Noah Cosamano
Trojan Forensic Report
May 2nd, 2026

Trojan Forensic Report

Executive Summary:

This forensic report highlights the analysis of a 32-bit trojan horse that is frequently identified by security vendors as gh0stRat, or labels such as Injuke, SilverFox, and SwimSnake. The primary executable is relatively large at 92.2MB and not packed. During execution, the malware uses evasion techniques by replicating itself into the systems temporary folder, _ir_sf_temp. It is designed for data exfiltration by using Advapi32.dll to capture sensitive account information including security identifiers, machine idle times, usernames, computer names, and connection types. The data gathered is formatted into strings such as __IRSID:, __IRAOFF:, and __IRAFN:, and exfiltrated to a command-and-control server. The program maintains a connection through a heartbeat loop until it receives a killswitch (0x200) from the server. Also, the malware contains a debug flag (/~DBG) that allows the author to test the malware without automatic self-deletion, which helps further aid development.

Static Analysis

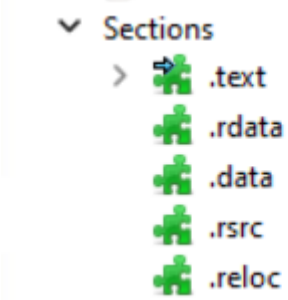
Primary File

File Hashes (Found using PEbear):

SHA256	003734348d05ac8087c497c214bd871355b168e41c9ee8d5aa5fcfcdbe6c8a89
SHA1	6e9016472c82fb65ce9bd2162557ec3aff529df5
MD5	edc2659fa344a270e8cd6cfee9782e49
Imp	1ff847646487d56f85778df99ff3728a

Architecture: Intel 386 (32-bit)

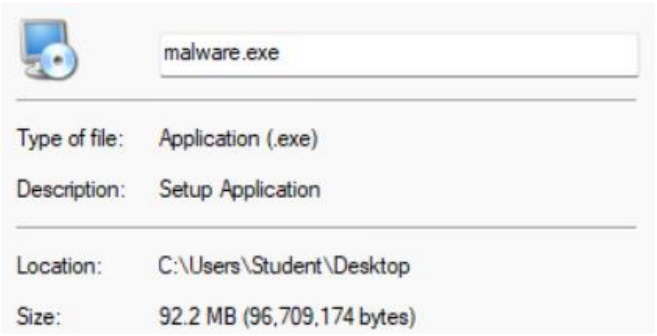
Offset	Name	Value	Meaning
DC	Machine	14c	Intel 386
DE	Sections Count	5	5
E0	Time Date Stamp	4fda0e4a	Thursday, 14.06.2012 16:16:10 UTC
E4	Ptr to Symbol Table	0	0
E8	Num. of Symbols	0	0
EC	Size of OptionalHeader	e0	224
EE	Characteristics	102	
		2	File is executable (i.e. no unresolved external references).
		100	32 bit word machine.



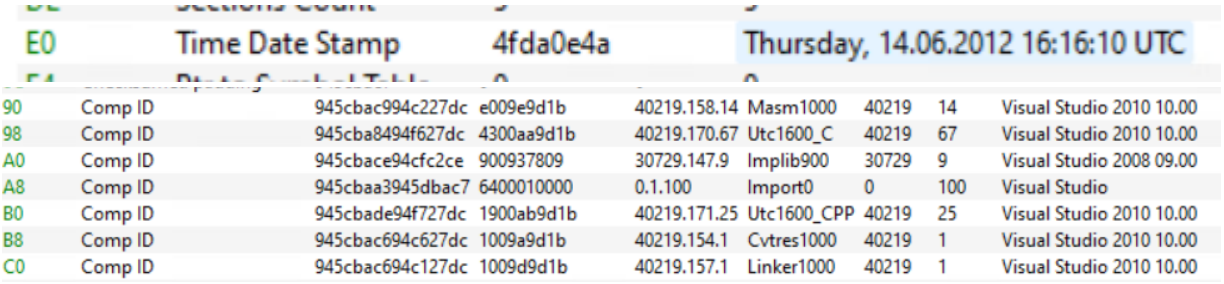
The executable's section table contained text, rdata, data, rsrc, and reloc headers. This hints towards the fact that the executable is not packed.

```
          Path: C:\users\Student\Desktop\malware.exe

          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..............
```

The text file shown above was found using strings. The magic number inside of the text file (4D 5A) shows that the file is an executable file.



The file size of the primary file is 92.2MB which is quite large.



The above screenshots also show that the executable was compiled using Visual Studio 2010, and was created on June 14, 2012, at roughly 4pm if the time and date have not been altered.

Many Security vendors have this file hash listed as a trojan labeled as Injuke, SilverFox, SwimSnake, etc.

