartesian Product).
3. The 228 resulting pairs are then rearranged with a permutation that is hard-coded into BazarLoader. This permutation is the seed of the BazarLoader DGA.
4. Four pairs are chosen from the 228 pairs and appended together to create the second level domain. The pairs are selected based on the current date (MM-YYYY), where the 2 month digits and last 2 digits of the year are significant. For example, given August 5, 2022, 0822 would be the significant digits used to select the pairs.

#### Character Pool Pairs

Before we jump into our pair selection, let's take a look at our embedded pairs so we understand the character pool pairs the algorithm selects from. When pairs are being selected, bytes are selected from the following two structures, with the two selections being XORed to produce an ASCII character. Each of the two structures were of length 0x1C8 (456), the length expected for 228 character pairs. In order to generate the total character pool, the two structures can be extracted and XORed to produce the pool of character pairs.

```

ciphertext =
bytearray(b'\x10\x9C\x57\xCD\x64\xB2\x33\xCA\x51\xF1\x1E\xF6\x55\xAF\x48\xDC\x4D\x87\
\x76\xFE\x17\x91\x2E\x89
\x35\xC9\x58\xF6\x25\xDB\x25\xB8\x69\xA9\x56\x8F\x1D\x9B\x67\xC9\x33\x83\x72\x86\x66\
\x92\x68\xBD\x46\x96\x2F\xB4\x1F\xC8\x1B\xE6\
\x72\x8B\x1C\xFB\x1A\xF1\x52\x9D\x62\xE9\x33\xBD\x59\x98\x0B\xAB\x4E\xF5\x42\xA7\x51\x
\xEC\x70\xBF\x1E\xCC\x2A\xFF\x38\xAF\x5C\xBA\x
2F\x82\x3E\xC7\x79\xC4\x5F\xD0\x09\xCA\x79\xB2\x22\xE3\x77\xD3\x72\xD5\x78\xD3\x5E\xF
5\x25\xF2\x0D\x8D\x0D\x9C\x79\xED\x00\xBF\x0
E\xB4\x4B\xED\x77\x87\x2A\xE9\x59\xC1\x09\xCC\x49\x81\x59\xED\x4D\xB6\x65\xDE\x14\xB7
\x2F\xB0\x30\xFE\x4F\xC4\x2D\xF2\x25\x91\x7F
\x9E\x5D\xBE\x1B\xA8\x6D\xE9\x22\xB2\x6E\xA8\x74\xA8\x7F\x9A\x49\xB3\x28\x8C\x1C\xEB\
\x0B\xC1\x6C\xA8\x18\xD4\x05\xDE\x58\xA7\x74\
\xDA\x3B\x92\x58\xA8\x18\x8C\x4D\xDB\x4D\x9F\x43\xCD\x62\x93\x0D\xF6\x2A\xC0\x3C\x92\x
5B\x83\x3D\xBD\x25\xF2\x70\xEF\x5E\xFA\x1E\x
E8\x7D\x9A\x34\xDC\x6A\xEA\x6C\xEB\x6A\xF7\x51\xC1\x3F\xC6\x19\xA5\x0B\xBB\x74\xDB\x0
4\x80\x04\x96\x4C\xD2\x65\xB9\x3D\xD0\x51\xF
8\x06\xEA\x5B\xA6\x5C\xC0\x5D\x8A\x72\xF1\x09\x87\x32\x8D\x3F\xDC\x55\xFF\x25\xC7\x34
\xB6\x74\xAA\x4D\x88\x07\x97\x6A\xC3\x2B\x80
\x78\x8E\x7A\x84\x75\xA5\x5F\x88\x3B\xAA\x12\xCA\x12\xF6\x78\x88\x1B\xE6\x0D\xF7\x44\
\x9A\x77\xEB\x27\xA1\x5D\x9C\x0A\xA3\x50\xE9\
\x4C\xAA\x53\xFE\x78\xA8\x12\xC7\x20\xF0\x20\xBC\x51\xA3\x21\x8A\x3E\xDE\x7F\xCA\x5C\x
\xDF\x07\xC9\x7A\xB8\x26\xF4\x6A\xDB\x6F\xCE\x
6E\xDA\x49\xF5\x25\xF1\x17\x87\x07\x92\x79\xE1\x17\xAD\x0E\xB9\x4E\xE3\x70\x8A\x3E\xF
\x5F\xCD\x11\xD4\x52\x8D\x55\xF2\x42\xAD\x6
B\xCB\x04\xA8\x28\xB2\x20\xEB\x46\xDD\x27\xF3\x31\x9B\x7D\x8E\x4B\xBA\x15\xA8\x65\xFD
\x3E\xAC\x62\xAF\x7A\xB5\x74\x8A\x59\xA6\x28
\x84\x0D\xE6\x09\xD5\x63\xAB\x1F\xDA\x1D\xC5\x58\xAB\x73\xCA\x2F\x9D\x48\xA4\x13\x80\
\x5E\xCC\x5C\x94\x5B\xD8\x7A\x9A\x13\xF7\x38\
\xC6\x36\x9B\x43\x9B\x23\xA0\x20\xE4\x66\xE9\x4D\xEF' )

