Next Version of the Bazar Loader DGA

johannesbader.ch/blog/next-version-of-the-bazarloader-dga/

Last week, a new version of the Bazar Loader Domain Generation Algorithm (DGA) appeared. I already analyzed two previous versions, so I'm keeping this post short.

The DGA still uses the eponymous *.bazar* top level domain, but the second level domains are shorter with 8 characters instead of 12 for the previous versions:

```
liybelac.bazar
izryudew.bazar
biymudqe.bazar
fuicibem.bazar
biykonem.bazar
aqtielew.bazar
yptaonem.bazar
exyxtoca.bazar
iqfisoew.bazar
aguponew.bazar
exogelqe.bazar
exybonyw.bazar
etymonac.bazar
```

I analysed the following sample without much obfuscation. There are many other samples that have additional reverse engineering counter measures such as junk code, but a quick comparison revealed no functional differences.

MD5
c6502d4dd27a434167686bfa4d183e89

SHA1
bddbceefe4185693ef9015d0a535eb7e034b9ec3

SHA256
35683ac5bbcc63eb33d552878d02ff44582161d1ea1ff969b14ea326083ea780

Size
336 KB (344576 Bytes)

Compile Timestamp
2020-12-10 13:05:18 UTC

Links
MalwareBazaar, Malpedia, Cape, VirusTotal

Filenames
1ld.3.v1.exe, 35683ac5bbcc63eb33d552878d02ff44582161d1ea1ff969b14ea326083ea780 (VirusTotal)

Detections
Virustotal: 8/72 as of 2020-12-11 02:58:32 - Win64/Bazar.Y (ESET-NOD32), Backdoor.Win32.Bazdor.co (Kaspersky), Trojan.Win64.BAZALOADER.SMYAAJ-A (TrendMicro), Trojan.Win64.BAZALOADER.SMYAAJ-A (TrendMicro-HouseCall)

Unpacking the sample leads to this:

MD5
e44cfd6ecc1ea0015c28a75964d19799

SHA1
cb294c79b5d48840382a06c4021bc2772fdbcf63

SHA256
52e72513fe2a38707aa63fbc52dabd7c7d2c5809ed7e27f384315375426f57bf

Size
96 KB (98816 Bytes)

Compile Timestamp
2020-12-09 10:16:56 UTC

Links
MalwareBazaar, Malpedia, Cape, VirusTotal

Filenames
content.28641.20903.13470.9122.7127 (VirusTotal)

Detections
Virustotal: 4/75 as of 2020-12-15 21:30:37

Reverse Engineering

Apart from the common dynamic loading of Windows API functions and encrypted strings, Bazar Loader relies on *arithmetic substitution via identities* to obfuscate the code. The following relationship is particularly often used:

$$ a \oplus b = (\sim a \cdot b) + (a \cdot \sim b) $$

The same obfuscation is also used by Zloader. It makes the code very hard to read. Here is a small snippet from the DGA:

```
.text:00007FF7A3022F81 push    rbx
.text:00007FF7A3022F82 sub     rsp, 28h
.text:00007FF7A3022F86 mov     r12, rdx
.text:00007FF7A3022F89 mov     r15, rcx
.text:00007FF7A3022F8C movsx   rax, byte ptr [rdx]
.text:00007FF7A3022F90 lea     rcx, [rax+rax*8]
.text:00007FF7A3022F94 lea     rdi, [rax+rcx*2]
.text:00007FF7A3022F98 mov     rbx, 0A74DC40A81F0DEE1h
.text:00007FF7A3022FA2 sub     rdi, rbx
.text:00007FF7A3022FA5 call    lcg
.text:00007FF7A3022FAA movsxd  rcx, eax
.text:00007FF7A3022FAD mov     rbp, 0D79435E50D79435Fh
.text:00007FF7A3022FB7 mov     rax, rcx
.text:00007FF7A3022FBA mul     rbp
.text:00007FF7A3022FBD shr     rdx, 4
.text:00007FF7A3022FC1 lea     rax, [rdx+rdx*8]
.text:00007FF7A3022FC5 lea     rax, [rdx+rax*2]
.text:00007FF7A3022FC9 sub     rcx, rax
.text:00007FF7A3022FCC add     rcx, rdi
.text:00007FF7A3022FCF lea     rdi, [rbx+rcx]
.text:00007FF7A3022FD3 add     rdi, -390h
.text:00007FF7A3022FDA movsx   rax, byte ptr [r12+1]
.text:00007FF7A3022FE0 lea     rcx, [rax+rax*8]
.text:00007FF7A3022FE4 lea     rbx, [rax+rcx*2]
.text:00007FF7A3022FE8 call    lcg
.text:00007FF7A3022FED movsxd  rcx, eax
.text:00007FF7A3022FF0 mov     rax, rcx
.text:00007FF7A3022FF3 mul     rbp
.text:00007FF7A3022FF6 shr     rdx, 4
.text:00007FF7A3022FFA lea     rax, [rdx+rdx*8]
.text:00007FF7A3022FFE lea     rax, [rdx+rax*2]
```