Popular threat label	trojan.generickdq/injuke	Threat categories	trojan	Family labels	generickdq	injuke	silverfox
Security vendors' analysis				Do you want to automate checks?			
AhnLab-V3	Trojan/Win.Generic.C5847116	Alibaba	Trojan:Win32/Injuke.3fcabc40				
AliCloud	Trojan:Win/Injuke.pfdb	ALYac	QD:Trojan.GenericKDQ.2FB68927B8				
Antiy-AVL	Trojan/Win32.SwimSnake	Arcabit	QD:Trojan.GenericQ.2FB68927B8				
BitDefender	QD:Trojan.GenericKDQ.2FB68927B8	CTX	Exe.trojan.generickdq				
DeepInstinct	MALICIOUS	DrWeb	Trojan.Siggen32.24717				
Emsisoft	QD:Trojan.GenericKDQ.2FB68927B8 (B)	eScan	QD:Trojan.GenericKDQ.2FB68927B8				
Fortinet	W32/GenKryptik.GXAW!tr	GData	QD:Trojan.GenericKDQ.2FB68927B8				
Google	Detected	Huorong	Trojan/Generic!2060F27EEBFBE1A7				
Ikarus	Trojan.Win32.Krypt	K7AntiVirus	Trojan (005d9d281)				
K7GW	Trojan (005d9d281)	Kaspersky	Trojan.Win32.Injuke.ploo				
Lionic	Trojan.Win32.GenericKDQ.4!c	McAfee Scanner	Tl!003734348D05				
Microsoft	Trojan:Win32/Malgent!MSR	Palo Alto Networks	Generic.ml				
QuickHeal	Trojan.Ghanarava.1774000637782e49	Rising	Backdoor.Lotok!8.111D5 (CLOUD)				

R12.exe

File Hashes (Found using PEbear)

SHA256	5b8f092f78bd81787ec9bc135fec572002da8347cc8251093c3b6f03f8875776
SHA1	bacde47397c0a801f78d00bd033d8a618f2bee5d
MD5	dae5a2a4eb1968bde668e2c41ea6aa48
Imp	bcf9b6e7de705057b360c929a58da40b

Architecture: Intel 386 (32-bit)

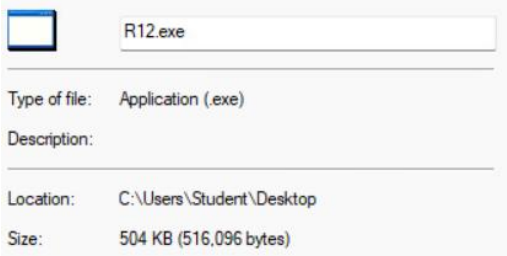
F4	Machine	14c	Intel 386
F6	Sections Count	4	4

The sections table includes text, rdata, data, and rsrc. This hints towards the file not being packed

Disasm: .text | General | Strings | DOS Hdr | Rich Hdr | Fil

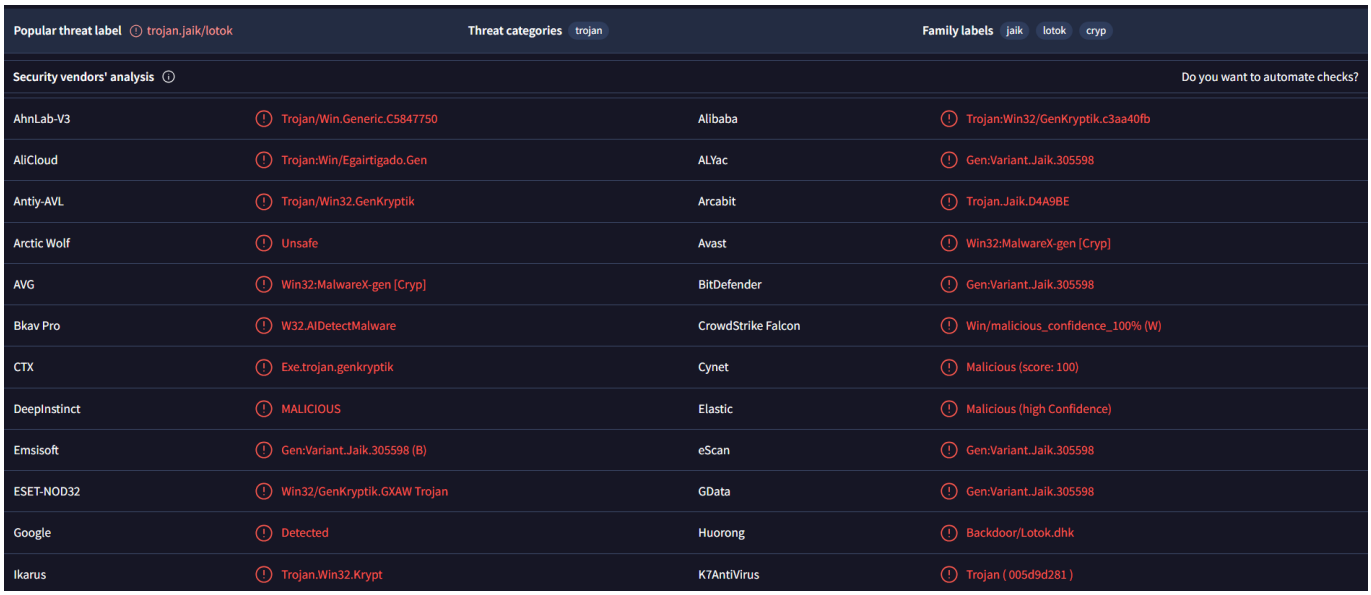
Offset	Name	Value
0	Magic number	5A4D

A magic number of 5A4D shows that the file is an executable.