```

```

key =
bytearray(b'\x68\xF9\x2D\xA8\x0A\xDD\x49\xBF\x38\x8B\x6E\x9F\x3A\xDE\x3E\xA9\x28\xEC\
\x1F\x98\x6F\xE8\x5B\xFA\x5A\xA
F\x39\x9A\x5C\xAF\x4E\xD9\x0D\xCC\x3F\xED\x68\xF9\x1E\xBA\x5E\xFA\x1E\xEF\x1F\xFE\x1E
\xDC\x2F\xFD\x5A\xD8\x6B\xBD\x7C\x8F\x19\xFE
\x69\x9F\x7E\x9E\x2B\xEC\x0E\x88\x5D\xD8\x2F\xFD\x69\xCA\x3F\x9A\x2B\xDF\x3E\x8B\x19\
\xDC\x6B\xBD\x4F\x9D\x4D\xD9\x3E\xCF\x59\xEB\
\x5B\xAF\x1B\xAB\x39\xA9\x6C\xAC\x0A\xDD\x49\x8C\x19\xBA\x1A\xBA\x0B\xAA\x39\x9A\x5C\x
\x9C\x6F\xE8\x68\xE8\x0E\x88\x6E\xCA\x7B\xDA\x
3E\x9A\x18\xEF\x5B\x9C\x2B\xA8\x7D\xAD\x28\xEC\x2C\x8A\x3B\xCF\x0C\xAA\x7D\xDA\x49\xD
9\x49\x8C\x2A\xA8\x4E\x9D\x5C\xFA\x1E\xEF\x2
C\xDF\x7A\xCB\x0C\x99\x4D\xD9\x0D\xDD\x0D\xCC\x0C\xFF\x3C\xDE\x49\xEA\x79\x8F\x6E\xAC
\x0A\xDD\x7A\xAD\x6C\xAC\x39\xCF\x1D\xAB\x4E
\xEA\x3D\xCF\x7B\xE9\x2C\xB9\x38\xED\x2C\xB9\x0B\xFF\x78\x9E\x5C\xAF\x5F\xEB\x2C\xEC\
\x4A\xC8\x48\x9D\x09\x99\x2B\x8A\x7F\x8F\x19\
\xEF\x5B\xBE\x0B\x99\x09\x99\x18\x98\x2B\xA8\x4E\xBF\x7C\xCB\x7B\xDA\x1C\xBA\x7C\xE9\x
68\xF9\x2D\xA8\x0A\xDD\x49\xBF\x38\x8B\x6E\x
9F\x3A\xDE\x3E\xA9\x28\xEC\x1F\x98\x6F\xE8\x5B\xFA\x5A\xAF\x39\x9A\x5C\xAF\x4E\xD9\x
D\xCC\x3F\xED\x68\xF9\x1E\xBA\x5E\xFA\x1E\xE
F\x1F\xFE\x1E\xDC\x2F\xFD\x5A\xD8\x6B\xBD\x7C\x8F\x19\xFE\x69\x9F\x7E\x9E\x2B\xEC\x0E
\x88\x5D\xD8\x2F\xFD\x69\xCA\x3F\x9A\x2B\xDF

```

```
\x3E\x8B\x19\xDC\x6B\xBD\x4F\x9D\x4D\xD9\x3E\xCF\x59\xEB\x5B\xAF\x1B\xAB\x39\xA9\x6C\xAC\x0A\xDD\x49\x8C\x19\xBA\x1A\xBA\x0B\xAA\x39\x9A\x5C\x9C\x6F\xE8\x68\x0E\x88\x6E\xCA\x7B\xDA\x3E\x9A\x18\xEF\x5B\x9C\x2B\xA8\x7D\xAD\x28\xEC\x2C\x8A\x3B\xCF\x0C\xAA\x7D\xDA\x49\xD9\x49\x8C\x2A\xA8\x4E\x9D\x5C\xFA\x1E\xEF\x2C\xDF\x7A\xCB\x0C\x99\x4D\xD9\x0D\xDD\x0D\xCC\x0C\xFF\x3C\xDE\x49\xEA\x79\x8F\x6E\xAC\x0A\xDD\x7A\xAD\x6C\xAC\x39\xCF\x1D\xAB\x4E\xEA\x3D\xCF\x7B\xE9\x2C\xB9\x38\xED\x2C\xB9\x0B\xFF\x78\x9E\x5C\xAF\x5F\xEB\x2C\xEC\x4A\xC8\x48\x9D\x09\x99\x2B\x8A' )
```

```
for i in range(len(ciphertext)):  
    ciphertext[i] ^= key[i%len(key)]
```

