It’s a BEE! It’s a… no, it’s ShadowPad.

medium.com/insomniacs/its-a-bee-it-s-a-no-it-s-shadowpad-aff6a970a1c2

asuna amawaka November 19, 2021

asuna amawaka

Nov 19, 2021

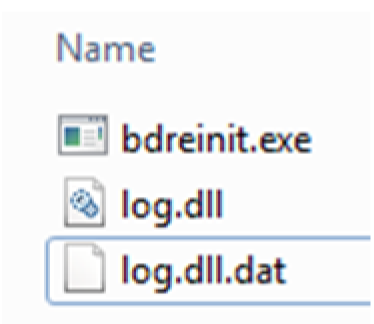
·

8 min read

ShadowPad, a malware name that is familiar to many, became widely known since 2017 through its participation in a series of supply-chain attacks in CCleaner, NetSarang and ASUS. There has been a lot of good work describing how clusters of activities by different threat actors (APT41, Fishmonger, Tonto Team, RedFoxTrot, BackdoorDiplomacy are all the “big names” we know) are linked by ShadowPad. I thought to join in the fun of reverse engineering one of the variant of ShadowPad that I found from VirusTotal. So let’s begin!

Side-note: Kaspersky calls this variant “ShadowShredder” [1]. This variant has also been documented by PTSecurity [2] in Jan 2021 and was also recently in TeamT5’s presentation in VB2021localhost conference [3].

The malware trinity (DLL load-order attack) analyzed in this post:



Legitimate EXE, bdreinit.exe (Bitdefender’s Crash Handler) — SHA256: 386EB7AA33C76CE671D6685F79512597F1FAB28EA46C8EC7D89E58340081E2BD

Malicious DLL, log.dll — SHA256: 8D1A5381492FE175C3C8263B6B81FD99AACE9E2506881903D502336A55352FEF

Encrypted Payload, log.dll.dat — SHA256: 0371FC2A7CC73665971335FC23F38DF2C82558961AD9FC2E984648C9415D8C4E

I found these files separately, so they may not be originally intended as a package. I managed to piece them together based on their filenames, and they work fine as a set. These files happened to have debugging strings included, which makes the analysis slightly more pleasing to follow.

Let’s start with observations from dynamic analysis.

Here’s what the log looked like in the debugger upon execution without breakpoints:

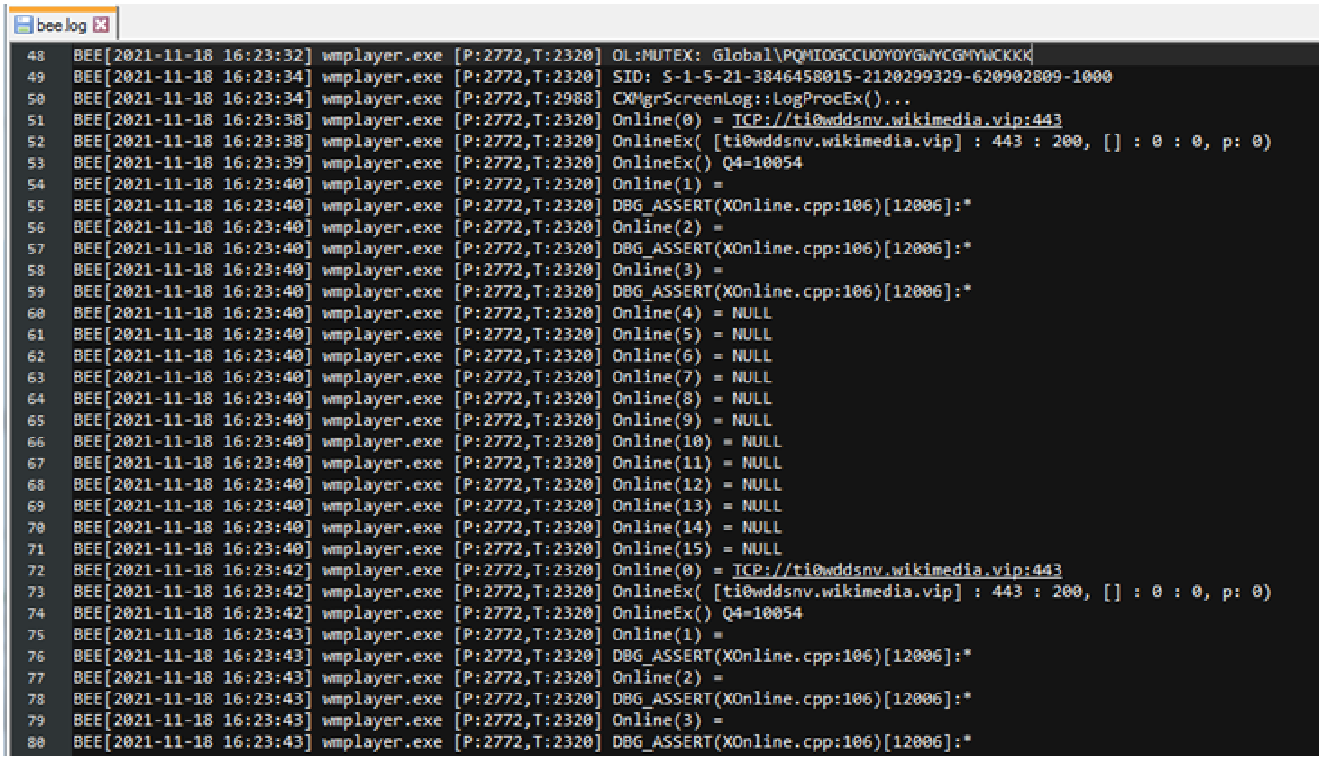
```
DLL Loaded: 6FA80000 C:\exe\log.dll
DebugString: "BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad::path_exe: C:\exe\bdreinit.exe"
DebugString: "BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad::path_dll: C:\exe\log.dll"
DebugString: "BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad::path_img: C:\exe\log.dll.dat"
DebugString: "BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad:KEY(SOFTWARE\Classes\CLSID\{e3f825af-f27f-5b95-50b2c77deeac30cf}:D572770E)"
DebugString: "BEE[2021-11-18 14:46:19] bdreinit.exe [P:1432,T:2684] ScLoad::ScQueryFromFile() OK"
DebugString: "BEE[2021-11-18 14:46:19] bdreinit.exe [P:1432,T:2684] ScLoad:shellcode()..."
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[0]InstDir: %ALLUSERSPROFILE%\DRM\Test\"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[1]InstExe: Test.exe"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[2]InstDll: log.dll"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[4]SvcName: MyTest"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[5]SvcDisp: MyTest"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[6]SvcDesc: MyTest"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[7]KeyPath: SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[8]KeyName: MyTest"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst::CopyFileW(C:\exe\bdreinit.exe,C:\ProgramData\DRM\Test\Test.exe)"
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] DBG_ASSERT(XInstall.cpp:484)
[32]:The process cannot access the file because it is being used by another process."
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] DBG_ASSERT(Sc.cpp:225)
[32]:The process cannot access the file because it is being used by another process."
DebugString: "BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] SID: S-1-5-21-3846458015-2120299329-620902809-1000"
DLL Loaded: 70980000 C:\Windows\SysWOW64\apphelp.dll
DebugString: "BEE[2021-11-18 14:46:22] bdreinit.exe [P:1432,T:2684] CreateProcessAsUserW(pid:2748) ok"
DebugString: "BEE[2021-11-18 14:46:22] bdreinit.exe [P:1432,T:2684] CXInject(pid:2748,path:C:\Program Files (x86)\Windows Media Player\wmplayer.exe) OK!"
DebugString: "BEE[2021-11-18 14:46:22] bdreinit.exe [P:1432,T:2684] Inject [C:\Program Files (x86)\Windows Media Player\wmplayer.exe] OK!"
Process stopped with exit code 0x0
Saving database to C:\Program Files\x64dbg\release\x32\db\bdreinit.exe.dd32 0ms
Debugging stopped!
```

