

It's a BEE! It's a... no, it's ShadowPad.

By asuna amawaka

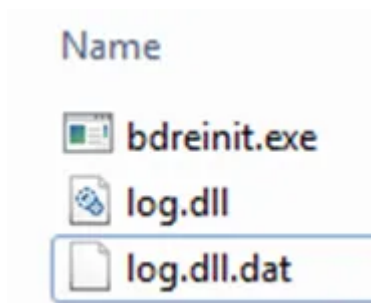
Published: 2021-11-19 · Archived: 2026-04-05 18:36:41 UTC



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:

8D1A5381492FE175C3C8263B6B81FD99AAACE9E2506881903D502336A55352FEF

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:

Press enter or click to view image in full size

```
DLL Loaded: 5FAB0000 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-5b95-50b2c77deaac30cf}: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(MInstall.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(Svc.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\System64\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\wmpplayer.exe) OK!"
DebugString: "BEE(2021-11-18 14:46:22) bdreinit.exe [P:1432,T:2684] Inject [C:\Program Files (x86)\Windows Media Player\wmpplayer.exe] OK!"
Process stopped with exit code 0x0
Saving database to C:\Program Files\x64dbg\release\x32\db\bdreinit.exe.dd32 Cms
Debugging stopped!
```

logged by x32dbg

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

Press enter or click to view image in full size

```
bee.log
1 BEE(2021-11-18 14:46:17) bdreinit.exe [P:1432,T:2684] ScLoad:path_exe: C:\exe\bdreinit.exe
2 BEE(2021-11-18 14:46:17) bdreinit.exe [P:1432,T:2684] ScLoad:path_dll: C:\exe\log.dll
3 BEE(2021-11-18 14:46:17) bdreinit.exe [P:1432,T:2684] ScLoad:path_img: C:\exe\log.dll.dat
4 BEE(2021-11-18 14:46:17) bdreinit.exe [P:1432,T:2684] ScLoad:KEY(SOFTWARE\Classes\CLSID\{e3f825af-f27f-5b95-50b2c77deaac30cf}:D572770E)
5 BEE(2021-11-18 14:46:19) bdreinit.exe [P:1432,T:2684] ScLoad:ScQueryFromFile() OK
6 BEE(2021-11-18 14:46:19) bdreinit.exe [P:1432,T:2684] ScLoad:shellcode()...
7 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[0]InstDir: %ALLUSERSPROFILE%\DRM\Test\
8 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[1]InstExe: Test.exe
9 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[2]InstDll: log.dll
10 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[4]SvcName: MyTest
11 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[5]SvcDisp: MyTest
12 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[6]SvcDesc: MyTest
13 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[7]KeyPath: SOFTWARE\Microsoft\Windows\CurrentVersion\Run
14 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] Inst[8]KeyName: MyTest
15 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)
16 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] DBG_ASSERT(MInstall.cpp:484)[32]:The process cannot access the file because it is being used by another process.
17 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] DBG_ASSERT(Svc.cpp:225)[32]:The process cannot access the file because it is being used by another process.
18 BEE(2021-11-18 14:46:21) bdreinit.exe [P:1432,T:2684] SID: S-1-5-21-3846458015-2120299329-620902809-1000
19 BEE(2021-11-18 14:46:22) bdreinit.exe [P:1432,T:2684] CreateProcessAsUserW(pid:2748) ok
20 BEE(2021-11-18 14:46:22) bdreinit.exe [P:1432,T:2684] CXInject(pid:2748,path:C:\Program Files (x86)\Windows Media Player\wmpplayer.exe) OK!
21 BEE(2021-11-18 14:46:22) bdreinit.exe [P:1432,T:2684] Inject [C:\Program Files (x86)\Windows Media Player\wmpplayer.exe] OK!
22 BEE(2021-11-18 14:46:24) wmpplayer.exe [P:2748,T:1268] OL:MUTEX: Global\PMIIOGCUOYOVGAYCQVMKXK
23 BEE(2021-11-18 14:46:30) wmpplayer.exe [P:2748,T:1268] SID: S-1-5-21-3846458015-2120299329-620902809-1000
24 BEE(2021-11-18 14:46:30) wmpplayer.exe [P:2748,T:1652] CQWgScreenDogs:logProcEx()...
25 BEE(2021-11-18 14:46:34) wmpplayer.exe [P:2748,T:1268] Online(0) = TCP://ti0addsnv.wikimedia.vip:443
26 BEE(2021-11-18 14:46:35) wmpplayer.exe [P:2748,T:1268] OnlineX( [ti0addsnv.wikimedia.vip] : 443 : 200, [ ] : 0 : 0, p: 0)
27
```