Hex Ray's decompiler also produces really messy code because the arithmetic identities are not simplified:

```
v4 = 19i64 * *szSeed + 0x58B23BF57E0F211Fi64;
v5 = v4 + lcg() % 19ui64 - 0x58B23BF57E0F24AFi64;
v6 = 19i64 * szSeed[1];
v7 = v6 + lcg() % 19ui64;
*szSld = (~ciphertext[2 * v5] & 0xCA | ciphertext[2 * v5] & 0x35) ^ (~key[2 * v5] & 0xCA | key[2 * v5] & 0x35);
*(szSld + 2) = (~ciphertext[2 * v5 + 1] & 0x21 | ciphertext[2 * v5 + 1] & 0xDE) ^ (~key[2 * v5 + 1] & 0x21 | key[2 * v5 + 1] & 0xDE);
*(szSld + 4) = (~ciphertext[2 * v7 - 0x720] & 0x57 | ciphertext[2 * v7 - 0x720] & 0xA8) ^ (~key[2 * v7 - 0x720] & 0x57 | key[2 * v7 - 0x720] & 0xA8);
v8 = (~(2 * v7 - 0x720) & 0x5D381927C02FBDF8i64 | (2 * v7 - 0x720) & 0xA2C7E6D83FD04207ui64) ^ 0x5D381927C02FBDF9i64;
*(szSld + 6) = (~ciphertext[v8] & 0x56 | ciphertext[v8] & 0xA9) ^ (~key[v8] & 0x56 | key[v8] & 0xA9);
LODWORD(v6) = 3 * szSeed[4];
v9 = lcg() % 6 + 2 * v6 - 0x120;
LODWORD(v6) = 3 * szSeed[5];
v10 = lcg() % 6 + 2 * v6 - 0x120;
*(szSld + 8) = (~ciphertext[2 * v9] & 2 | ciphertext[2 * v9] & 0xFD) ^ (~key[2 * v9] & 2 | key[2 * v9] & 0xFD);
v11 = ~(2i64 * v9) & 0x4EC418F7351DF58Fi64 | (2i64 * v9) & 0xB13BE708CAE20A70ui64;
*(szSld + 10) = (key[v11 ^ 0x4EC418F7351DF58Ei64] & ~ciphertext[v11 ^ 0x4EC418F7351DF58Ei64] | ciphertext[v11 ^ 0x4EC418F7351DF58Ei64] & ~key[v11 ^ 0x4EC418F7351DF58Ei64]);
*(szSld + 12) = (~ciphertext[2 * v10] & 0x17 | ciphertext[2 * v10] & 0xE8) ^ (~key[2 * v10] & 0x17 | key[2 * v10] & 0xE8);
v12 = key[2 * v10 + 1];
v13 = ciphertext[2 * v10 + 1];
result = (v12 & ~v13 | v13 & ~v12);
*(szSld + 14) = result;
return result;
}
```

The DGA uses the current month and year as the seed. The seed is stored as a string, and its four ASCII characters are the basis for picking four character pairs. These four pairs are joined to form the 8 second level characters.