logged by x32dbg

The same debugging information is also written into a file C:\ProgramData\bee.log.

```
BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad::path_exe: C:\exe\bdreinit.exe
BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad::path_dll: C:\exe\log.dll
BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad::path_img: C:\exe\log.dll.dat
BEE[2021-11-18 14:46:17] bdreinit.exe [P:1432,T:2684] ScLoad:KEY(SOFTWARE\Classes\CLSID\{e3f825af-f27f-5b95-50b2c77deeac30cf}:D572770E)
BEE[2021-11-18 14:46:19] bdreinit.exe [P:1432,T:2684] ScLoad::ScQueryFromFile() OK
BEE[2021-11-18 14:46:19] bdreinit.exe [P:1432,T:2684] ScLoad:shellcode()...
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[0]InstDir: %ALLUSERSPROFILE%\DRM\Test\
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[1]InstExe: Test.exe
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[2]InstDll: log.dll
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[4]SvcName: MyTest
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[5]SvcDisp: MyTest
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[6]SvcDesc: MyTest
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[7]KeyPath: SOFTWARE\Microsoft\Windows\CurrentVersion\Run
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst[8]KeyName: MyTest
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] Inst::CopyFileW(C:\exe\bdreinit.exe,C:\ProgramData\DRM\Test\Test.exe)
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] DBG_ASSERT(XInstall.cpp:484)[32]:The process cannot access the file because it is being used by another process.
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] DBG_ASSERT(Sc.cpp:225)[32]:The process cannot access the file because it is being used by another process.
BEE[2021-11-18 14:46:21] bdreinit.exe [P:1432,T:2684] SID: S-1-5-21-3846458015-2120299329-620902809-1000
BEE[2021-11-18 14:46:22] bdreinit.exe [P:1432,T:2684] CreateProcessAsUserW(pid:2748) ok
BEE[2021-11-18 14:46:22] bdreinit.exe [P:1432,T:2684] CXInject(pid:2748,path:C:\Program Files (x86)\Windows Media Player\wmplayer.exe) OK!
BEE[2021-11-18 14:46:22] bdreinit.exe [P:1432,T:2684] Inject [C:\Program Files (x86)\Windows Media Player\wmplayer.exe] OK!
BEE[2021-11-18 14:46:24] wmplayer.exe [P:2748,T:1268] OL:MUTEX: Global\PQMIOGCCUOYOYGWYCGMYWCKKK
BEE[2021-11-18 14:46:30] wmplayer.exe [P:2748,T:1268] SID: S-1-5-21-3846458015-2120299329-620902809-1000
BEE[2021-11-18 14:46:30] wmplayer.exe [P:2748,T:1652] CXMgrScreenLog::LogProcEx()...
BEE[2021-11-18 14:46:34] wmplayer.exe [P:2748,T:1268] Online(0) = TCP://ti0wddsnv.wikimedia.vip:443
BEE[2021-11-18 14:46:35] wmplayer.exe [P:2748,T:1268] OnlineEx( [ti0wddsnv.wikimedia.vip] : 443 : 200, [] : 0 : 0, p: 0)
```

bee.log



```
BEE[2021-11-18 16:23:32] wmplayer.exe [P:2772,T:2320] OL:MUTEX: Global\PQMIOGCCUOYOYGWYCGMYWCKKK
BEE[2021-11-18 16:23:34] wmplayer.exe [P:2772,T:2320] SID: S-1-5-21-3846458015-2120299329-620902809-1000
BEE[2021-11-18 16:23:34] wmplayer.exe [P:2772,T:2988] CXMgrScreenLog::LogProcEx()...
BEE[2021-11-18 16:23:38] wmplayer.exe [P:2772,T:2320] Online(0) = TCP://ti0wddsnv.wikimedia.vip:443
BEE[2021-11-18 16:23:38] wmplayer.exe [P:2772,T:2320] OnlineEx( [ti0wddsnv.wikimedia.vip] : 443 : 200, [] : 0 : 0, p: 0)
BEE[2021-11-18 16:23:39] wmplayer.exe [P:2772,T:2320] OnlineEx() Q4=10054
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(1) =
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(2) =
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(3) =
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(4) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(5) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(6) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(7) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(8) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(9) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(10) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(11) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(12) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(13) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(14) = NULL
BEE[2021-11-18 16:23:40] wmplayer.exe [P:2772,T:2320] Online(15) = NULL
BEE[2021-11-18 16:23:42] wmplayer.exe [P:2772,T:2320] Online(0) = TCP://ti0wddsnv.wikimedia.vip:443
BEE[2021-11-18 16:23:42] wmplayer.exe [P:2772,T:2320] OnlineEx( [ti0wddsnv.wikimedia.vip] : 443 : 200, [] : 0 : 0, p: 0)
BEE[2021-11-18 16:23:42] wmplayer.exe [P:2772,T:2320] OnlineEx() Q4=10054
BEE[2021-11-18 16:23:43] wmplayer.exe [P:2772,T:2320] Online(1) =
BEE[2021-11-18 16:23:43] wmplayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
BEE[2021-11-18 16:23:43] wmplayer.exe [P:2772,T:2320] Online(2) =
BEE[2021-11-18 16:23:43] wmplayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
BEE[2021-11-18 16:23:43] wmplayer.exe [P:2772,T:2320] Online(3) =
BEE[2021-11-18 16:23:43] wmplayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
```

more of bee.log

It seems that I've picked up a test sample. The C2 domain configured within is a subdomain of wikimedia[.]vip, which has been associated with Funnydll and other ShadowPad samples.

