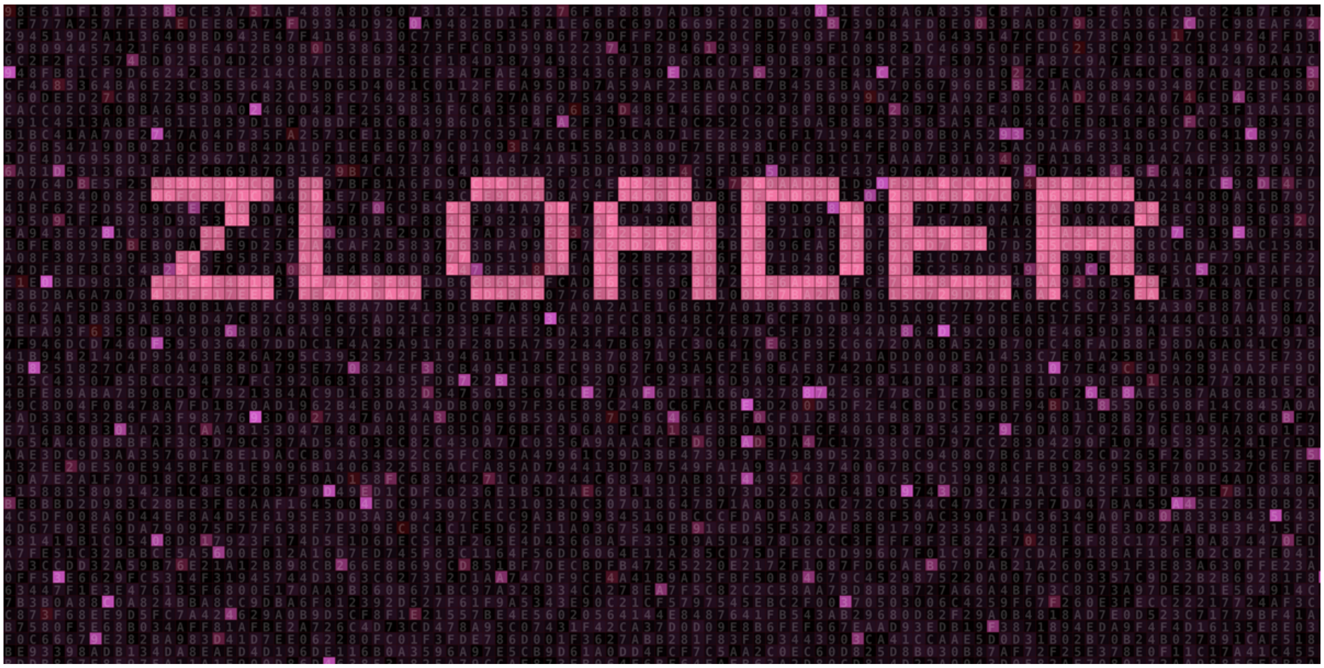
The DGA of Zloader

johannesbader.ch/blog/the-dga-of-zloader/



Zloader — also known as Terdot, DELoader or Zeus Sphinx [1] — is a malware from May 2016 that has resurged in the last few weeks [2] [3] [4] [5] [6]. The last two references [5] [6] are posts by Brad Duncan that mention and list random domain names:

> […] then it started generating DNS queries for random 20-character alphabetic strings with .com as the top level domain (TLD). I've included some examples below.

```
jgqhigsjkulmsvvhshmk.com
wapjdxlstholqwakofgi.com
aiavxvlshmkweccksfky.com
liswrfujohqsnbnohetn.com
hciqylualwcnyvajdkqq.com
pdtlshacpbacpnhcndpd.com
kdacggcctwcavdgvpbmk.com
wapwtpwciertrhkdaxrp.com
shyjgiyhyegxeqqpdtya.com
gccggcctwcerlshacpba.com
cpnhcndpdkylibtlbeco.com
bxhwpdkqdakbplfvfqwn.com
bioonshmwrbecckfcavh.com
```

Similar 20-character domains had also been referenced by Twitter User TomasP, who apparently also reverse engineered the domain generation algorithm (DGA) [7], yet did not publish it. Twitter User DynamicAnalysis subsequently published DGA domains on Pastebin

— for example [8] [9] — but only for one particular seed.