Resulting Char Pool:

xezenozuizpioqvuekifxyusofalytkadeibubysmyliylvaikultugikuuddoyqlanevebaqoixogicuqebuvbuviehbofyefsokonihosygoynbeetwenuunuwohquritaamugvyitimfiyrelcoykaqqacapokcuydseumafedemfubyirahiquxegceaburotiluhvocywowumoyvupagduobaserroziqyenpahaxiloazodtoishuaxbiufmifoiesleyhzoystreontyuzfaezkypuarywnyavrysiovyczraciosgumuatzommeolxaeqdaevkepeoxsauteppoymxoozwiyygucpyheectelyzayxybgaypakigluinmacageocidsuorwyxuexantigyivewqiadnaawukhirudywaqekidipowihhyopfe

Date Seed

As mentioned above, the current date is used to select character pairs. The analysis below was performed August 04, 2022, meaning **0822** will be used to calculate the pairs.

First Pair

The first pair is selected by splitting the character pool pairs into groups of 19. The first digit of the date is then used as the index of the groups to select. Since the first digit of a month will either be a 0 or 1 (01, 02, 03...10, 11, 12), only two groups can be selected from.

Note: I will use a DGA domain generated during a debugging session as a visual example. To denote which character pair was chosen during this session, the pair will be highlighted in bold font.

```
xe ze no zu iz pi oq vu ek if xy us of al yt ka de ib ub  
ys my li yl va ik ul tu gi ku ud do yq la ne ve ba qo ix
```

Below is what this selection looks like in the debugger.

	<pre> RIP → 000000018000711C 0000000180007120 0000000180007124 0000000180007128 000000018000712A 000000018000712E 0000000180007132 0000000180007137         </pre>	<pre> 42:8A1441 42:321440 48:8B43 20 8810 49:8B4F 10 49:8B46 10 42:8A5441 01 42:325440 01         </pre>	<pre> mov dl,byte ptr ds:[rcx+r8*2] xor dl,byte ptr ds:[rax+r8*2] mov rax,qword ptr ds:[rbx+20] mov byte ptr ds:[rax],dl mov rcx,qword ptr ds:[r15+10] mov rax,qword ptr ds:[r14+10] mov dl,byte ptr ds:[rcx+r8*2+1] xor dl,byte ptr ds:[rax+r8*2+1]         </pre>
<pre> d1=6E 'n' byte ptr [rax+r8*2]=[0000000002C8BE3A]=1E 0000000180007120         </pre>			

Figure 2: Calculate first char of first pair == 'p'

	<pre> RIP → 0000000180007132 0000000180007137 000000018000713C 0000000180007140 0000000180007143         </pre>	<pre> 42:8A5441 01 42:325440 01 48:8B43 20 4D:63C1 8850 01         </pre>	<pre> mov dl,byte ptr ds:[rcx+r8*2+1] xor dl,byte ptr ds:[rax+r8*2+1] mov rax,qword ptr ds:[rbx+20] movsd r8,r9d mov byte ptr ds:[rax+1],dl         </pre>
<pre> d1=9F byte ptr [rax+r8*2+1]=[0000000002C8BE3B]=F6 'ö' 0000000180007137         </pre>			

Figure 3: Calculate second char of first pair == 'i'

Second Pair

The second pair is selected in the same way as the first pair, but groups are picked based on the second digit. The second digit can range from 0-9, so ten different groups are possible.

xe ze no zu iz pi oq vu ek if xy us of al yt ka de ib ub  
 ys my li yl va ik ul tu gi ku ud do yq la ne ve ba qo ix  
 og ic uq eb uv bu vi eh bo fy ef so ko ni ho sy go yn be  
 et we nu un uw oh qu ri ta am ug vy it im fi yr el co yk  
 aq qa ac ap ok cu yd se um af ed em fu by ir ah iq ux eg  
 ce ab ur ot il uh vo cy wo wu mo yv up ag du ob as er ro  
 zi qy en pa ha xi lo az od to is hu ax bi uf mi fo iw es  
 le yh zo yf re on ty uz fa ez ky pu ar yw ny av ry si ov  
 yc zy ra ci os gu mu at yz om me ol xa eq da ev ke **pe** ox  
 sa ut ep po ym xo oz wi yg uc py he ec te ly za yx yb ga