I'm curious about the author's choice of name — What does "BEE" stand for?

I extracted all the interesting filename strings, for reference if anyone is interested at guessing the full suite of capabilities within ShadowPad:

```
XMgrService.cpp
XMgrScreen.cpp
XMgrScreenLog.cpp
XMgrShell.cpp
XMgrProcess.cpp
XMgrDisk.cpp
XMgrKeyLogger.cpp
XMgrRegister.cpp
XMgrPortMap.cpp
XMgrRecentFiles.cpp
XSo.cpp
XSoClass.cpp
XSoTcp.cpp
XSoUDP.cpp
XSoRTP.cpp
XSoPipe.cpp
XJoinSvr.cpp
XJoin.cpp
XEveryone.cpp
XHandle.cpp
XService.cpp
XInterface.cpp
XDebug.cpp
XOnline.cpp
XFireWall.cpp
XImpUserService.cpp
XImpUser.cpp
XInstall.cpp
XInject.cpp
XStream.cpp
XStreamFile.cpp
XProxy.cpp
XPacket.cpp
XPktMap.cpp
XConfig.cpp
XString.cpp
XDIBBitmap.cpp
Sc.cpp
User.cpp
```

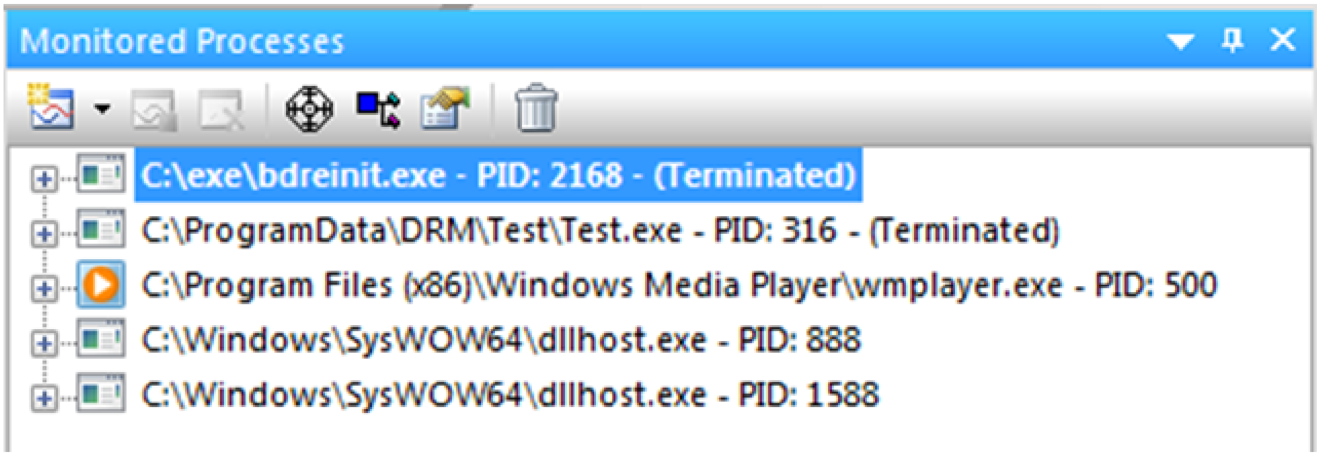
source filenames

And here’s the list of debugging messages found in memory:

```
CXInstall::Uninst():delete file2: %s
CXInstall::Uninst():delete file1: %s
CXInstall::InstDeleteCallback():delete file2: %s
CXInstall::InstDeleteCallback():delete file1: %s
UninstSvc(State:%d) wait
OnlineEx( [%s] : %d : %d, [%s] : %d : %d, p: %d)
OnlineEx() Q0=%d
OnlineEx() Q1=%d
OnlineEx() Q2=%d
OnlineEx() Q3=%d
OnlineEx() Q4=%d
OnlineEx() Q7
OnlineEx() Q8
Online(%d) = %s
Online(%d) = NULL
Inst::CopyFileW(%s,%s)
Inst[8]KeyName: %s
Inst[7]KeyPath: %s
Inst[6]SvcDesc: %s
Inst[5]SvcDisp: %s
Inst[4]SvcName: %s
Inst[2]InstDll: %s
Inst[1]InstExe: %s
Inst[0]InstDir: %s
SoShutdown(soType:%d): ERROR_NOT_SUPPORTED
SoCreate(id:%d): ERROR_NOT_SUPPORTED
SoConnect(soType:%d): ERROR_NOT_SUPPORTED
SoSend(soType:%d): ERROR_NOT_SUPPORTED
SoRecv(soType:%d): ERROR_NOT_SUPPORTED
SoClose(soType:%d): ERROR_NOT_SUPPORTED
DoPacket() Unknow Cmd: %8.8X
recvfrom() = %d
CXSoRTP::OnRecv(pkt->wCd=%4.4X)...
CXSoRTP::OnRecvData()-EXPROID!!!
OnRecvSyncAck()-EXPROID!!!
ImpUserSession::0...
ImpUserSession::1...
ImpUserSession::2...
ImpUserSession::WaitForSingleObject(pid:%d, exit:%d(%8.8X)) ok
ImpUserSession::ImpUserCreateProcess(%d)
ImpUserService(%s) quit...
ImpUserService(%s) start...
ImpUserSession::session changed (%d:%d)...
ImpUserSessionUser::WaitForSingleObject(pid:%d, exit:%d(%8.8X)) ok
ImpUser(%s)...
x:%d,y:%d,flag:%d
Inject [%s] ERROR: %d
Inject [%s] OK!
XMgrScreenLog::LogProcEx()...
XMgrScreenLog::LogProcEx() quit...
CreateProcessAsUserW(pid:%d) ok
QueryPLP(): NOT FOUND PLP
OL:MUTEX: %s
```