R12.exe is only a 504 KB file, which is quite small, this is expected because from the report below, R12.exe is assumed to be the setup executable.

Many vendors have flagged R12.exe as a trojan, commonly under the labels of jaik, lotok, and cryp.



Popular threat label: trojan.jaik/lotok | Threat categories: trojan | Family labels: jaik, lotok, cryp

Security vendors' analysis — Do you want to automate checks?

AhnLab-V3	Trojan/Win.Generic.C5847750	Alibaba	Trojan:Win32/GenKryptik.c3aa40fb
AliCloud	Trojan:Win/Egairtigado.Gen	ALYac	Gen:Variant.Jaik.305598
Antiy-AVL	Trojan/Win32.GenKryptik	Arcabit	Trojan.Jaik.D4A9BE
Arctic Wolf	Unsafe	Avast	Win32:MalwareX-gen [Cryp]
AVG	Win32:MalwareX-gen [Cryp]	BitDefender	Gen:Variant.Jaik.305598
Bkav Pro	W32.AIDetectMalware	CrowdStrike Falcon	Win/malicious_confidence_100% (W)
CTX	Exe.trojan.genkryptik	Cynet	Malicious (score: 100)
DeepInstinct	MALICIOUS	Elastic	Malicious (high Confidence)
Emsisoft	Gen:Variant.Jaik.305598 (B)	eScan	Gen:Variant.Jaik.305598
ESET-NOD32	Win32/GenKryptik.GXAW Trojan	GData	Gen:Variant.Jaik.305598
Google	Detected	Huorong	Backdoor/Lotok.dhk
Ikarus	Trojan.Win32.Krypt	K7AntiVirus	Trojan (005d9d281)

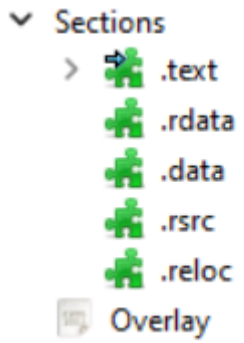
setupbeta_jisu.exe

File Hashes (Found using PEbear)

SHA256	b6dc8de481a4ad669f5e03a28e95268c00abc4bf27a654b23e23f53a7d2482
SHA1	eeb064f4929c4d666f128a7913903b2f3ec68f2
MD5	721a5b50e5ca0fdc3341577a6fcc4396
Imp	7787029a5717afac6bac673ea7aaac67

Architecture: Intel 386 (32-bit)

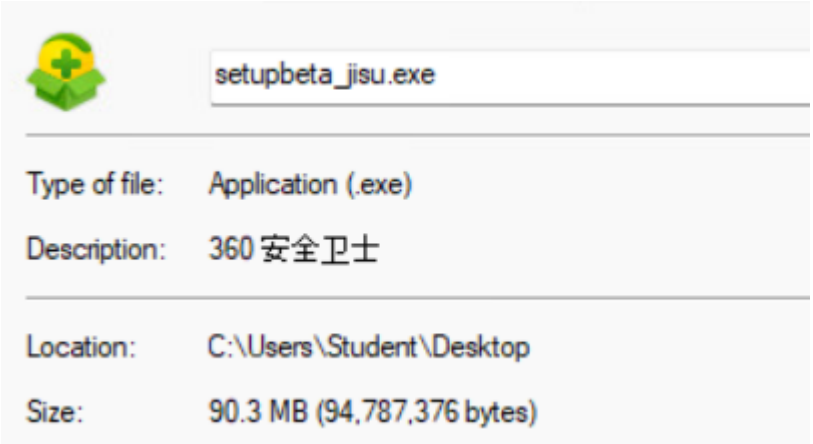
Offset	Name	Value	Meaning
104	Machine	14c	Intel 386



The sections table for this executable shows text, rdata, data, rsrc, and reloc. This hints towards the program not being packed.

Offset	Name	Value
0	Magic number	5A4D

The magic number of 5A4D shows the file is executable.



The size of the file on the disk is 90.3MB which is quite large.

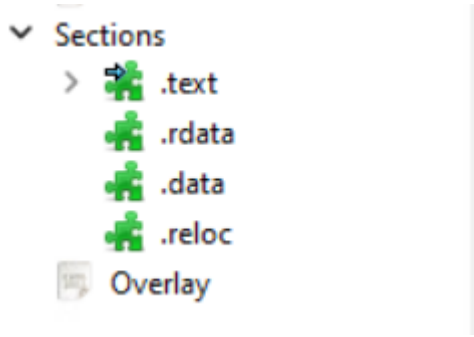
Lua5.1.dll

File Hashes (Found using PEbear)

SHA256	12cb9b8f59c0ef40ea8f28bfc59a29f12dc28332bf44b1a5d8d6a8823365650
SHA1	280fac9cf711d93c95f6b80ac97d89cf5853c096
MD5	b5fc476c1bf08d5161346cc7dd4cb0ba
Imp	15d95afb470c5f82193b2d9e98fc96d1