	<pre> RIP → 0000000180007152 0000000180007156 000000018000715A 000000018000715D 0000000180007161 0000000180007165 000000018000716A 000000018000716F 0000000180007173 0000000180007176 0000000180007179 000000018000717D         </pre>	<pre> 42:321440 48:8B43 20 8850 02 49:8B4F 10 49:8B46 10 42:8A5441 01 42:325440 01 48:8B43 20 8850 03 41:8BC2 48:8857 20 41:83E2 03         </pre>	<pre> xor dl,byte ptr ds:[rax+r8*2] mov rax,qword ptr ds:[rbx+20] mov byte ptr ds:[rax+2],dl mov rcx,qword ptr ds:[r15+10] mov rax,qword ptr ds:[r14+10] mov dl,byte ptr ds:[rcx+r8*2+1] xor dl,byte ptr ds:[rax+r8*2+1] mov rax,qword ptr ds:[rbx+20] mov byte ptr ds:[rax+3],dl mov eax,r10d mov rdx,qword ptr ds:[rdi+20] and r10d,3         </pre>
<pre> d1=A '\n' byte ptr [rax+r8*2]=[0000000002C8BF82]=7A 'z' 0000000180007152         </pre>			

Figure 4: Calculate first char of second pair == 'p'

```

RIP → 0000000180007165 42:8A5441 01 mov dl,byte ptr ds:[rcx+r8*2+1]
      000000018000716A 42:325440 01 xor dl,byte ptr ds:[rax+r8*2+1]
      000000018000716F 48:8B43 20 mov rax,qword ptr ds:[rbx+20]
      0000000180007173 8850 03 mov byte ptr ds:[rax+3],dl
      0000000180007176 41:8BC2 mov eax,r10d
      0000000180007179 48:8B57 20 mov rdx,qword ptr ds:[rdi+20]
      000000018000717D 41:83E2 03 and r10d,3

dl=DD 'Y'
byte ptr [rax+r8*2+1]=[0000000002C8BF83]=B8 '.'

000000018000716A

```

Figure 5: Calculate second char of first pair == 'e'

Third Pair

The third pair is selected from groups with a size of 4 pairs. The third digit can range from 0-9, so ten different groups are possible. This digit represents the current decade and therefore the group of 4 pairs **ek if xy us** will remain the same for quite some time.

```

xe ze no zu
iz pi oq vu
ek if xy us
of al yt ka
de ib ub ys
my li yl va
ik ul tu gi
ku ud do yq
la ne ve ba
qo ix og ic

```

RIP →

00000001800071B0	42:321448	xor dl,byte ptr ds:[rax+r9*2]
00000001800071B4	48:8B43 20	mov rax,qword ptr ds:[rbx+20]
00000001800071B8	8850 04	mov byte ptr ds:[rax+4],dl
00000001800071BB	49:8B4F 10	mov rcx,qword ptr ds:[r15+10]
00000001800071BF	49:8B46 10	mov rax,qword ptr ds:[r14+10]
00000001800071C3	42:8A5449 01	mov dl,byte ptr ds:[rcx+r9*2+1]
00000001800071C8	42:325448 01	xor dl,byte ptr ds:[rax+r9*2+1]
00000001800071CD	48:8B43 20	mov rax,qword ptr ds:[rbx+20]
00000001800071D1	8850 05	mov byte ptr ds:[rax+5],dl
00000001800071D4	49:8B46 10	mov rcx,qword ptr ds:[r14+10]
00000001800071D8	49:8B4F 10	mov rax,qword ptr ds:[r15+10]
00000001800071DC	42:8A1441	mov dl,byte ptr ds:[rcx+r8*2]
00000001800071E0	42:321440	xor dl,byte ptr ds:[rax+r8*2]
00000001800071E4	48:8B43 20	mov rax,qword ptr ds:[rbx+20]
00000001800071E8	8850 06	mov byte ptr ds:[rax+6],dl
00000001800071EB	49:8B46 10	mov rcx,qword ptr ds:[r14+10]
00000001800071EF	49:8B4F 10	mov qword ptr ss:[rbp-20],r11
00000001800071F3	4C:895D E0	mov qword ptr ss:[rbp-18],r11
00000001800071F7	4C:895D E8	mov qword ptr ss:[rbp-10],r11
00000001800071FB	4C:895D F0	mov dl,byte ptr ds:[rcx+r8*2+1]
00000001800071FF	42:8A5441 01	xor dl,byte ptr ds:[rax+r8*2+1]
0000000180007204	42:325440 01	mov rax,qword ptr ds:[rbx+20]
0000000180007209	48:8B43 20	mov byte ptr ds:[rax+7],dl
000000018000720D	8850 07	mov rcx,qword ptr ds:[rbx+20]
0000000180007210	48:8B43 20	mov byte ptr ss:[rbp-8],r11b
0000000180007214	44:8858 08	mov byte ptr ss:[rbp-30],r11b
0000000180007218	44:885D D0	mov dword ptr ss:[rbp-2C],57E6691A
000000018000721C	C745 D4 1A69E657	mov dword ptr ss:[rbp-28],20877955
0000000180007223	C745 D8 5579872D	mov eax,dword ptr ss:[rbp-2C]
000000018000722A	8B45 D4	mov al,byte ptr ss:[rbp-30]
000000018000722D	8A45 D0	test al,al
0000000180007230	84C0	jne 18000724C
0000000180007232	75 18	