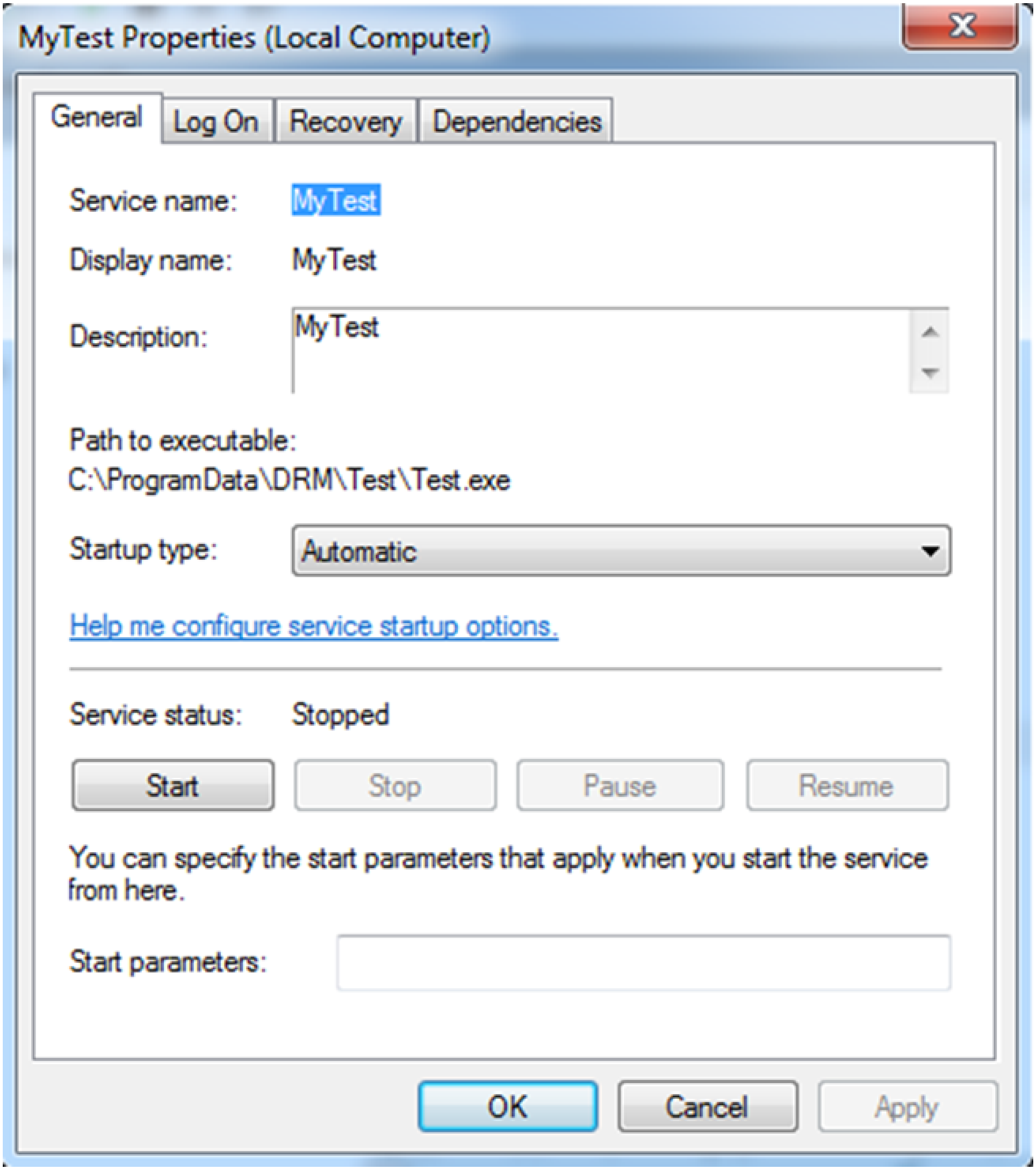
debugging messages

The processes created are as such:



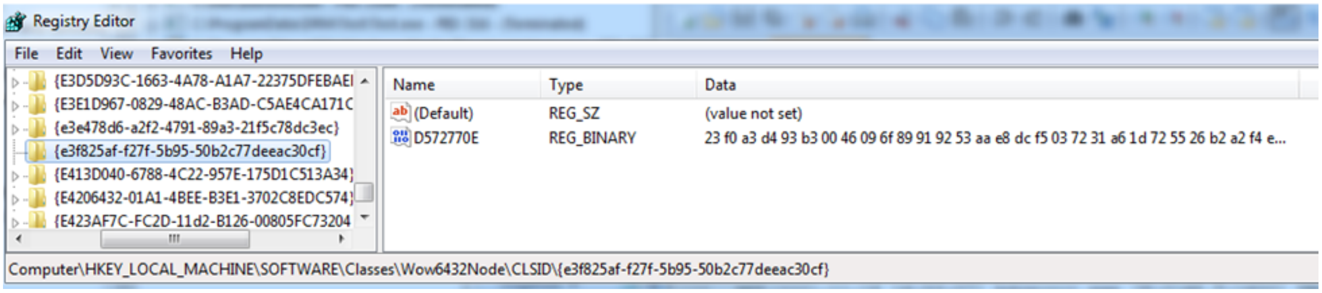
logged with APIMonitor

"Test.exe" is a copy of bdreinit.exe and is part of the persistency mechanism of the malware — The EXE and DLL are started via Services, and the encrypted payload is loaded from Registry. I supposed if service installation fails, then the malware shall persist through Run regkey.

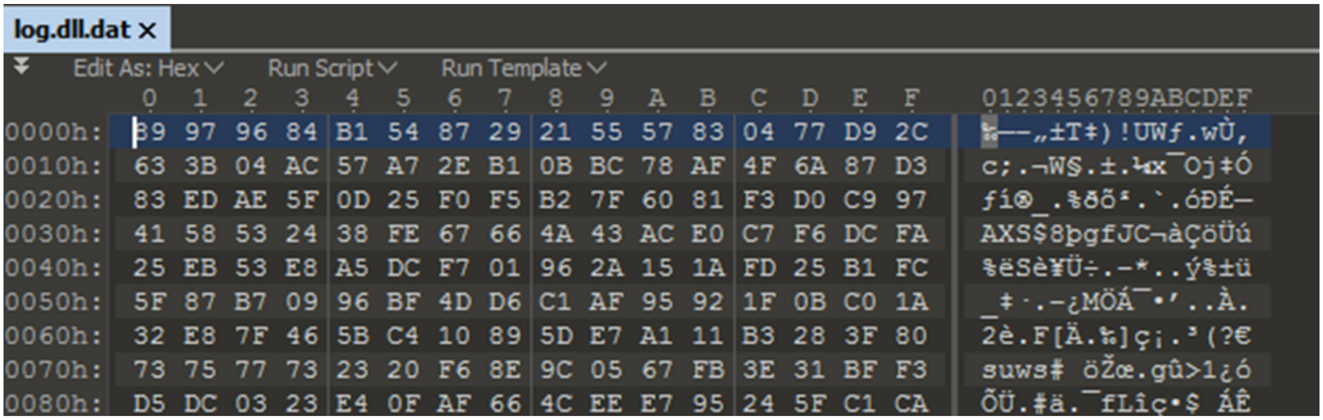


The EXE and DLL are copied to C:\ProgramData\DRM\Test (specified by configuration), while the DAT payload file is deleted after the first execution. For subsequent executions, the payload is read from Registry.