bee.log

Press enter or click to view image in full size

```

bee.log
48 BEE[2021-11-18 16:23:32] wmpayer.exe [P:2772,T:2320] OL:MUTEX: Global\PMIIOGCCUOYOGWYCGMYWCKKK
49 BEE[2021-11-18 16:23:34] wmpayer.exe [P:2772,T:2320] SID: 5-1-5-21-3846458015-2120299329-620902809-1000
50 BEE[2021-11-18 16:23:34] wmpayer.exe [P:2772,T:2988] CXMgrScreenLog::LogProcEx(...)
51 BEE[2021-11-18 16:23:38] wmpayer.exe [P:2772,T:2320] Online(0) = TCP://ti0wddsnv.wikimedia.vip:443
52 BEE[2021-11-18 16:23:38] wmpayer.exe [P:2772,T:2320] OnlineEx( [ti0wddsnv.wikimedia.vip] : 443 : 200, [ ] : 0 : 0, p: 0)
53 BEE[2021-11-18 16:23:39] wmpayer.exe [P:2772,T:2320] OnlineEx() Q4=10054
54 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(1) =
55 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
56 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(2) =
57 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
58 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(3) =
59 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
60 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(4) = NULL
61 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(5) = NULL
62 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(6) = NULL
63 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(7) = NULL
64 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(8) = NULL
65 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(9) = NULL
66 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(10) = NULL
67 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(11) = NULL
68 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(12) = NULL
69 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(13) = NULL
70 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(14) = NULL
71 BEE[2021-11-18 16:23:40] wmpayer.exe [P:2772,T:2320] Online(15) = NULL
72 BEE[2021-11-18 16:23:42] wmpayer.exe [P:2772,T:2320] Online(0) = TCP://ti0wddsnv.wikimedia.vip:443
73 BEE[2021-11-18 16:23:42] wmpayer.exe [P:2772,T:2320] OnlineEx( [ti0wddsnv.wikimedia.vip] : 443 : 200, [ ] : 0 : 0, p: 0)
74 BEE[2021-11-18 16:23:42] wmpayer.exe [P:2772,T:2320] OnlineEx() Q4=10054
75 BEE[2021-11-18 16:23:43] wmpayer.exe [P:2772,T:2320] Online(1) =
76 BEE[2021-11-18 16:23:43] wmpayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
77 BEE[2021-11-18 16:23:43] wmpayer.exe [P:2772,T:2320] Online(2) =
78 BEE[2021-11-18 16:23:43] wmpayer.exe [P:2772,T:2320] DBG_ASSERT(XOnline.cpp:106)[12006]:*
79 BEE[2021-11-18 16:23:43] wmpayer.exe [P:2772,T:2320] Online(3) =
80 BEE[2021-11-18 16:23:43] wmpayer.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`:

Press enter or click to view image in full size

```
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

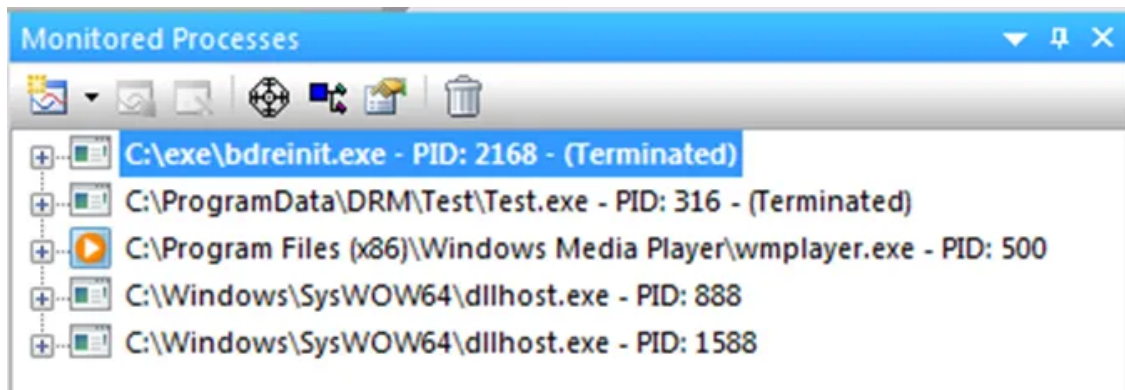
And here's the list of debugging messages found in memory:

[Press enter or click to view image in full size](#)

```
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
```

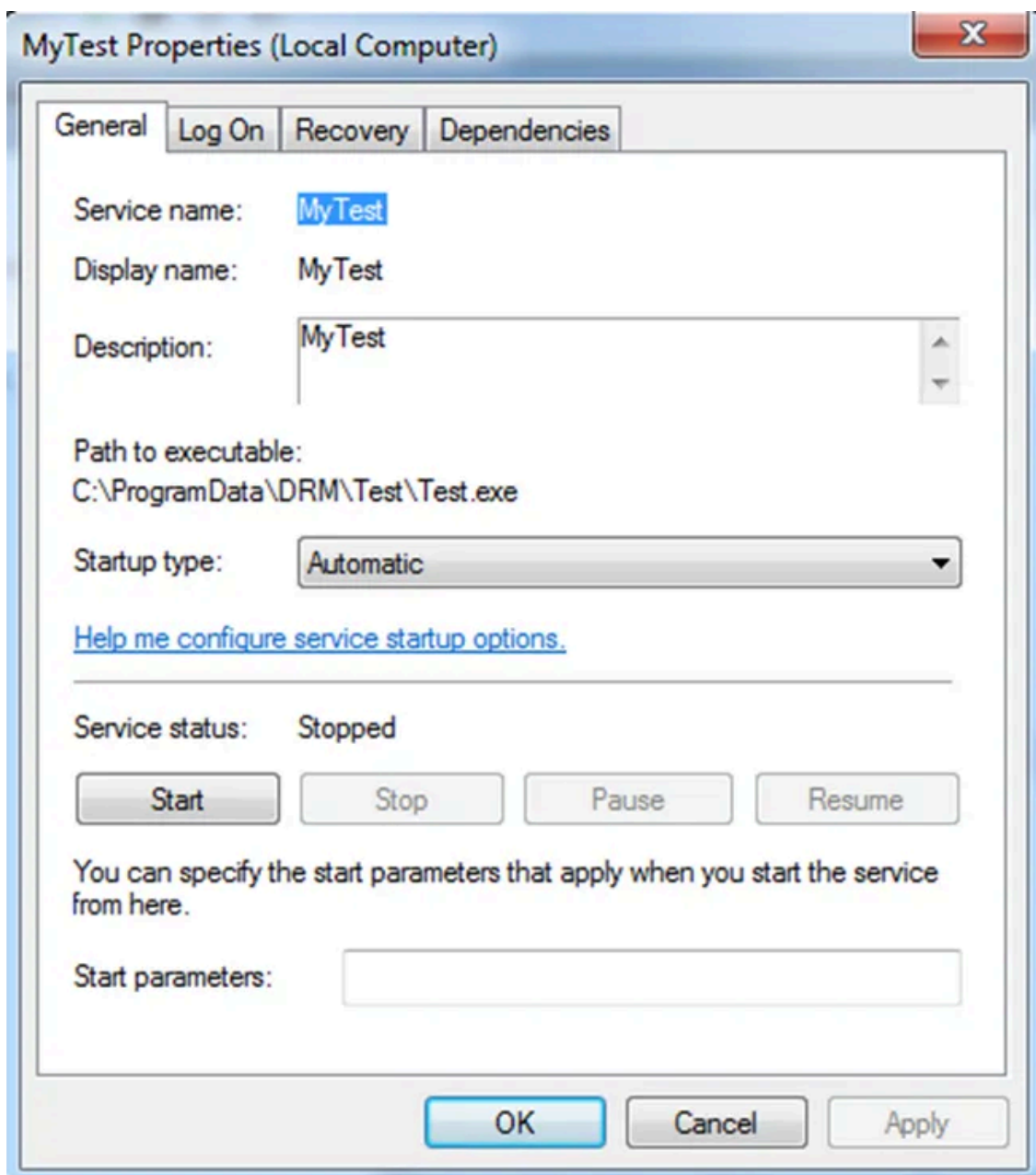
debugging messages

The processes created are as such:



logged with APIMonitor

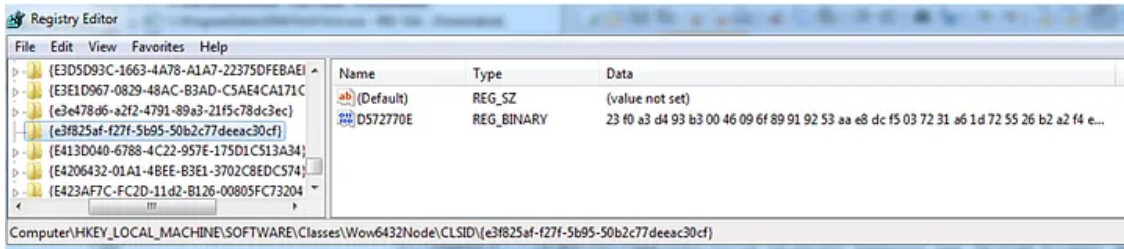
“Test.exe” is a copy of bdreinit.exe and is part of the persistency mechanism of the malware — The EXE and DLL are started via Services, and the encrypted payload is loaded from Registry. I supposed if service installation fails, then the malware shall persist through Run regkey.



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.

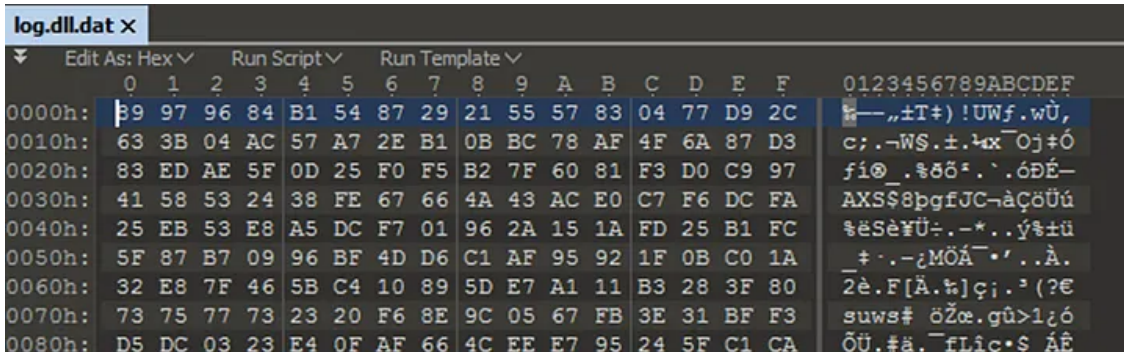
The payload is re-encrypted by the malware before it is being written into the Registry for persistency. For some reasons, the initialization value used in the re-encryption algorithm is the compilation timestamp of the malicious DLL file, while the initialization value used in the original encryption 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.

Press enter or click to view image in full size



encrypted payload in registry

Press enter or click to view image in full size



encrypted payload file

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

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

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\LECHIOGK
```

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):

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.

Get asuna amawaka's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

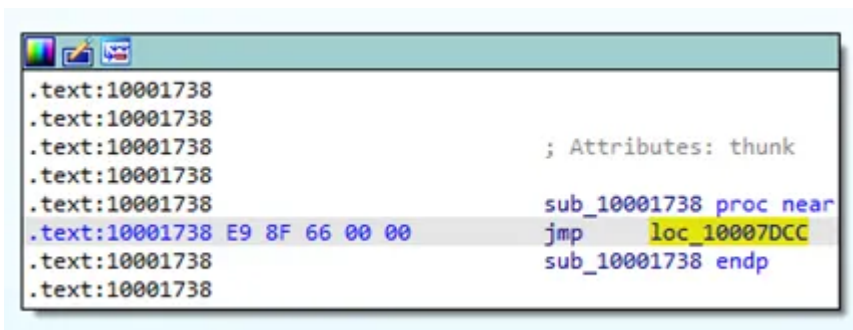
Up till this point at address 0x100011D6, everything is normal.

Press enter or click to view image in full size

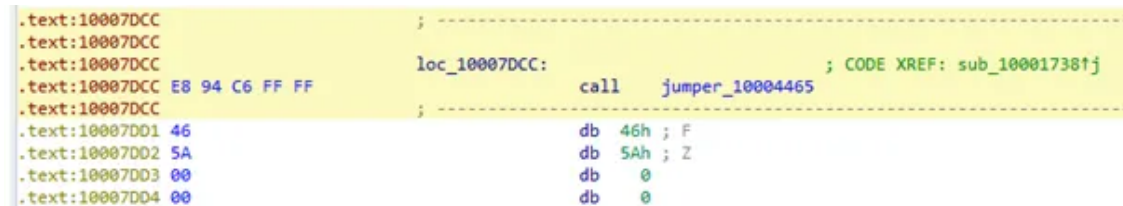


disassembly at beginning of malware logic

Then came this jmp:



Followed by another call:



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

Press enter or click to view image in full size

```

0x10004465 xchg dword ptr ss:[esp], eax
0x10004468 jnb 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.