dl=6F 'o'  
 byte ptr [rax+r9\*2]=[0000000002C8BE44]=17  
 00000001800071B0

Figure 6: Calculate first char of third pair == 'x'

RIP →

00000001800071C8	42:325448 01	xor dl,byte ptr ds:[rax+r9*2+1]
00000001800071CD	48:8B43 20	mov rax,qword ptr ds:[rbx+20]
00000001800071D1	8850 05	mov byte ptr ds:[rax+5],dl
00000001800071D4	49:8B46 10	mov rcx,qword ptr ds:[r14+10]
00000001800071D8	49:8B4F 10	mov rax,qword ptr ds:[r15+10]
00000001800071DC	42:8A1441	mov dl,byte ptr ds:[rcx+r8*2]
00000001800071E0	42:321440	xor dl,byte ptr ds:[rax+r8*2]
00000001800071E4	48:8B43 20	mov rax,qword ptr ds:[rbx+20]
00000001800071E8	8850 06	mov byte ptr ds:[rax+6],dl
00000001800071EB	49:8B46 10	mov rcx,qword ptr ds:[r14+10]
00000001800071EF	49:8B4F 10	mov qword ptr ss:[rbp-20],r11
00000001800071F3	4C:895D E0	mov qword ptr ss:[rbp-18],r11
00000001800071F7	4C:895D E8	mov qword ptr ss:[rbp-10],r11
00000001800071FB	4C:895D F0	mov dl,byte ptr ds:[rcx+r8*2+1]
00000001800071FF	42:8A5441 01	xor dl,byte ptr ds:[rax+r8*2+1]
0000000180007204	42:325440 01	mov rax,qword ptr ds:[rbx+20]
0000000180007209	48:8B43 20	mov byte ptr ds:[rax+7],dl
000000018000720D	8850 07	mov rcx,qword ptr ds:[rbx+20]
0000000180007210	48:8B43 20	mov byte ptr ss:[rbp-8],r11b
0000000180007214	44:8858 08	mov byte ptr ss:[rbp-30],r11b
0000000180007218	44:885D D0	mov dword ptr ss:[rbp-2C],57E6691A
000000018000721C	C745 D4 1A69E657	mov dword ptr ss:[rbp-28],20877955
0000000180007223	C745 D8 5579872D	mov eax,dword ptr ss:[rbp-2C]
000000018000722A	8B45 D4	mov al,byte ptr ss:[rbp-30]
000000018000722D	8A45 D0	test al,al
0000000180007230	84C0	jne 18000724C
0000000180007232	75 18	

dl=E8 'è'  
 byte ptr [rax+r9\*2+1]=[0000000002C8BE45]=91  
 00000001800071C8

Figure 7: Calculate second char of third pair == 'y'

Fourth Pair

The fourth and final pair are selected in the same manner as the third pair. Again, there are 10 potential groups of 4 pairs.

```

xe ze no zu
iz pi oq vu
ek if xy us
of al yt ka
de ib ub ys
my li yl va
ik ul tu gi
ku ud do yq
la ne ve ba
qo ix og ic