The payload is re-encrypted by the malware before it is being written into the Registry for persistency. For some reasons, the initialization value used in the re-encryption algorithm is the compilation timestamp of the malicious DLL file, while the initialization value used in the original encryption algorithm is the first 4 bytes within the dat file, hence the encrypted data seen in the registry differs from the log.dll.dat file.



encrypted payload in registry



encrypted payload file

Here is the python script I used to decrypt the payload file log.dll.dat:

SHADOWPAD-analysis/decrypt_payload_dat.py at main · asuna-amawaka/SHADOWPAD-analysis

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below…

github.com

And here is the script for decrypting the payload from registry:

SHADOWPAD-analysis/decrypt_payload_reg.py at main · asuna-amawaka/SHADOWPAD-analysis

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below…

github.com

The malware has keystroke logging capabilities, and the keystroke log file is encrypted and saved to a random-looking filepath in %PROGRAMDATA% like this:

> C:\ProgramData\MOO\JKISIQ\EWGSE\LECIIOGK

Decryption of the keystroke log file is done in the same manner as the configuration data (I'll talk about how to get this in awhile):

[SHADOWPAD-analysis/decrypt_keystroke_log.py at main · asuna-amawaka/SHADOWPAD-analysis]

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…

github.com

Anti-reverse engineering obfuscation

Now, here comes details on how I handled the reverse engineering. This is slightly harder than usual, because of the obfuscation technique used in the malware. The original malware binary is "shredded" into pieces, with 1 instruction per piece, and put back together with a "jumper" function as glue. Oh, and there are some junk "cmp" followed by "jb" instructions just to make your eyes hurt.

Let's see what it looks like in IDA.

Up till this point at address 0x100011D6, everything is normal.

disassembly at beginning of malware logic

Then came this jmp:

```
.text:10001738
.text:10001738
.text:10001738                          ; Attributes: thunk
.text:10001738
.text:10001738                          sub_10001738 proc near
.text:10001738 E9 8F 66 00 00           jmp     loc_10007DCC
.text:10001738                          sub_10001738 endp
.text:10001738
```

Followed by another call:

```
.text:10007DCC                 ; ---------------------------------------------------------------
.text:10007DCC
.text:10007DCC                 loc_10007DCC:                           ; CODE XREF: sub_10001738↑j
.text:10007DCC E8 94 C6 FF FF          call    jumper_10004465
.text:10007DCC                 ; ---------------------------------------------------------------
.text:10007DD1 46                      db  46h ; F
.text:10007DD2 5A                      db  5Ah ; Z
.text:10007DD3 00                      db    0
.text:10007DD4 00                      db    0
```

If we follow the instructions starting from 0x10004465, you will see something like this:

```
0x10004465 xchg dword ptr ss:[esp], eax
0x10004468 jb 0x10012C6F
0x1000446E nop
0x1000446F xchg dx, dx
0x10004472 jae 0x10012C6F
0x10012C6F pushfd
0x10012C70 je 0x1000F507
0x10012C76 jne 0x1000F507
0x1000F507 add eax, dword ptr ds:[eax]
0x1000F509 jno 0x1000AD76
0x1000AD76 popfd
0x1000AD77 jae 0x10012C5E
0x10012C5E xchg dword ptr ss:[esp], eax
0x10012C61 jbe 0x10006EC3
0x10012C67 ja 0x10006EC3
0x10006EC3 ret
```

within "jumper" function

It looks terribly complicated, but all it does is to read the next dword after the call and add it to original intended return address.

```
.text:10007DCC                 ; ---------------------------------------------------------------
.text:10007DCC
.text:10007DCC                 loc_10007DCC:                           ; CODE XREF: sub_10001738↑j
.text:10007DCC E8 94 C6 FF FF          call    jumper_10004465 ; 0x1000d817
.text:10007DCC                 ; ---------------------------------------------------------------
.text:10007DD1 46                      db  46h ; F
.text:10007DD2 5A                      db  5Ah ; Z
.text:10007DD3 00                      db    0           0x10007DD1 + 0x5A46 = 0x1000D817
.text:10007DD4 00                      db    0
```

illustrate jumper (1)

And these "bits and pieces" of instructions occur from here onwards, throughout the whole malware. The "real" instruction that is part of the malware's logic is the single instruction before the call to jumper. Trying to recover these instructions makes me feel like I am picking up the pieces from the shredding machine and gluing them back.

```
.text:1000D817                          ; ---------------------------------------------------------------
.text:1000D817 55                                       push    ebp
.text:1000D818 E8 48 6C FF FF                           call    jumper_10004465 ; 0x10012766
.text:1000D818                          ; ---------------------------------------------------------------
.text:1000D81D 49                                       db  49h ; I
.text:1000D81E 4F                                       db  4Fh ; O
.text:1000D81F 00                                       db    0
.text:1000D820 00                                       db    0
```

0x1000D81D + 0x4F49 = **0x10012766**

illustrate jumper (2)

```
.text:10012766                          ; ---------------------------------------------------------------
.text:10012766 8B EC                                    mov     ebp, esp
.text:10012768 E8 F8 1C FF FF                           call    jumper_10004465 ; 0x1000e876
.text:10012768                          ; ---------------------------------------------------------------
.text:1001276D 09                                       db    9
.text:1001276E C1                                       db 0C1h ; Á
.text:1001276F FF                                       db 0FFh ; ÿ
.text:10012770 FF                                       db 0FFh ; ÿ
```

0x1001276D + 0xFFFFC109 = **0x1000E876**

illustrate jumper (3)

```
.text:1000E876                          ; ---------------------------------------------------------------
.text:1000E876 81 FC 54 C9 00 00                        cmp     esp, 0C954h
.text:1000E87C E8 E4 5B FF FF                           call    jumper_10004465 ; 0x1000918f
.text:1000E87C                          ; ---------------------------------------------------------------
.text:1000E881 0E                                       db  0Eh
.text:1000E882 A9                                       db 0A9h ; ©
.text:1000E883 FF                                       db 0FFh ; ÿ
.text:1000E884 FF                                       db 0FFh ; ÿ
```

illustrate jumper (4)

```
.text:1000918F                          ; ---------------------------------------------------------------
.text:1000918F 0F 82 63 3B 00 00                        jb      loc_1000CCF8
.text:10009195 E8 CB B2 FF FF                           call    jumper_10004465 ; 0x1001153c
.text:10009195                          ; ---------------------------------------------------------------
.text:1000919A A2                                       db 0A2h ; ¢
.text:1000919B 83                                       db  83h ; ƒ
.text:1000919C 00                                       db    0
.text:1000919D 00                                       db    0
```

illustrate jumper (5)

```
.text:1001153C                          ; ---------------------------------------------------------------
.text:1001153C 83 E4 F8                                 and     esp, 0FFFFFFF8h
.text:1001153F E8 21 2F FF FF                           call    jumper_10004465 ; 0x10007894
.text:1001153F                          ; ---------------------------------------------------------------
.text:10011544 50                                       db  50h ; P
.text:10011545 63                                       db  63h ; c
.text:10011546 FF                                       db 0FFh ; ÿ
.text:10011547 FF                                       db 0FFh ; ÿ
```

illustrate jumper (6)