This blog post shows how to reverse engineer the algorithm and presents as a result a reimplementation in Python. The post is based on the following sample, but I also looked at other samples to find additional seeds.

MD5
afdf2fbc0756ed304d1a33083a5f2b0f

SHA1
f3a25627f925390097a64a84ef34c952fe8af036

SHA256
a947c216ea52ce23457b3babb1e1eb6275cabe2150d3995553e4de4b8c3d97f4

Size
323 KB (330752 Bytes)

Compile Timestamp
2019-05-27 07:19:22 UTC

Links
MalwareBazaar, URLHaus, Twitter, VirusTotal

Filenames
antiamsi.bin (MalwareBazaar), antiamsi.bin (VirusTotal)

Detections
MalwareBazaar: ZLoader, **Virustotal**: 52/74 as of 2020-04-25 03:46:18 - TrojanSpy:Win32/Glupteba.ef0afc48 (Alibaba), Trojan:Win32/Glupteba.RRS!MTB (Microsoft), Win32.Trojan-spy.Zbot.Lscl (Tencent), Trojan-Spy.Win32.Zbot.zzac (ZoneAlarm)

The sample is — as is customary — packed. Unpacking it leads to this sample:

MD5
c844efe1b7e76cbdea36ce62ff788de9

SHA1
d8143cf09bff7b0ca2a0c777912746a5922104ee

SHA256
835048e00ba3babf6f920c9a4c2863865a5dcf8e0b6ede4f57c63aeb9cb5c147

Size
184 KB (188416 Bytes)

Compile Timestamp
2020-04-08 18:19:58 UTC

Links
MalwareBazaar, Malpedia, Dropped_by_md5, VirusTotal

Detections
Virustotal: 30/74 as of 2020-04-25 20:55:07 - a variant of Win32/Spy.Zbot.ADI (ESET-NOD32), W32/Zbot.ADI!tr (Fortinet), HEUR:Backdoor.Win32.Dridex.vho (Kaspersky), BehavesLike.Win32.Adopshel.ch (McAfee-GW-Edition), HEUR:Backdoor.Win32.Dridex.vho (ZoneAlarm)

This sample creates a new Windows installer process `msiexec.exe` in suspended state. It then writes an encrypted copy of itself into `msiexec.exe`, as well as a decryption stub. The thread context is set to the stub and execution of the thread is resumed. The decryption stub decrypts the injected binary and jumps to the first subroutine at offset 0x1C90. I dumped the sample with entry point set to this starting point. It should be "runnable" if loaded with image base `0x03090000`:

MD5
5c76c41f9d0cc939240b3101541b5475

SHA1
da361ec6976d3d9225ce40951b26d1d8ecdb7fd1

SHA256
4029f9fcba1c53d86f2c59f07d5657930bd5ee64cca4c5929cbd3142484e815a

Size
208 KB (212992 Bytes)

Compile Timestamp
2020-04-08 18:19:58 UTC

Links
MalwareBazaar, Malpedia, Dropped_by_md5, VirusTotal

Detections
Virustotal: 22/74 as of 2020-04-25 20:55:24 - Win32/Spy.Zbot.ADI (ESET-NOD32), BScope.Trojan-Spy.Zbot (VBA32)

The following analysis is based on this last sample (f3f2393a838d417ff8f823a235bd83f2) loaded at image base 0x03091CD2.

Reverse Engineering

The analysis of the sample is complicated mainly by three techniques:

1. The strings are encrypted. I chose the Appcall functionality of IDA Pro to decrypt them dynamically.
2. API calls are hidden by dynamically resolving them using function hashes. Again, Appcalls to evaluate the routine that resolves the API reveal most API names.