```

<b>RIP</b>	→	<b>00000001800071E0</b>	42:321440	xor <b>d1</b> ,byte ptr ds:[rax+r8*2]
•	00000001800071E4	48:8B43 20	mov rax,qword ptr ds:[rbx+20]	
•	00000001800071E8	8850 06	mov byte ptr ds:[rax+6],d1	
•	00000001800071EB	49:8B46 10	mov rax,qword ptr ds:[r14+10]	
•	00000001800071EF	49:8B4F 10	mov rcx,qword ptr ds:[r15+10]	
•	00000001800071F3	4C:895D E0	mov qword ptr ss:[rbp-20],r11	
•	00000001800071F7	4C:895D E8	mov qword ptr ss:[rbp-18],r11	
•	00000001800071FB	4C:895D F0	mov qword ptr ss:[rbp-10],r11	
•	00000001800071FF	42:8A5441 01	mov d1,byte ptr ds:[rcx+r8*2+1]	
•	<b>0000000180007204</b>	42:325440 01	xor d1,byte ptr ds:[rax+r8*2+1]	
•	0000000180007209	48:8B43 20	mov rax,qword ptr ds:[rbx+20]	
•	000000018000720D	8850 07	mov byte ptr ds:[rax+7],d1	
•	0000000180007210	48:8B43 20	mov rax,qword ptr ds:[rbx+20]	
•	0000000180007214	44:8858 08	mov byte ptr ss:[rbp-8],r11b	
•	0000000180007218	44:885D D0	mov byte ptr ss:[rbp-30],r11b	
•	<b>000000018000721C</b>	C745 D4 1A69E657	mov dword ptr ss:[rbp-2C],57E6691A	
•	<b>0000000180007223</b>	C745 D8 5579872D	mov dword ptr ss:[rbp-28],2D877955	
•	000000018000722A	8B45 D4	mov eax,dword ptr ss:[rbp-2C]	
•	000000018000722D	8A45 D0	mov al,byte ptr ss:[rbp-30]	
•	0000000180007230	84C0	test al,al	
•	0000000180007232	75 18	<b>jne 18000724C</b>	
dl=28 'C' byte ptr [rax+r8*2]=[0000000002C8BE40]=4D 'M'				
00000001800071E0				

Figure 8: Calculate first char of fourth pair == 'e'

<b>RIP</b>	→	<b>0000000180007204</b>	42:325440 01	xor <b>d1</b> ,byte ptr ds:[rax+r8*2+1]
•	0000000180007209	48:8B43 20	mov rax,qword ptr ds:[rbx+20]	
•	000000018000720D	8850 07	mov byte ptr ds:[rax+7],d1	
•	0000000180007210	48:8B43 20	mov rax,qword ptr ds:[rbx+20]	
•	0000000180007214	44:8858 08	mov byte ptr ds:[rax+8],r11b	
•	0000000180007218	44:885D D0	mov byte ptr ss:[rbp-30],r11b	
•	<b>000000018000721C</b>	C745 D4 1A69E657	mov dword ptr ss:[rbp-2C],57E6691A	
•	<b>0000000180007223</b>	C745 D8 5579872D	mov dword ptr ss:[rbp-28],2D877955	
•	000000018000722A	8B45 D4	mov eax,dword ptr ss:[rbp-2C]	
•	000000018000722D	8A45 D0	mov al,byte ptr ss:[rbp-30]	
•	0000000180007230	84C0	test al,al	
•	0000000180007232	75 18	<b>jne 18000724C</b>	
dl=EC 'I' byte ptr [rax+r8*2+1]=[0000000002C88E41]=87				
0000000180007204				

Figure 9: Calculate second char of fourth pair == 'k'

Combine Second Level Domain with Top Level Domain

Now that the second level domain of **pipexyek** has been generated, the top level domain of **.bazar** is appended to complete the DGA. ".bazar" can be seen in the above figure as a "tight string." The two hex-values of **0x57E6691A** and **0x2D877955** are combined and XORed with

0x2D870B34, resulting in .bazar.