After doing away with the jumper calls, here’s a snippet of recovered “shreds” of instructions:

```
.text:1000D817 55                                 push    ebp
.text:10012766 8B EC                              mov     ebp, esp
.text:1000E876 81 FC 54 C9 00 00                  cmp     esp, 0C954h
.text:1000918F 0F 82 63 3B 00 00                  jb      loc_1000CCF8
.text:1001153C 83 E4 F8                           and     esp, 0FFFFFFF8h
.text:10007894 81 FC A0 F6 00 00                  cmp     esp, 0F6A0h
.text:10011444 0F 82 E1 EC FF FF                  jb      loc_1001012B
.text:10006591 81 EC 34 04 00 00                  sub     esp, 434h
.text:10007433 8B 4D 08                           mov     ecx, [ebp+8]
.text:10009284 53                                 push    ebx
.text:100067E8 56                                 push    esi
.text:1000D612 57                                 push    edi
.text:10002CD0 8D 44 24 28                        lea     eax, [esp+28h]
.text:100115C5 81 FC 6A 98 00 00                  cmp     esp, 986Ah
.text:100087BA 0F 82 A0 3D 00 00                  jb      loc_1000C560
.text:1000C358 33 F6                              xor     esi, esi
.text:100130DC 50                                 push    eax
.text:100037A9 51                                 push    ecx
.text:10002CEB 89 74 24 30                        mov     [esp+30h], esi
.text:10015CDD 89 74 24 34                        mov     [esp+34h], esi
.text:1000999E 89 74 24 3C                        mov     [esp+3Ch], esi
.text:1000884B 89 74 24 38                        mov     [esp+38h], esi
.text:10005A84 E8 A4 83 00 00                     call    loc_1000DE2D
…
```

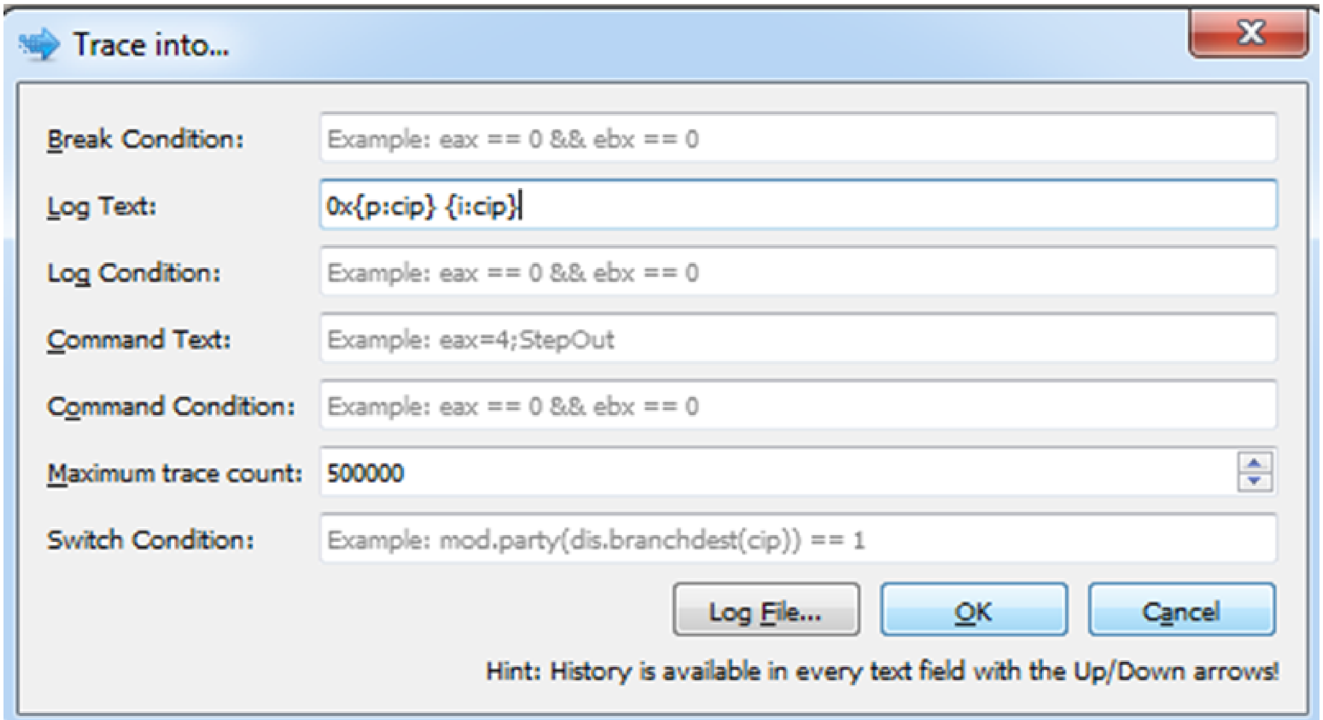
I greyed out the junk “cmp” and “jb” instructions. Up till this point, the process of recovery is very manual, with abit of help from this IDAPython script I wrote:

SHADOWPAD-analysis/ida_get_next_instr.py at main · asuna-amawaka/SHADOWPAD-analysis