3. Constant unfolding, dead code insertion and artithmetic substitution via indentities. The first two are mostly removed by the Hex Rays decompiler, and the arithemetic identities can be easily simplified with basic logical equivalences. I'll show an example of this when analysing the DGA

Decryption of strings takes one function argument - the offset to the ciphertext:

```
.text:03091CDB 68 B4 CA 0B 03          push    offset dword_30BCAB4
.text:03091CE0 E8 1B 17 01 00          call    decrypt_string  ; BOT-INFO
```

The comment next to the `decrypt_string` function call with the plaintext was found by running the following IDA script:

```
from idc import *
from idautils import *
import idaapi
import sys
import string
import re

RESOLVER_TYPE_DEC = "char *__cdecl decrypt_string(char *a1, char *a2);"
m = re.search("\s([^ (@]+)[(@]", RESOLVER_TYPE_DEC)
RESOLVER_NAME = m.group(1)

resolver_addr = get_name_ea_simple(RESOLVER_NAME)
if resolver_addr == idaapi.BADADDR:
    print(RESOLVER_NAME + " not defined")
    sys.exit()

resolver = idaapi.Appcall.typedobj(RESOLVER_TYPE_DEC)
resolver.ea = resolver_addr


def previous_heads(ea):
    """ iterator to get previous instructions of an address (no including itself) """
    if not idc.is_head(idc.get_full_flags(ea)):
        ea = idaapi.next_head(ea, ea+1000)
    ea = idaapi.prev_head(ea,0)
    while ea != idaapi.BADADDR:
        yield ea
        ea = idaapi.prev_head(ea, 0)

def do():
    """ count the nr of references to the resolver function """
    xrefs = list(CodeRefsTo(resolver_addr,1))
    """ iterate over all references """
    for i, xr in enumerate(xrefs):
        print("[-] tackling {:08X}".format(xr))
        args = []
        for x in previous_heads(xr):
            args.append(get_operand_value(x, 0))
            if len(args) >= 1:
                break
        empty = Appcall.buffer(" ", 1000)
        args.append(empty)
        try:
            r = resolver(*args)
        except Exception as e:
            print("FAILED: appcall failed: {}".format(e))
            continue
        try:
            name = empty.value
        except:
            print("FAILED: to read back buffer)
            continue
        print("OK: found {}".format(name))
        set_cmt(xr, name, True)
```