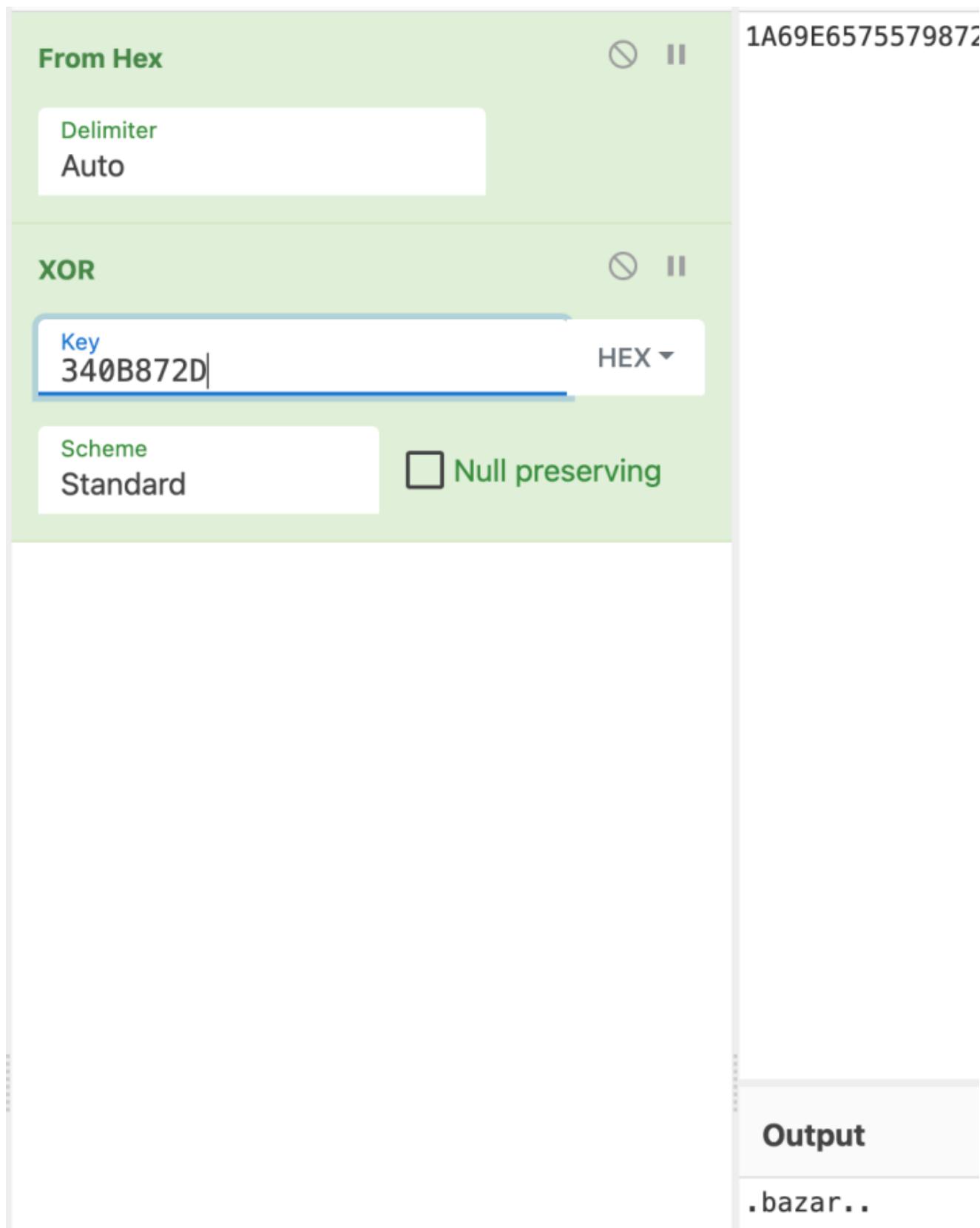


Figure 10: Bazar tight string XOR decrypted

## Algorithm Replication in Python

Once again, I would like to link this [blog post](#) that contained a Python script to emulate this DGA. I learned a ton from this blog and look forward to reading and reversing more DGAs in the future.

```

from binascii import hexlify, unhexlify
import argparse
import logging
import traceback
import os
from datetime import datetime
from collections import namedtuple
from itertools import product

def configure_logger(log_level):
    log_file = os.path.join(os.path.dirname(os.path.realpath(__file__)),
    'bazardga.log')
    log_levels = {0: logging.ERROR, 1: logging.WARNING, 2: logging.INFO, 3:
    logging.DEBUG}
    log_level = min(max(log_level, 0), 3) #clamp to 0-3 inclusive
    logging.basicConfig(level=log_levels[log_level],
                        format='%(asctime)s - %(name)s - %(levelname)-8s %
    (message)s',
                        handlers=[

                            logging.FileHandler(log_file, 'a'),
                            logging.StreamHandler()
                        ])
    class DGA:

        def __init__(self, date: datetime):
            self.logger = logging.getLogger('BazarLoader DGA Generator')
            self.seed = datetime.strftime(date, '%m%Y')

        def decrypt_permutation(self):

            ciphertext =
bytearray(b'\x10\x9C\x57\xCD\x64\xB2\x33\xCA\x51\xF1\x1E\xF6\x55\xAF\x48\xDC\x4D\x87\x
    x76\xFE\x17\x91\x2E\x89\x35\xC9\x58\xF6\x25\xDB\x25\xB8\x69\xA9\x56\x8F\x1D\x9B\x67\x
    C9\x33\x83\x72\x86\x66\x92\x68\xBD\x46\x96\x2F\xB4\x1F\xC8\x1B\xE6\x72\x8B\x1C\xFB\x1
    A\xF1\x52\x9D\x62\xE9\x33\xBD\x59\x98\x0B\xAB\x4E\xF5\x42\xA7\x51\xEC\x70\xBF\x1E\xCC
    \x2A\xFF\x38\xAF\x5C\xBA\x2F\x82\x3E\xC7\x79\xC4\x5F\xD0\x09\xCA\x79\xB2\x22\xE3\x77\x
    D3\x72\xD5\x78\xD3\x5E\xF5\x25\xF2\x0D\x8D\x0D\x9C\x79\xED\x00\xBF\x0E\xB4\x4B\xED\x
    77\x87\x2A\xE9\x59\xC1\x09\xCC\x49\x81\x59\xED\x4D\xB6\x65\xDE\x14\xB7\x2F\xB0\x30\xF
    E\x4F\xC4\x2D\xF2\x25\x91\x7F\x9E\x5D\xBE\x1B\xA8\x6D\xE9\x22\xB2\x6E\xA8\x74\xA8\x7F
    \x9A\x49\xB3\x28\x8C\x1C\xEB\x0B\xC1\x6C\xA8\x18\xD4\x05\xDE\x58\xA7\x74\xDA\x3B\x92\x
    x58\xA8\x18\x8C\x4D\xDB\x4D\x9F\x43\xCD\x62\x93\x0D\xF6\x2A\xC0\x3C\x92\x5B\x83\x3D\x