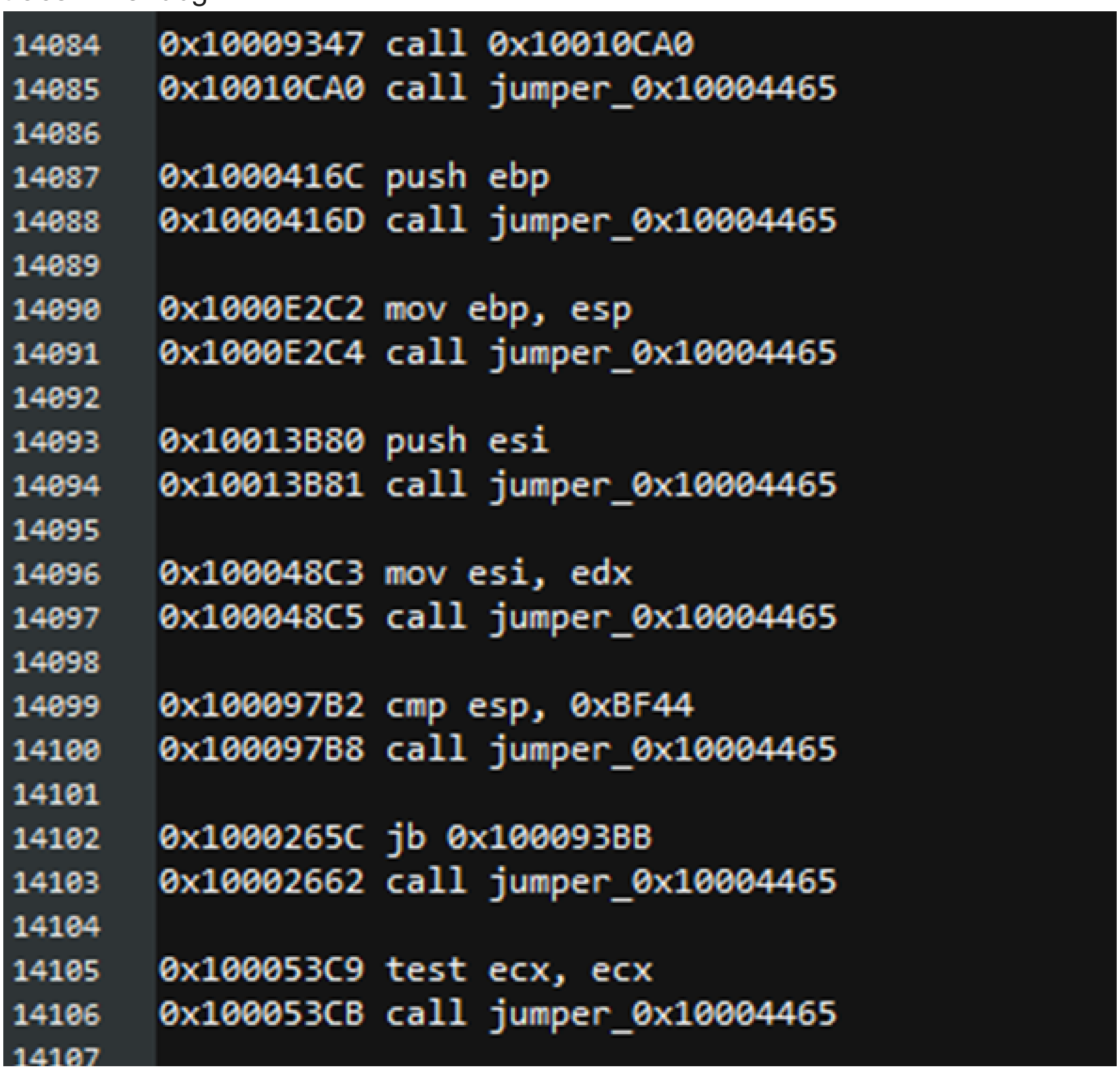
You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…

github.com

The python script is not perfect — at some point it will fail to work as intended, and I will have to apply the script again at the position where it failed. Furthermore, this manner of “advancing” through the disassembly in IDA didn’t feel very efficient. As it turns out, using the “trace” feature in the debugger produced the expected results with ease.



trace in x32dbg



```
0x10009347 call 0x10010CA0
0x10010CA0 call jumper_0x10004465

0x1000416C push ebp
0x1000416D call jumper_0x10004465

0x1000E2C2 mov ebp, esp
0x1000E2C4 call jumper_0x10004465

0x10013B80 push esi
0x10013B81 call jumper_0x10004465

0x100048C3 mov esi, edx
0x100048C5 call jumper_0x10004465

0x100097B2 cmp esp, 0xBF44
0x100097B8 call jumper_0x10004465

0x1000265C jb 0x100093BB
0x10002662 call jumper_0x10004465

0x100053C9 test ecx, ecx
0x100053CB call jumper_0x10004465
```

```
0x100024D4 jle 0x10004C22
0x100024DA call jumper_0x10004465

0x10012309 push ebx
0x1001230A call jumper_0x10004465

0x100138C2 push edi
0x100138C3 call jumper_0x10004465

0x1000621B mov edi, dword ptr ss:[ebp+0x8]
0x1000621E call jumper_0x10004465

0x1000F8A1 cmp esp, 0x16C
0x1000F8A7 call jumper_0x10004465

0x10009EA8 jb 0x1000C880
0x10009EAE call jumper_0x10004465
```