Architecture: Intel 386 (32-bit)

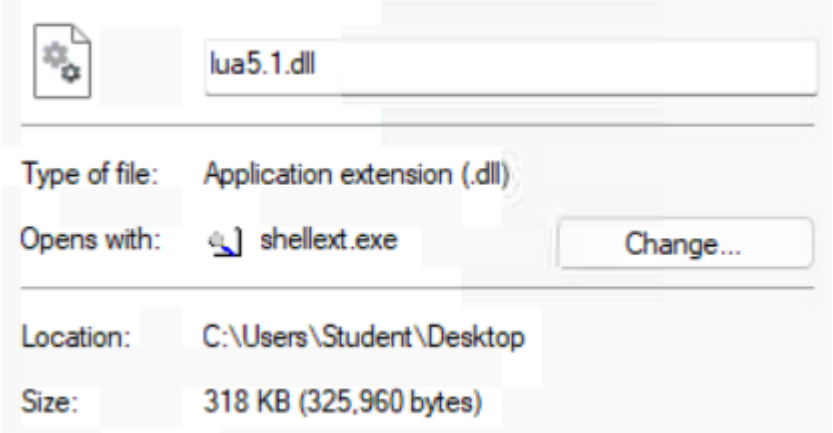
Offset	Name	Value	Meaning
E4	Machine	14c	Intel 386



The sections table contains text, rdata, data, and reloc, meaning the file is not packed.

Offset	Name	Value
0	Magic number	5A4D

The Magic Number of the file shows that it is an executable.



The size of the file is 318KB, after looking at the file sizes of this and the ones listed above, it is assumed that setupbeta_jisu.exe is the file doing most of the heavy lifting.

Preparing the Environment

Creating the Executable Handle

While analyzing the file using Ghidra, the file starts at a function that begins by fetching the path of the current file in execution. This is done via the GetModuleFileNameA function.

```
                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    undefined __fastcall GetExePath(int ExePath)
                      assume FS_OFFSET = 0xffdff000
    undefined         AL:1           <RETURN>
    int               ECX:4          ExePath
                    GetExePath                                      XREF[1]:     FUN_00401
0040121e 68 04 01       PUSH        0x104
         00 00
00401223 81 c1 20       ADD         ExePath,0x1120
         11 00 00
00401229 51             PUSH        ExePath
0040122a 6a 00          PUSH        0x0
0040122c ff 15 10       CALL        dword ptr [->KERNEL32.DLL::GetModuleFileNameA]   = 0000
         70 40 00
00401232 c3             RET

00401f7f e8 9a f2       CALL        GetExePath
         ff ff
00401f84 8b ce          MOV         ExePath,ESI
00401f86 e8 a8 f2       CALL        FUN_00401233
```

After the file path is found, the program moves the file path into a variable that is used in the function call to FUN_00401233.

In FUN_00401233, The previously found file path is opened and a handle for the file is created.

```
00401252 ff 15 1c       CALL        dword ptr [->KERNEL32.DLL::_lopen]               = 0000
         70 40 00
00401258 8b d8          MOV         EBX,EAX
0040125a 89 9e 30       MOV         dword ptr [ESI + 0x1530],EBX
         15 00 00
00401260 83 fb ff       CMP         EBX,-0x1
00401263 75 1b          JNZ         LAB_00401280
00401265 68 9c 72       PUSH        s_Unable_to_open_archive_file_0040729c           = "Una
         40 00
```

Allocating Memory to the Executable

After the program successfully creates a handle to the currently executing function, the program attempts to allocate 128KB of memory to the running executable.

```
00401265 68 9c 72        PUSH       s_Unable_to_open_archive_file_0040729c          = "Una
         40 00
0040126a 83 c6 08        ADD        ESI,0x8
0040126d 56              PUSH       ESI
0040126e ff 15 8c        CALL       dword ptr [->KERNEL32.DLL::lstrcpyA]            = 0000
         70 40 00
00401274 c7 45 f8        MOV        dword ptr [EBP + local_c],0x32
         32 00 00 00
0040127b e9 3f 02        JMP        LAB_004014bf
         00 00

                     LAB_00401280                                    XREF[1]:     00401263(
00401280 68 00 f4        PUSH       0x1f400
         01 00
00401285 e8 22 15        CALL       _malloc                                       void *
```

It is important to note, as this function and the following functions are executing, the program is tracking the success of the functions by returning the value stored in the variable *local_c*. The job of this variable is to tell the calling function how successful the execution was of the called function to determine the next path of execution.

```
004014c5 8b 45 f8        MOV        EAX,dword ptr [EBP + local_c]
004014c8 59              POP        param_1
004014c9 5f              POP        EDI
004014ca 5e              POP        ESI
004014cb 5b              POP        EBX
004014cc c9              LEAVE
004014cd c3              RET
```