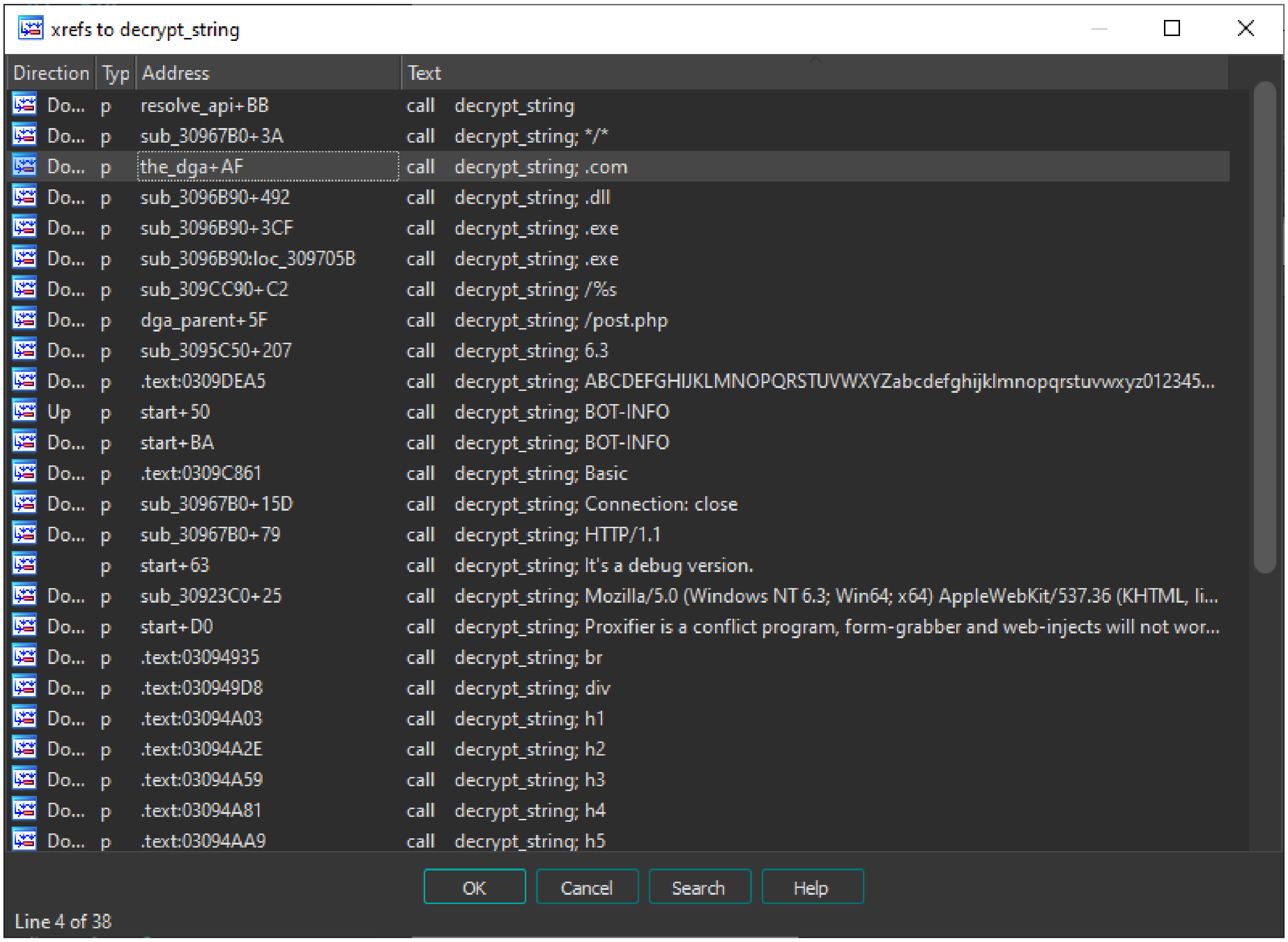
```
do()
```

Windows API functions such as *InternetConnectA* are dynamically resolved and then called:

```
.text:030917DC 68 E1 75 E7 0A          push    0AE775E1h
.text:030917E1 6A 13                   push    13h
.text:030917E3 E8 88 19 01 00          call    resolve_api     ;
wininet_InternetConnectA
.text:030917E8 83 C4 08                add     esp, 8
.text:030917EB 0F B7 4D 10             movzx   ecx, [ebp+arg_8]
.text:030917EF 6A 00                   push    0
.text:030917F1 6A 00                   push    0
.text:030917F3 6A 03                   push    3
.text:030917F5 6A 00                   push    0
.text:030917F7 6A 00                   push    0
.text:030917F9 51                      push    ecx
.text:030917FA 53                      push    ebx
.text:030917FB 56                      push    esi
.text:030917FC FF D0                   call    eax
```

A similar IDA Pro script as the one to decrypt the strings was used to find the API names and comment the disassembly.

Listing all references to the string decryption routine, we see one producing the plaintext “*.com*”:



Because the strings are deciphered immediatly before they are used, the decryption call for *.com* leads to the DGA routine. IDA Pro does a very good job of decompiling the routine. I renamed a few variables and subroutines to get to this C code:

```
int __cdecl the_dga(int dwSeed, int nNumberOfDomains, int pArrayOfDomains)
{
  int result; // eax
  unsigned int r; // esi
  int i; // edi
  unsigned int offset; // ebx
  char the_letter; // al
  unsigned int dwSeedXored_1; // ebx
  char *szTLD_1; // eax
  int i_1; // [esp-10h] [ebp-48h]
  char szTLD[19]; // [esp+1h] [ebp-37h]
  _DWORD the_domain_object[3]; // [esp+14h] [ebp-24h]
  unsigned int dwSeedXored; // [esp+20h] [ebp-18h]
  int iDomainNr; // [esp+24h] [ebp-14h]
  char szDomain[13]; // [esp+2Bh] [ebp-Dh]

  if ( nNumberOfDomains )
  {
    r = dwSeed;
    result = 0;
    dwSeedXored = dwSeed ^ 0x81716ECC;
    do
    {
      iDomainNr = result;
      initialize(the_domain_object);
      i = 0;
      do
      {
        offset = r % get_nr_25();
        the_letter = offset + get_nr_97();
        dwSeedXored_1 = dwSeedXored;
        szDomain[0] = the_letter;
        update_domain_object(szDomain);
        r = dwSeedXored_1 ^ or(~(r + szDomain[0]) & 0x81716ECC, (r + szDomain[0]) &
0x7E8E9133, 0);
        i_1 = i++;
        plus(i_1, 1, 0, 0);
      }
      while ( i != get_nr_20() );
      szTLD_1 = decrypt_string(szTLDCiphertext, szTLD);
      concatenate(szTLD_1);
      save_in_array((_DWORD *)pArrayOfDomains, (int)the_domain_object);
      reset(the_domain_object);
      result = plus_0(iDomainNr + 0x6A6E645D, 1u, 0) - 0x6A6E645D;
    }
    while ( result != nNumberOfDomains );
  }
  return result;
}
```

The code is already pretty readable. The only non obvious part is the random number calculation — variable *r* — which requires some basic logical computation. Let the seed `dwSeed` be *s*, and `szDomain[0]` be *l*, then the next number is determined as follows ($\cdot$, $\oplus$, and $+$ stand for logical and, xor and or respectively):

$$ \begin{align*} r &= (s \oplus \text{0x81716ECC}) \oplus (\sim (r + l) \cdot \text{0x81716ECC}) + ((r + l)\cdot \text{0x7E8E9133}) \end{align*} $$

The two constants have the following relationship:

$$ 0x81716ECC = \sim 0x7E8E9133 \text{ mod } 2^{32} $$

Futhermore

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

So by setting *k= 0x81716ECC* we get:

$$ \begin{align*} r &= (s \oplus k) \oplus (\sim (r + l) \cdot k) + ((r + l)\cdot \sim k) \\ &= s \oplus k \oplus ((r+l) \oplus k) \\ &= s \oplus (r + l) \end{align*} $$

This leads to the following Python code for the DGA:

```
def dga(seed, nr_of_domains):
    domains = []

    r = seed;
    for i in range(nr_of_domains):
        domain = ""
        for j in range(20):
            letter = ord('a') + (r % 25)
            domain += chr(letter)
            r = seed ^ ( (r + letter) & 0xFFFFFFFF)
        domain += ".com"
        print(domain)
```

Looking at the caller of the domain generation routine, we see how the seed is calculated:

```
.text:03095540 push    ebp
.text:03095541 mov     ebp, esp
.text:03095543 push    ebx
.text:03095544 push    edi
.text:03095545 push    esi
.text:03095546 sub     esp, 16Ch
.text:0309554C lea     edi, [ebp+pS]
.text:03095552 mov     [ebp+var_1C], ecx
.text:03095555 push    edi
.text:03095556 call    decrypt_config_rc4
.text:0309555B add     esp, 4
.text:0309555E lea     esi, [ebp+pArrayOfDomains]
.text:03095561 mov     ecx, esi
.text:03095563 call    sub_30BA8E0
.text:03095568 call    get_today_at_0UTC
.text:0309556D mov     [ebp+dwSeed], eax
.text:03095570 call    sub_30A5260
.text:03095575 lea     ecx, [ebp+dwSeed]
.text:03095578 push    edi
.text:03095579 push    eax
.text:0309557A push    ecx
.text:0309557B call    rc4_encrypt
.text:03095580 add     esp, 0Ch
.text:03095583 mov     edi, [ebp+dwSeed]
.text:03095586 call    get_nr_of_domains
.text:0309558B push    esi
.text:0309558C push    eax
.text:0309558D push    edi
.text:0309558E call    the_dga
```