The list of character pairs is generated by calculating the cartesian product of the consonants "bcdfghklmnpqrstvwxz" and vowels (with y) "aeiouy". The product is calculated both ways, leading to 19·6·2 character pairs. These pairs are then concatenated into a large string of 456 characters by using a hardcoded sequence of random numbers:

```
qeewcaacywemomedekwyuhidontoibeludsocuexvuuftyliaqydhuizuctuiqow
agypetehfubitiaziceblaogolryykosuptaymodisahfiybyxcoleafkudarapu
qoawyluxqagenanyoxcygyqugiutlyvegahepovyigqyqibaeqynyfkiobpeepby
xaciyvusocaripfyoftesaysozureginalifkazaadytwuubzuvoothymivazyyz
hoevmeburedeviihiravygkemywaerdonoyryqloammoseweesuvfopiriboikuz
orruzemuulimyhceukoqiwfexuefgoycwiokitnuneroxepyanbekyixxiuqsias
```

The string is then encrypted using a random xor key of the same length.

Apart from the date-based seed, the DGA also uses a standard linear congruential generator (LCG) to pick the four character pairs. The LCG is seeded with the current processor tick count and thus unpredictable. For the first two character pairs, the random number is taken mod 19, and for the remaining two pairs mod 6. These numbers correspond to the length of the consonants and vowels array, but make no sense in this context. Because the random numbers are unpredictable, any combination of the 19·19·6·6 = 12996 character pairs could be picked. Bazar Loader generates 10'000 domains per run, but does not guarantee they are unique. On average, 6975 unique domains are expected:

$$ \mathbb{E} = 12996\left(1 - \left(\frac{12996-1}{12966}\right)^{10000} \right) = 6975 $$

Even with the short waiting time between resolving domains, the malware will need to run a long time to get through the list of domains.

Reimplementation in Python

The following Python code shows how the domains are generated:

```
from itertools import product
from datetime import datetime
import argparse
from collections import namedtuple

Param = namedtuple('Param', 'mul mod idx')
pool = (
    "qeewcaacywemomedekwyuhidontoibeludsocuexvuuftyliaqydhuizuctuiqow"
    "agypetehfubitiaziceblaogolryykosuptaymodisahfiybyxcoleafkudarapu"
    "qoawyluxqagenanyoxcygyqugiutlyvegahepovyigqyqibaeqynyfkiobpeepby"
    "xaciyvusocaripfyoftesaysozureginalifkazaadytwuubzuvoothymivazyyz"
    "hoevmeburedeviihiravygkemywaerdonoyryqloammoseweesuvfopiriboikuz"
    "orruzemuulimyhceukoqiwfexuefgoycwiokitnuneroxepyanbekyixxiuqsias"
    "xoapaxmaohezwoildifaluzihipanizoecxyopguakdudyovhaumunuwsusyenko"
    "atugabiv"
)


def dga(date):
    seed = date.strftime("%m%Y")
    params = [
        Param(19, 19, 0),
        Param(19, 19, 1),
        Param(6, 6, 4),
        Param(6, 6, 5)
    ]
    ranges = []
    for p in params:
        s = int(seed[p.idx])
        lower = p.mul*s
        upper = lower + p.mod
        ranges.append(list(range(lower, upper)))

    for indices in product(*ranges):
        domain = ""
        for index in indices:
            domain += pool[index*2:index*2 + 2]
        domain += ".bazar"
        yield domain


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-d", "--date", help="date used for seeding, e.g., 2020-06-28",
        default=datetime.now().strftime('%Y-%m-%d'))
    args = parser.parse_args()
    d = datetime.strptime(args.date, "%Y-%m-%d")
    for domain in dga(d):
        print(domain)
```

Here are all the domains for December 2020, January 2021, February 2021, and March 2021.

Characteristics

The following table summarizes the properties of the new Bazar Loader DGA.

property	value
type	TDD (time-dependent-deterministic)
generation scheme	arithmetic
seed	current date
domain change frequency	every month
unique domains per month	12996
sequence	random selection, might pick domains multiple times
wait time between domains	10 seconds
top level domain	`.bazar`
second level characters	a-z, without j
regex	`[a-ik-z]{8}\.bazar`
second level domain length	8