If the first function runs successfully and the handle is opened, meaning that *local_c* returned as zero, the program falls through the jump if not zero command to the function *FUN_0040188b*.

```
00401f86 e8 a8 f2        CALL       FUN_00401233                                  undefi
         ff ff
00401f8b 8b d8           MOV        EBX,EAX
00401f8d 33 ff           XOR        EDI,EDI
00401f8f 3b df           CMP        EBX,EDI
00401f91 75 30           JNZ        LAB_00401fc3
00401f93 8b ce           MOV        ExePath,ESI
00401f95 e8 f1 f8        CALL       FUN_0040188b                                  undefi
```

At *FUN_0040188b*, the program saves the current directory of the executable inside of the variable *local_10c*.

```
004018bd ff 15 54        CALL       dword ptr [->KERNEL32.DLL::GetCurrentDirectoryA] = 00(
         70 40 00
004018c3 56              PUSH       ESI
004018c4 8d 85 f8        LEA        EAX=>local_10c,[EBP + 0xfffffef8]
         fe ff ff
```

Hiding the Malware

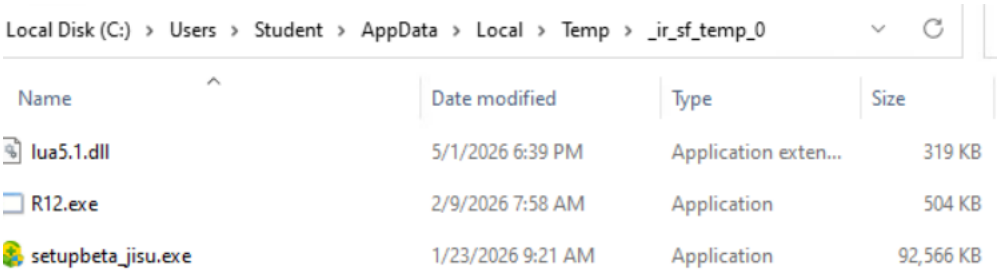
After the program successfully saved the current directory, it attempts to locate the temp folder on the system. If the temp folder is found the program replicates and stores itself inside of the machines temp folder to avoid detection.

```
004018dd ff 15 90        CALL       dword ptr [->KERNEL32.DLL::GetTempPathA]      = 0000
         70 40 00
004018e3 8b 1d 50        MOV        EBX,dword ptr [->KERNEL32.DLL::lstrlenA]      = 0000
         70 40 00
004018e9 8d b7 08        LEA        ESI,[EDI + 0x1008]
         10 00 00
004018ef 56              PUSH       ESI
004018f0 ff d3           CALL       EBX=>KERNEL32.DLL::lstrlenA
004018f2 8b 3d 8c        MOV        EDI,dword ptr [->KERNEL32.DLL::lstrcpyA]      = 0000
         70 40 00
004018f8 83 f8 02        CMP        EAX,0x2
004018fb 7e 1a           JLE        LAB_00401917
004018fd 8b 8d e0        MOV        param_1,dword ptr [EBP + local_424]
         fb ff ff
00401903 56              PUSH       ESI
00401904 e8 3e fe        CALL       FUN_00401747                                  undefi
         ff ff
00401909 85 c0           TEST       EAX,EAX
0040190b 74 0a           JZ         LAB_00401917
0040190d 56              PUSH       ESI
0040190e 8d 85 f8        LEA        EAX=>local_10c,[EBP + 0xfffffef8]
         fe ff ff
00401914 50              PUSH       EAX
00401915 ff d7           CALL       EDI=>KERNEL32.DLL::lstrcpyA
```

```
003E196E  return to malware.003E196E from ???
0099E60C
003E746C  malware."%s%s_%d"
0099E710  "C:\\Users\\Student\\AppData\\Local\\Temp\\"
003E7474  malware."_ir_sf_temp"
```

The screenshot to the left is from x32dbg, where the address of the temp folder is captured by the program.

The screenshot to the right shows the program files after they have relocated themselves into the machines temp folder.



After moving the program files to the temp folder, the program continues execution and falls through the jump conditional and continues to function *FUN_004014ce.*

```
00401f95 e8 f1 f8        CALL        FUN_0040188b                                undefi
         ff ff
00401f9a 8b d8           MOV         EBX,EAX
00401f9c 3b df           CMP         EBX,EDI
00401f9e 75 23           JNZ         LAB_00401fc3
00401fa0 8b ce           MOV         ExePath,ESI
00401fa2 e8 27 f5        CALL        FUN_004014ce                                undefi
```

Launching the Setup Executable

In *FUN_00401ce,* the program attempts to launch the setup executable that was moved inside of temp. If Execution is successful, the program continues to *FUN_004015e0.*

```
00401fa2 e8 27 f5        CALL        FUN_004014ce                                undefi
         ff ff
00401fa7 8b d8           MOV         EBX,EAX
00401fa9 3b df           CMP         EBX,EDI
00401fab 75 16           JNZ         LAB_00401fc3
00401fad 8b ce           MOV         ExePath,ESI
00401faf e8 2c f6        CALL        FUN_004015e0                                int FU
         ff ff
```