First, the config of Zloader is decrypted with RC4. The RC4 key `djluflczrgefphtiwegc` is hardcoded in the sample. The config contains the hardcoded domains which are contacted before the DGA domains are used — if at all. At the end of the config, there is a new RC4 key `q23Cud3xsNf3` which is used for seeding the DGA:

```
s
TelegramCrypt
AntiAMSIdoc
http://wmwifbajxxbcxmucxmlc.com/post.php
http://pwkqhdgytsshkoibaake.com/post.php
http://snnmnkxdhflwgthqismb.com/post.php
http://iawfqecrwohcxnhwtofa.com/post.php
http://nlbmfsyplohyaicmxhum.com/post.php
http://fvqlkgedqjiqgapudkgq.com/post.php
http://cmmxhurildiigqghlryq.com/post.php
http://nmqsmbiabjdnuushksas.com/post.php
http://fyratyubvflktyyjiqgq.com/post.php
q23Cud3xsNf3
```

The seed is based on the unix timestamp for the current date at time 00:00 UTC. This 32bit value is representated in little endian order, and the four bytes are RC4 encrypted with the key from the config, i.e., `q23Cud3xsNf3`. The result is then interpreted as the little endian representation of the seed. The following Python snippet shows the seeding procedure.

```
key = "q23Cud3xsNf3"
rc4 = RC4(key)
d = d.replace(hour=0, minute=0, second=0)
timestamp = int((d - datetime(1970, 1, 1)).total_seconds())
p = struct.pack("<I", timestamp)
c = rc4.encrypt(p)
seed = struct.unpack("<I", c)[0]
```

With the seeding procedure and the DGA finished, we can now give a complete reimplementation of the DGA.

Reimplementation in Python

The following Python code can be used to generate the Zloader domains for any date and RC4 seed value. For example, to generate the domains for April 25, 2020 and seed `q23Cud3xsNf3` do `dga.py -d 2020-04-25 --rc4 q23Cud3xsNf3`. You also find the algorithm in my domain generation GitHub repository.

```
from datetime import datetime
import struct
import argparse


class RC4:

    def __init__(self, key_s):
        key = [ord(k) for k in key_s]

        S = 256*[0]
        for i in range(256):
            S[i] = i

        j = 0
        for i in range(256):
            j = (j + S[i] + key[i % len(key)]) % 256
            S[i], S[j] = S[j], S[i]

        self.S = S
        self.i = 0
        self.j = 0

    def prng(self):
        self.i = (self.i + 1) % 256
        self.j = (self.j + self.S[self.i]) % 256
        self.S[self.i], self.S[self.j] = self.S[self.j], self.S[self.i]
        K = self.S[(self.S[self.i] + self.S[self.j]) % 256]
        return K

    def encrypt(self, data):
        res = bytearray()
        for d in data:
            c = d ^ self.prng()
            res.append(c)
        return res

    def __str__(self):
        r = ""
        for i, s in enumerate(self.S):
            r += f"{i}: {hex(s)}\n"
        return r


def seeding(d, key):
    rc4 = RC4(key)
    d = d.replace(hour=0, minute=0, second=0)
    timestamp = int((d - datetime(1970, 1, 1)).total_seconds())
    p = struct.pack("<I", timestamp)
    c = rc4.encrypt(p)
    seed = struct.unpack("<I", c)[0]
    return seed


def dga(seed, nr_of_domains):
```

```
    r = seed
    for i in range(nr_of_domains):
        domain = ""
        for j in range(20):
            letter = ord('a') + (r % 25)
            domain += chr(letter)
            r = seed ^ ((r + letter) & 0xFFFFFFFF)
        domain += ".com"
        print(domain)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--date", help="date when domains are generated")
    parser.add_argument("-r", "--rc4",
            help="rc4 key from config",
            choices=["q23Cud3xsNf3","41997b4a729e1a0175208305170752dd",
"kZieCw23gffpe43Sd"],
            default="q23Cud3xsNf3")


    args = parser.parse_args()
    if args.date:
        d = datetime.strptime(args.date, "%Y-%m-%d")
    else:
        d = datetime.now()
    seed = seeding(d, args.rc4)
    dga(seed, 32)
```