    BD\x25\xF2\x70\xEF\x5E\xFA\x1E\xE8\x7D\x9A\x34\xDC\x6A\xEA\x6C\xEB\x6A\xF7\x51\xC1\x3
    F\xC6\x19\xA5\x0B\xBB\x74\xDB\x04\x80\x04\x96\x4C\xD2\x65\xB9\x3D\xD0\x51\xF8\x06\xEA
    \x5B\xA6\x5C\xC0\x5D\x8A\x72\xF1\x09\x87\x32\x8D\x3F\xDC\x55\xF\x25\xC7\x34\xB6\x74\x
    AA\x4D\x88\x07\x97\x6A\xC3\x2B\x80\x78\x8E\x7A\x84\x75\xA5\x5F\x88\x3B\xAA\x12\xCA\x
    12\xF6\x78\x88\x1B\xE6\x0D\xF7\x44\x9A\x77\xEB\x27\xA1\x5D\x9C\x0A\xA3\x50\xE9\x4C\xA
    A\x53\xFE\x78\xA8\x12\xC7\x20\xF0\x20\xBC\x51\xA3\x21\x8A\x3E\xDE\x7F\xCA\x5C\xDF\x07\x
    C9\x7A\xB8\x26\xF4\x6A\xDB\x6F\xCE\x6E\xDA\x49\xF5\x25\xF1\x17\x87\x07\x92\x79\xE1\x
    17\xAD\x0E\xB9\x4E\xE3\x70\x8A\x3E\xFF\x5F\xCD\x11\xD4\x52\x8D\x55\xF2\x42\xAD\x6B\x
    CB\x04\xAA\x28\xB2\x20\xEB\x46\xDD\x27\xF3\x31\x9B\x7D\x8E\x4B\xBA\x15\xA8\x65\xFD\x3
    E\xAC\x62\xAF\x7A\xB5\x74\x8A\x59\xA6\x28\x84\x0D\xE6\x09\xD5\x63\xAB\x1F\xDA\x1D\xC5
)

```



```

        upper = lower + p.mod
        ranges.append(list(range(lower, upper)))

    self.logger.debug(ranges)

    # Generate Domains looping indices of Cartesian product
    domains = set()
    for indices in product(*ranges):
        self.logger.debug(indices)
        domain = ""
        for index in indices:
            domain += charpool[index * 2 : index * 2 + 2]
        domain += '.bazar'
        domains.add(domain)

    for domain in domains:
        print(domain)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='BazarLoader String Decryptor')
    parser.add_argument('-v', '--verbose', action='count', default=0,
                        help='Increase verbosity. Can specify multiple times for more verbose
output')
    parser.add_argument('-d', '--date', default=datetime.now().strftime('%Y-%m-%d'),
                        help='Date used for seeding. (e.g. 2022-08-05)')
    args = parser.parse_args()
    configure_logger(args.verbose)
    date = datetime.strptime(args.date, '%Y-%m-%d')
    dga = DGA(date)
    try:
        dga.generate_domains()
    except Exception as e:
        print(f'Exception generating DGA domains.')
        print(traceback.format_exc())

```

Finally, now that we have a list of all the domains and know the algorithm used to generate them, a simple regex can be used to identify any network communications:

**[a-ik-z]{8}\.bazar**

**bazar bazarloader dga**