In *FUN_004015e0,* the program begins to execute an encryption function to cover up the setup executables behavior after the environmental setup is complete (setup.exe is inside of temp folder). First, the program locates and attempts to read the setup file inside of the 128KB buffer that was created.

```
004014f5 57              PUSH        EDI
004014f6 6a 00           PUSH        0x0
004014f8 ff b6 38        PUSH        dword ptr [ESI + 0x1538]
         15 00 00
004014fe ff b6 30        PUSH        dword ptr [ESI + 0x1530]
         15 00 00
00401504 ff 15 18        CALL        dword ptr [->KERNEL32.DLL::_llseek]          = 000098
         70 40 00
0040150a ff b6 40        PUSH        dword ptr [ESI + 0x1540]
         15 00 00
00401510 53              PUSH        EBX
00401511 ff b6 30        PUSH        dword ptr [ESI + 0x1530]
         15 00 00
00401517 ff 15 14        CALL        dword ptr [->KERNEL32.DLL::_lread]           = 000098
         70 40 00
```

Encrypting the Setup Executable

Once the setup folder is read, the program begins xor encryption with a value of seven on each byte inside of the setup executable using using a for while loop.

```
                    LAB_00401562                                    XREF[1]:     00401582(j)
00401562 8b c1          MOV         EAX,param_1
00401564 99             CDQ
00401565 3b 96 44       CMP         EDX,dword ptr [ESI + 0x1544]
         15 00 00
0040156b 77 17          JA          LAB_00401584
0040156d 72 08          JC          LAB_00401577
0040156f 3b 86 40       CMP         EAX,dword ptr [ESI + 0x1540]
         15 00 00
00401575 73 0d          JNC         LAB_00401584

                    LAB_00401577                                    XREF[1]:     0040156d(j)
00401577 80 34 19 07    XOR         byte ptr [param_1 + EBX*0x1],0x7
0040157b 41             INC         param_1
0040157c 81 f9 d0       CMP         param_1,0x7d0
         07 00 00
00401582 7c de          JL          LAB_00401562
```

After encryption has completed, the program writes the memory back to the buffer.

```
                    LAB_00401584                                    XREF[2]:     0040156b(j)
00401584 ff b6 40       PUSH        dword ptr [ESI + 0x1540]
         15 00 00
0040158a 53             PUSH        EBX
0040158b 57             PUSH        EDI
0040158c ff 15 20       CALL        dword ptr [->KERNEL32.DLL::_lwrite]          = 000098
```

Once the program successfully encrypts the setup executable inside of temp, it continues execution to the function *FUN_004015e0.*

```
00401fa2 e8 27 f5       CALL        FUN_004014ce                                 undefine
         ff ff
00401fa7 8b d8          MOV         EBX,EAX
00401fa9 3b df          CMP         EBX,EDI
00401fab 75 16          JNZ         LAB_00401fc3
00401fad 8b ce          MOV         ExePath,ESI
00401faf e8 2c f6       CALL        FUN_004015e0                                 int FUN_
         ff ff
```

Parsing the Lua DLL

At *FUN_004015e0*, the program navigates to a specific offset of the file using *_llseek* and then calls *_lread* to read 8 bytes of the buffer at that offset. This is the file size of a Lua DLL that the program is trying to read from.

```
004015f3 50             PUSH        EAX
004015f4 ff b6 38       PUSH        dword ptr [ESI + 0x1538]
         15 00 00
004015fa 89 45 fc       MOV         dword ptr [EBP + local_8],EAX
004015fd ff b6 30       PUSH        dword ptr [ESI + 0x1530]
         15 00 00
00401603 89 45 f0       MOV         dword ptr [EBP + local_14],EAX
00401606 89 45 f4       MOV         dword ptr [EBP + local_10],EAX
00401609 ff d7          CALL        EDI=>KERNEL32.DLL::_llseek
0040160b 8b 1d 14       MOV         EBX,dword ptr [->KERNEL32.DLL::_lread]          = 000098
         70 40 00
00401611 6a 08          PUSH        0x8
00401613 8d 45 f0       LEA         EAX=>local_14,[EBP + -0x10]
00401616 50             PUSH        EAX
00401617 ff b6 30       PUSH        dword ptr [ESI + 0x1530]
         15 00 00
0040161d ff d3          CALL        EBX=>KERNEL32.DLL::_lread
0040161f 83 f8 08       CMP         EAX,0x8
00401622 74 18          JZ          LAB_0040163c
00401624 68 5c 73       PUSH        s_Could_not_find_Lua_DLL_file_size_0040735c      = "Could
         40 00
00401629 8d 46 08       LEA         EAX,[ESI + 0x8]
0040162c 50             PUSH        EAX
0040162d ff 15 8c       CALL        dword ptr [->KERNEL32.DLL::lstrcpyA]            = 000098
```

After the program knows exactly how much data needs to be extracted from the DLL, it attempts to allocate memory using *malloc*.

```
0040165e 6a 00          PUSH        0x0
00401660 ff b6 38       PUSH        dword ptr [ESI + 0x1538]
         15 00 00
00401666 ff b6 30       PUSH        dword ptr [ESI + 0x1530]
         15 00 00
0040166c ff d7          CALL        EDI=>KERNEL32.DLL::_llseek
0040166e ff 75 f0       PUSH        dword ptr [EBP + local_14]
00401671 ff 75 f8       PUSH        dword ptr [EBP + local_c]
00401674 ff b6 30       PUSH        dword ptr [ESI + 0x1530]
         15 00 00
0040167a ff d3          CALL        EBX=>KERNEL32.DLL::_lread
0040167c 33 c9          XOR         param_1,param_1
0040167e 3b 45 f0       CMP         EAX,dword ptr [EBP + local_14]
00401681 0f 85 81       JNZ         LAB_00401708
         00 00 00
```