Other Samples - Other Seeds

For reference, this section lists three more samples that I have analyzed and which have resulted in two additional seeds. You find precalculated lists of the DGA domains for all three seeds in my domain generation GitHub repository [10].

md5	seed	list of domains
afdf2fbc0756ed304d1a33083a5f2b0f	q23Cud3xsNf3	list
2169e871d4ca668d1872722d1a0695dc	q23Cud3xsNf3	list
fa9b3dfdb4b97dfe0db5991472f89399	41997b4a729e1a0175208305170752dd	list
306212efebc6ac92000687393e56a5cb	kZieCw23gffpe43Sd	list

2169e871d4ca668d1872722d1a0695dc

MD5
2169e871d4ca668d1872722d1a0695dc

SHA1

add2bbbac042c328ed71c9fd2efcb9cbce5a89f7

SHA256
cc87e6581ca91f941f65332b2de0e681d58491b54aff9d0b30afae828a5f5790

Size
539 KB (552448 Bytes)

Compile Timestamp
2020-04-14 11:20:46 UTC

Links
MalwareBazaar, URLhaus, VirusTotal

Filenames
SecuriteInfo.com.Win32.GenKryptik.EILT.4491 (MalwareBazaar), output.155861665.txt, Thusput, Thusput.DLL, znvmzdd.dll, ZnVmZdD.dll, april14.dll (VirusTotal)

Detections
Virustotal: 42/75 as of 2020-04-18 16:11:27

unpacks to

MD5
6a900d6f8af3a1a0e31ca5bb63637d03

SHA1
221ab3d8ab16a0a7790026aab9b26904be6db436

SHA256
e4d0a79d2463c5d3a71874e3389fa753f480b96639ad32baf1997baf8e5f714a

Size
187 KB (191488 Bytes)

Compile Timestamp
2020-04-08 18:20:42 UTC

Links
MalwareBazaar, Malpedia, Dropped_by_md5, VirusTotal

Detections
Virustotal: 29/75 as of 2020-04-25 20:58:26

The config is encrypted with RC4 key `edykepkrqahpyxabcwgm`. These are the hardcoded domains:

```
http://wmwifbajxxbcxmucxmlc.com/post.php
http://ojnxjgfjlftfkkuxxiqd.com/post.php
http://pwkqhdgytsshkoibaake.com/post.php
http://snnmnkxdhflwgthqismb.com/post.php
http://iawfqecrwohcxnhwtofa.com/post.php
http://nlbmfsyplohyaicmxhum.com/post.php
http://fvqlkgedqjiqgapudkgq.com/post.php
http://cmmxhurildiigqghlryq.com/post.php
http://nmqsmbiabjdnuushksas.com/post.php
http://fyratyubvflktyyjiqgq.com/post.php
```

The RC4 key for the DGA seed is `q23Cud3xsNf3`.

fa9b3dfdb4b97dfe0db5991472f89399

MD5
fa9b3dfdb4b97dfe0db5991472f89399

SHA1
5677f26e926c8c8d7f7bf7eb085a9e48549a268b

SHA256
3648fe001994cb9c0a6b510213c268a6bd4761a3a99f3abb2738bf84f06d11cf

Size
512 KB (524288 Bytes)

Compile Timestamp
2020-04-20 10:48:16 UTC

Links
MalwareBazaar, URLHaus, Twitter, VirusTotal

Filenames
f.dll (MalwareBazaar), Letter ease, Letter ease.DLL, f.dll (VirusTotal)

Detections
MalwareBazaar: ZLoader, **Virustotal**: 50/75 as of 2020-04-24 02:51:48

unpacks to

MD5
133b1861b3590bf00308509227f82872

SHA1
eb6f12759da7aa84077143e3e2694b6fda3d5631

SHA256
dd11381223ab1902db2963df4cbe3299e42064a5857545560f913647c1f70c5a

Size
187 KB (191488 Bytes)