Press enter or click to view image in full size

```

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

0x10007DD1 + 0x5A46 = 0x1000D817

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.

Press enter or click to view image in full size

```

.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)

Press enter or click to view image in full size

```

.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 ; Ā
.text:1001276F FF          db 0FFh ; ŷ
.text:10012770 FF          db 0FFh ; ŷ 0x1001276D + 0xFFFFC109 = 0x1000E876

```

illustrate jumper (3)

Press enter or click to view image in full size

```

.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 ; ŷ
.text:1000E884 FF          db 0FFh ; ŷ

```

illustrate jumper (4)

Press enter or click to view image in full size

```

.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 ; f
.text:1000919C 00          db 0
.text:1000919D 00          db 0

```

illustrate jumper (5)

Press enter or click to view image in full size

```

.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 ; ŷ
.text:10011547 FF          db 0FFh ; ŷ

```

illustrate jumper (6)

After doing away with the jumper calls, here's a snippet of recovered "shreds" of instructions:

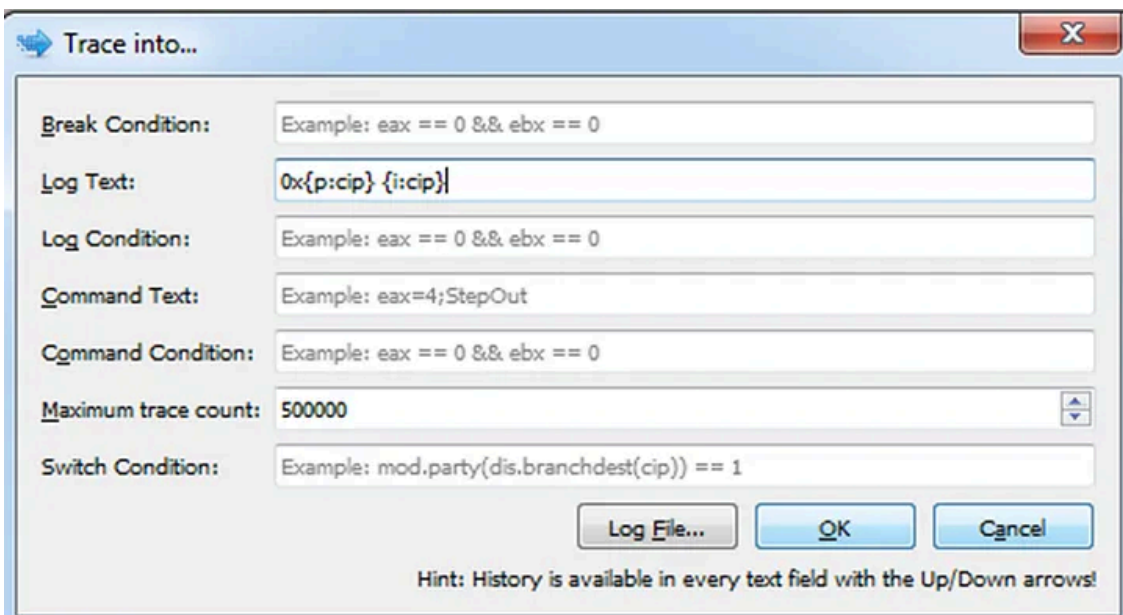
Press enter or click to view image in full size

```
.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:

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.

Press enter or click to view image in full size



trace in x32dbg

Press enter or click to view image in full size

```
14084 0x10009347 call 0x10010CA0
14085 0x10010CA0 call jumper_0x10004465
14086
14087 0x1000416C push ebp
14088 0x1000416D call jumper_0x10004465
14089
14090 0x1000E2C2 mov ebp, esp
14091 0x1000E2C4 call jumper_0x10004465
14092
14093 0x10013B80 push esi
14094 0x10013B81 call jumper_0x10004465
14095
14096 0x100048C3 mov esi, edx
14097 0x100048C5 call jumper_0x10004465
14098
14099 0x100097B2 cmp esp, 0xBF44
14100 0x100097B8 call jumper_0x10004465
14101
14102 0x1000265C jb 0x100093BB
14103 0x10002662 call jumper_0x10004465
14104
14105 0x100053C9 test ecx, ecx
14106 0x100053CB call jumper_0x10004465
14107
14108 0x100024D4 jle 0x10004C22
14109 0x100024DA call jumper_0x10004465
14110
14111 0x10012309 push ebx
14112 0x1001230A call jumper_0x10004465
14113
14114 0x100138C2 push edi
14115 0x100138C3 call jumper_0x10004465
14116
14117 0x1000621B mov edi, dword ptr ss:[ebp+0x8]
14118 0x1000621E call jumper_0x10004465
14119
14120 0x1000F8A1 cmp esp, 0x16C
14121 0x1000F8A7 call jumper_0x10004465
14122
14123 0x10009EA8 jb 0x1000C880
14124 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.