If allocation fails, it jumps to *LAB_00401729* where the program prints “Failed to alloc memory”

```
                    LAB_00401729                                    XREF[1]:     00401658(j)
00401729 68 f0 72        PUSH        s_Failed_to_alloc_memory._004072f0              = "Faile
         40 00
0040172e 83 c6 08        ADD         ESI,0x8
00401731 56              PUSH        ESI
00401732 ff 15 8c        CALL        dword ptr [->KERNEL32.DLL::lstrcpyA]            = 000098
         70 40 00
```

Once memory is allocated, the program attempts to read contents from the DLL into the newly created memory buffer.

```
0040165e 6a 00           PUSH        0x0
00401660 ff b6 38        PUSH        dword ptr [ESI + 0x1538]
         15 00 00
00401666 ff b6 30        PUSH        dword ptr [ESI + 0x1530]
         15 00 00
0040166c ff d7           CALL        EDI=>KERNEL32.DLL::_llseek
0040166e ff 75 f0        PUSH        dword ptr [EBP + local_14]
00401671 ff 75 f8        PUSH        dword ptr [EBP + local_c]
00401674 ff b6 30        PUSH        dword ptr [ESI + 0x1530]
         15 00 00
0040167a ff d3           CALL        EBX=>KERNEL32.DLL::_lread
0040167c 33 c9           XOR         param_1,param_1
0040167e 3b 45 f0        CMP         EAX,dword ptr [EBP + local_14]
00401681 0f 85 81        JNZ         LAB_00401708
         00 00 00
```

If the read fails, the program jumps to *LAB_00401708,* where it prints, “Failed to read Lua DLL” .

```
                    LAB_00401708                                    XREF[2]:     00401681(j), 00
00401708 68 08 73        PUSH        s_Failed_to_read_Lua_DLL_00407308               = "Failed t
         40 00
0040170d 83 c6 08        ADD         ESI,0x8
00401710 56              PUSH        ESI
00401711 ff 15 8c        CALL        dword ptr [->KERNEL32.DLL::lstrcpyA]            = 0000982a
         70 40 00
```

Writing From the Lua DLL

The program then attempts to create a new file with the data read from the Lua DLL using *_lcreate*. If the program fails to do so, it prints “Unable to open Lua DLL file”. Otherwise, it jumps to *LAB_004016be.*

```
00401691 51             PUSH    param_1
00401692 8d 86 2c       LEA     EAX,[ESI + 0x142c]
         14 00 00
00401698 50             PUSH    EAX
00401699 ff 15 24       CALL    dword ptr [->KERNEL32.DLL::_lcreat]          = 0000988a
         70 40 00
0040169f 8b f8          MOV     EDI,EAX
004016a1 83 ff ff       CMP     EDI,-0x1
004016a4 75 18          JNZ     LAB_004016be
004016a6 68 40 73       PUSH    s_Unable_to_open_Lua_DLL_file_00407340       = "Unable t
         40 00
004016ab 8d 46 08       LEA     EAX,[ESI + 0x8]
004016ae 50             PUSH    EAX
004016af ff 15 8c       CALL    dword ptr [->KERNEL32.DLL::lstrcpyA]         = 0000982a
         70 40 00
```

At *LAB_004016be*, the program attempts to write to the new file with the data from the Lua DLL using *_lwrite*.

```
                    LAB_004016be                                    XREF[1]:     004016a4(j)
004016be ff 75 f0       PUSH    dword ptr [EBP + local_14]
004016c1 ff 75 f8       PUSH    dword ptr [EBP + local_c]
004016c4 57             PUSH    EDI
004016c5 ff 15 20       CALL    dword ptr [->KERNEL32.DLL::_lwrite]          = 00009880
         70 40 00
004016cb 33 c9          XOR     param_1,param_1
004016cd 3b 45 f0       CMP     EAX,dword ptr [EBP + local_14]
004016d0 75 05          JNZ     LAB_004016d7
004016d2 3b 4d f4       CMP     param_1,dword ptr [EBP + local_10]
004016d5 74 16          JZ      LAB_004016ed
```

The Debug Flag

It is important to note that if any one of the above functions fails, meaning the return value was nonzero, execution will be redirected to *LAB_00401fc3.* It is at this point the program must decide whether to hide, or continue. One way the function decides how to continue is the global variable *DAT_0040ab80* which acts as a debug flag.

```
                    LAB_00401fc3                                    XREF[4]:     00401f91(
                                                                                 00401fab(
00401fc3 39 3d 80       CMP     dword ptr [DAT_0040ab80],EDI
         ab 40 00
00401fc9 75 5d          JNZ     LAB_00402028
```