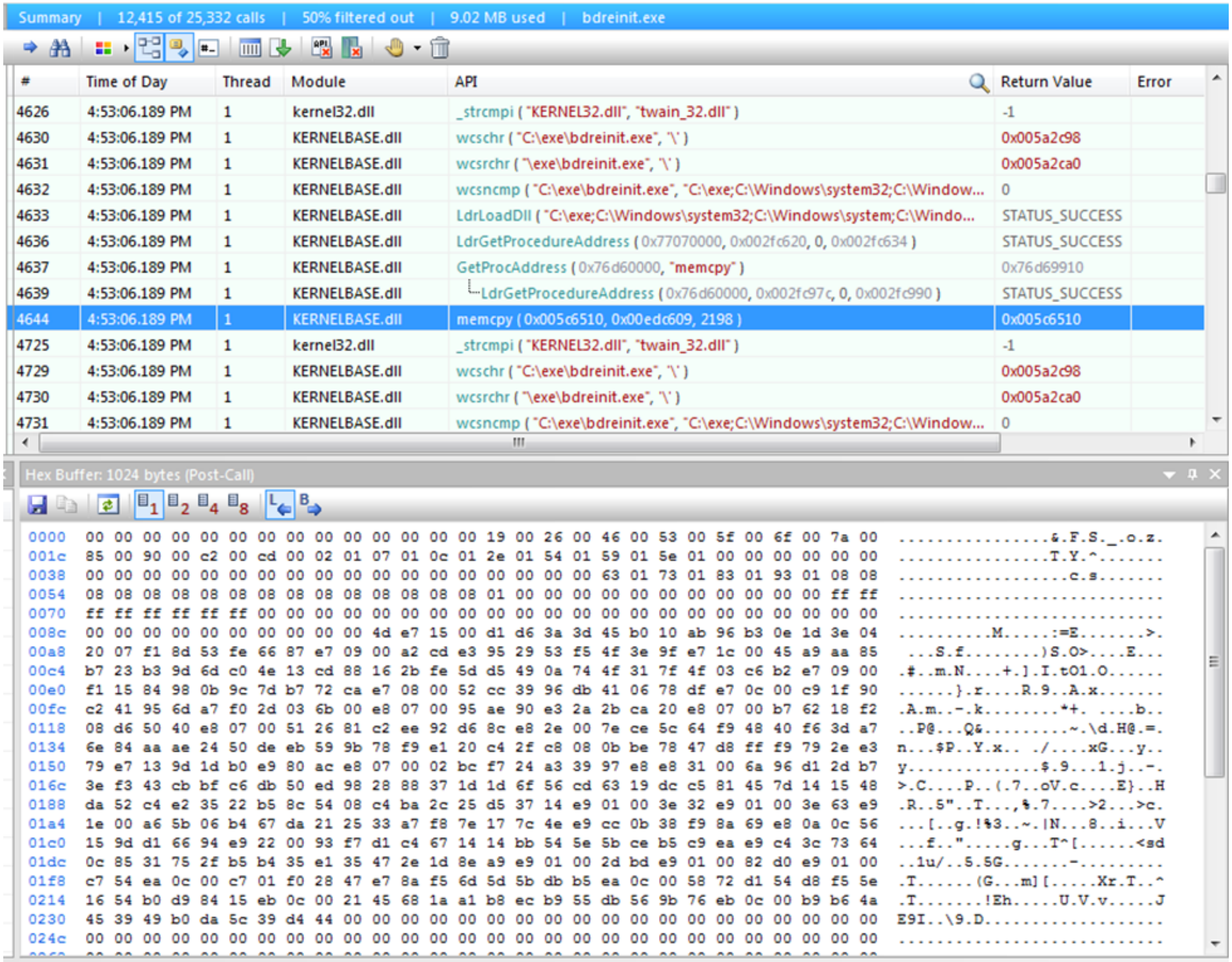
traced log

After some cleaning up, here’s the code logic that decrypts the payload from registry, for a taste of what we can see after doing away with the jumper calls and junk cmp-jbs.

```
0x1001082D mov edi, dword ptr ss:[ebp+0x8]
0x100075B6 call 0x10010247
|
->     0x10013ECE push esi
       0x10003506 test edi, edi
       0x10015384 jne 0x10013F36
       0x100063D1 movzx eax, word ptr ds:[edi]
       0x1000248D xor eax, 0xE3798186
       0x1000E4BB cmp eax, 0xE379DBCB
       0x10008B35 je 0x10011115
       0x10006E89 mov esi, dword ptr ds:[edi+0x3C]
       0x1000A63C mov ecx, dword ptr ds:[edi+esi*1]
       0x1001027C xor ecx, 0xCD5D5126
       0x1000A9CF cmp ecx, 0xCD5D1476
       0x1000C57B je 0x1000D957
       0x10010E32 mov eax, dword ptr ds:[edi+esi*1+0x8]
       0x10013F90 pop esi
       0x1000583D ret
0x10012F94 mov edi, dword ptr ss:[ebp-0x8]
0x10009650 mov esi, dword ptr ss:[ebp-0x14]
0x10013AA4 xor eax, 0xA7847046
0x1000CA8D push edi
0x1000A5F1 mov edx, edi
0x1000DE14 mov ecx, esi
0x10009347 call 0x10010CA0
|
->     0x1000416C push ebp
       0x1000E2C2 mov ebp, esp
       0x10013B80 push esi
       0x100048C3 mov esi, edx
       0x100053C9 test ecx, ecx
       0x100024D4 jle 0x10004C22
       0x10012309 push ebx
       0x100138C2 push edi
       0x1000621B mov edi, dword ptr ss:[ebp+0x8]
       0x10007D86 sub edi, esi
       ->     0x1000B41E mov edx, eax
       |      0x10015F00 add edx, edx
       |      0x10006429 lea eax, ds:[eax+edx*8+0x107E666D]
       |      0x1000667B mov edx, eax
       |      0x10007AB0 shr edx, 0x18
       |      0x10005A26 mov ebx, eax
       |      0x10001C9D shr ebx, 0x10
       |      0x1000F523 add dl, bl
       |      0x10008B8E mov ebx, eax
       |      0x1000E383 shr ebx, 0x8
       |      0x100027F9 add dl, bl
       |      0x1000566A add dl, al
       |      0x1000FDAC xor dl, byte ptr ds:[esi+edi*1]
       |      0x10013BB5 inc esi
       |      0x10012E95 dec ecx
       |      0x10004594 mov byte ptr ds:[esi-0x1], dl
       --     0x10005D43 jne 0x10009E6E
       ...
```
traced instructions for decrypting payload from registry

Malware Configuration

In order to recover the configuration data, it helps to know what it looks like from older variants of ShadowPad (without the shredding obfuscation) — so that we can recognize it in memory. I used APIMonitor to look out for memory copies, because I knew that was what ShadowPad will do with its configuration.



APIMonitor log of memcpy

One visual characteristic of the configuration data is that it will start and end with many zeroes, and it is not very long (the part that looks like encrypted data is approx. ~400 bytes long). In this particular sample, the memory size expected for the configuration is 2198 bytes long. This can perhaps be a helping value to look out for the memcpy call dealing with the configuration.

Decryption of the configuration data uses the exact same algorithm as what is used to decrypt the keystroke log file. Earlier I shared the standalone python script used to decrypt the keystroke log file. Here's the IDAPython version to handle the configuration data in IDA:

SHADOWPAD-analysis/ida_decrypt_config.py at main · asuna-amawaka/SHADOWPAD-analysis

[You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…]

github.com

And here's the decrypted configuration:

```
8/22/2021 9:42:57 PM
exchange
%ALLUSERSPROFILE%\DRM\Test\
Test.exe
log.dll
log.dll.dat
MyTest
MyTest
MyTest
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
MyTest
%ProgramFiles%\Windows Media Player\wmplayer.exe
%windir%\system32\svchost.exe
TCP://ti0wddsnv[.]wikimedia[.]vip:443
SOCKS4
SOCKS4
SOCKS5
SOCKS5
```

ShadowPad configuration decrypted

Hmmm~ it looks like someone was testing his BEE some 3 months ago, based on that timestamp in the configuration.

Well then, that is all I have. Come chat with me on Twitter if you have any idea how I can automate this analysis; I seem to have done most stuff in the painful way :|

Network IOC:

ti0wddsnv[.]wikimedia[.]vip:443

Host IOCs:

log.dll — SHA256:
8D1A5381492FE175C3C8263B6B81FD99AACE9E2506881903D502336A55352FEF

log.dll.dat — SHA256:
0371FC2A7CC73665971335FC23F38DF2C82558961AD9FC2E984648C9415D8C4E

Scripts mentioned in this post are here:

GitHub - asuna-amawaka/SHADOWPAD-analysis

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…

github.com

[1] https://securelist.com/apt-trends-report-q3-2021/104708/

[2] Higaisa or Winnti? APT41 backdoors, old and new - PTSecurity (https://ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/higaisa-or-winnti-apt-41-backdoors-old-and-new)

[3] Evolution after prosecution: Psychedelic APT41 - TeamT5 at VB2021 (https://vblocalhost.com/conference/presentations/evolution-after-prosecution-psychedelic-apt41)

~~

Asuna | https://twitter.com/AsunaAmawaka