Press enter or click to view image in full size

```
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, 0xE379DBC8
    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
...

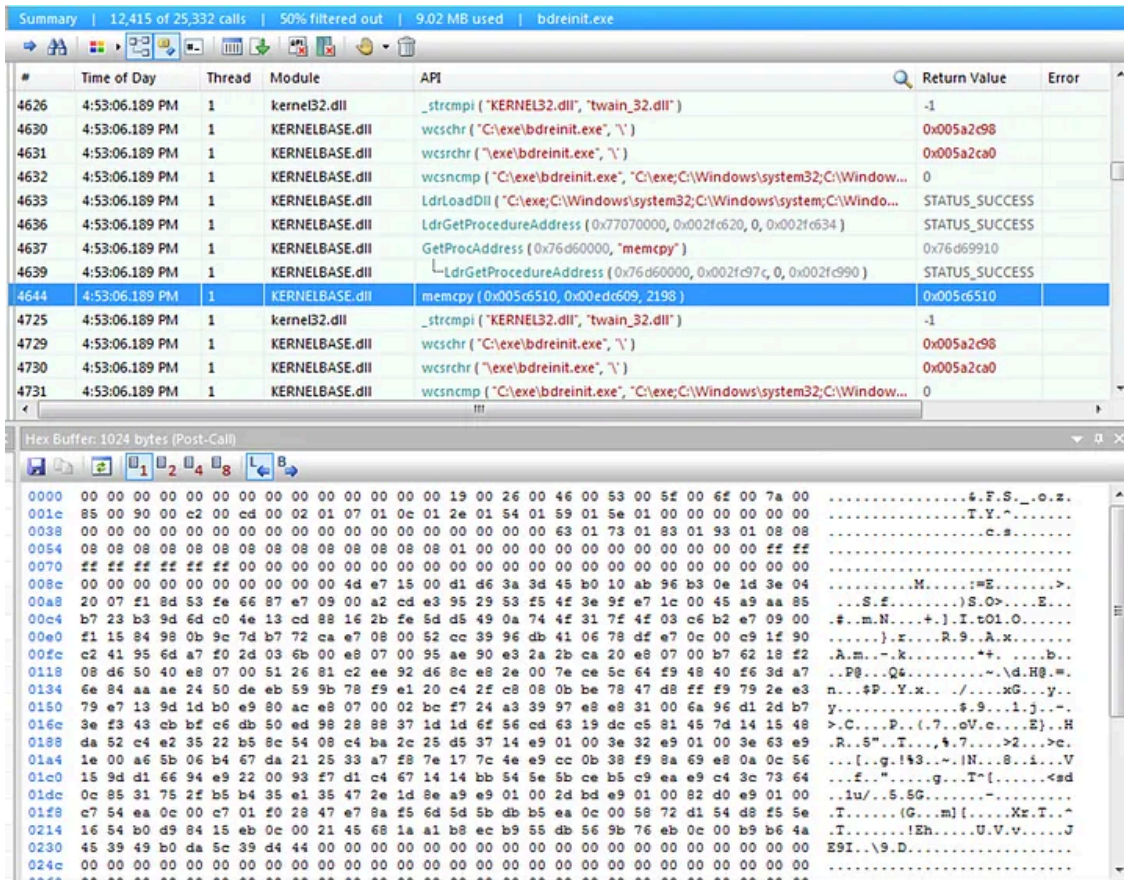
```

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.

Press enter or click to view image in full size



APIMonitor log of memcopy

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 memcopy 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:

And here's the decrypted configuration:

Press enter or click to view image in full size

```
8/22/2021 9:42:57 PM
exchange
%ALLUSERSPROFILE%\DRM\Test\
Test.exe
log.dll
log.dll.dat
MyTest
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 :|

Source: <https://medium.com/insomniacs/its-a-bee-it-s-a-no-it-s-shadowpad-aff6a970a1c2>