Compile Timestamp
2020-04-08 18:20:42 UTC

Links
MalwareBazaar, Malpedia, Dropped_by_md5, VirusTotal

Detections
Virustotal: 29/74 as of 2020-04-25 21:00:11

The config is encrypted with RC4 key `dqhfltvppmucpvebkqtn`. These are the hardcoded domains:

```
https://dcaiqjgnbt.icu/wp-config.php
https://nmttxggtb.press/wp-config.php
```

The RC4 key for the DGA seed is `41997b4a729e1a0175208305170752dd`.

306212efebc6ac92000687393e56a5cb

MD5
306212efebc6ac92000687393e56a5cb

SHA1
dc0b678e9ad7cadd5de907bf80fa351d5d3347cc

SHA256
8d5a770975e52ce1048534372207336f6cc657b43887daa49994e63e8d7f6ce1

Size
856 KB (877056 Bytes)

Compile Timestamp
2020-04-05 16:19:02 UTC

Links
MalwareBazaar, VirusTotal

Filenames
JtVhjtbGMAbrWft.dll (MalwareBazaar), FfIYXQPKpCQymHQ.exe, PkRWAytIAsEHwhy.exe, qKMCMByhJjQpfmZ.exe, FmgJjYLZmscJaur.exe, gGwBVwnpxkyFNlc.exe, ZhbIdJYZzrkPQGs.exe, eIGmAdVpMFJxmrk.exe, VUCJyZshHrMGvdT.exe, WHFQhvaOzqkkTFk.exe, dFVlQGPNqrdhrCE.exe, tnXoUCMnjELKOYm.exe, dTEAUJnMdnADEVG.exe, omih.dll, ikhaapd.dll, 2020-04-07-ZLoader-DLL-binary.bin, etidwuv.dll, ekydn.dll, upiqwoq.dll, ryubn.dll, JtVhjtbGMAbrWft.exe, icobyg.dll, GnbjtDwFOsvocUW.exe, CbxfejTbfqXuuIT.exe, JtVhjtbGMAbrWft.bin (VirusTotal)

Detections
Virustotal: 58/75 as of 2020-04-20 00:40:47

unpacks to

MD5
4a74e2d34230bbc705f39e6943c859d3

SHA1
410c1c03a52dbd56e78b0487ec532e68eb1c64e4

SHA256
60544c6694620488b69e568b15c96b33971dd7343ba63da31f993332852871c2

Size
172 KB (176640 Bytes)

Compile Timestamp
2020-03-30 18:35:43 UTC

Links
MalwareBazaar, Malpedia, Dropped_by_md5, VirusTotal

Detections
Virustotal: 34/75 as of 2020-04-25 20:59:59

The config is encrypted with RC4 key `cbstobypqnbsnnpehdtb`. These are the hardcoded domains:

```
https://knalc.com/sound.php
https://namilh.com/sound.php
https://ronswank.com/sound.php
https://stagolk.com/sound.php
https://mioniough.com/sound.php
https://ergensu.com/sound.php
```

The RC4 key for the DGA seed is `kZieCw23gffpe43Sd`.

---

1. Zloader on Malpedia ↩
2. Zloader downloads on URLHaus ↩
3. Threatpost - Zeus Sphinx Banking Trojan Arises Amid COVID-19 ↩
4. Security Intelligence - Zeus Sphinx Trojan Awakens Amidst Coronavirus Spam Frenzy ↩
5. Malware-Traffic-Analysis.net - A2020-04-07 - PCAP AND MALWARE FOR AN ISC DIARY (ZLOADER) ↩ ↩
6. SANS ISC Diary - German malspam pushes ZLoader malware ↩ ↩
7. Tweet by TomasP mentioning the DGA ↩
8. Zloader domains of seed 41997b4a729e1a0175208305170752dd for 2020-04-20 ↩

9. Zloader domains of seed 41997b4a729e1a0175208305170752dd for 2020-04-14 ↩

10. Github repository ↩