The debug flag is initalized as zero, meaning that the program is not being ran in debug mode.
For the debug flag to be changed, the string "/~DBG" would have to be present, alerting the program that It is being ran in debug mode.

```
00401122 68 b8 71        PUSH        s_/~DBG_004071b8                                  = "/~I
         40 00
00401127 6a ff           PUSH        -0x1
00401129 8d 85 f8        LEA         EAX=>local_10c,[EBP + 0xfffffef8]
         fe ff ff
0040112f 50              PUSH        EAX
00401130 56              PUSH        ESI
00401131 6a 7f           PUSH        0x7f
00401133 ff 15 94        CALL        dword ptr [->KERNEL32.DLL::CompareStringA]        = 000(
         70 40 00
00401139 83 f8 02        CMP         EAX,0x2
0040113c 75 06           JNZ         LAB_00401144
0040113e 89 35 80        MOV         dword ptr [DAT_0040ab80],ESI
```

If the program is being ran in debug mode, meaning the debug flag is nonzero, the program will jump to *LAB_00402028*, where the program ends prematurely.

```
                    LAB_00402028                                    XREF[2]:     00401fc9(
00402028 5f              POP         EDI
00402029 5e              POP         ESI
0040202a 8b c3           MOV         EAX,EBX
0040202c 5b              POP         EBX
0040202d c3              RET
```

The purpose of this is not evasion of a debugger, it is a feature that the developer of the program can use during creation and testing. If the malware runs into an error, and it is not in debug mode, there are two options; Both of which depend on if the program handle was created successfully. If the program handle failed to initialize, the program falls through the jump conditional and deletes itself from the machines temp folder to avoid detection and clean uo after itself.

```
00401fcd ff 15 70        CALL        dword ptr [->KERNEL32.DLL::Sleep]                 = 0000
         70 40 00
00401fd3 8d 86 24        LEA         EAX,[ESI + 0x1224]
         12 00 00
00401fd9 39 be 0c        CMP         dword ptr [ESI + 0x110c],EDI
         11 00 00
00401fdf 74 21           JZ          LAB_00402002
00401fe1 8b 3d 4c        MOV         EDI,dword ptr [->KERNEL32.DLL::DeleteFileA]       = 0000
         70 40 00
00401fe7 50              PUSH        EAX
00401fe8 ff d7           CALL        EDI=>KERNEL32.DLL::DeleteFileA
```

Without the debug flag, the developer would run into the program deleting itself every time that it fails, leading to a nightmare during testing.

Data Collection

Creating The Information Buffer

Once the program has written data from the Lua DLL, the program continues to the data collection phase on the target. In *FUN_00401b8c* (which has been renamed to MalwareMain), before GetCurrentProcess is called EAX and 0x8 are pushed onto the stack. It is important to note that GetCurrentProcess takes no arguments. After GetCurrentProcess executes, EAX is filled with the process handle. After this, OpenProcessToken is called with the process handle as the return value from GetCurrentProcess, the desired permissions as 0x8 (Token Query), and local_134 and the place to put the token that is opened as a result. Since the desired permissions are so low, the function will almost always complete successfully, which will skip the jump at memory address LAB_00401de8.

```
00401d13 83 a5 d0        AND        dword ptr [EBP + local_134],0x0
         fe ff ff 00
00401d1a 8d 85 d0        LEA        EAX=>local_134,[EBP + 0xfffffed0]
         fe ff ff
00401d20 50              PUSH       EAX
00401d21 6a 08           PUSH       0x8
00401d23 ff 15 68        CALL       dword ptr [->KERNEL32.DLL::GetCurrentProcess]    = 000099b0
         70 40 00
00401d29 50              PUSH       EAX
00401d2a ff 15 04        CALL       dword ptr [->ADVAPI32.DLL::OpenProcessToken]     = 00009a92
         70 40 00
00401d30 85 c0           TEST       EAX,EAX
00401d32 0f 84 b0        JZ         LAB_00401de8
         00 00 00
```

If the jump instruction falls through, the program will then allocate memory using malloc. If malloc succeeds, it returns a pointer to the allocated memory space, and then stores that saved memory space into local_138. Then, the program checks if malloc succeeded, and falls through the jump if it did.

```
00401d3f 68 00 40        PUSH       0x4000
         00 00
00401d44 e8 63 0a        CALL       _malloc                                  void * _malloc(size_t _
         00 00
00401d49 59              POP        param_1
00401d4a 89 85 cc        MOV        dword ptr [EBP + local_138],EAX
         fe ff ff
00401d50 85 c0           TEST       EAX,EAX
00401d52 0f 84 90        JZ         LAB_00401de8
         00 00 00
```

Fetching Security Identifier

Next, the program passes five arguments onto the stack before calling GetTokenInformation. The first argument is the token handle that the function will gather information on. This handle was created in OpenProcessToken. Argument two is 0x1 which tells the function that the buffer passed in next should be filled the user account information of the token. Argument three is the buffer that the program will fill with the user account information of the token. This argument is passed in as EAX which is the 16KB buffer created from malloc previously. Argument four is the token information length which just tells the function the length of the buffer it is writing into, which is 16KB in this case (0x4000). The last argument is a buffer that the function writes how many bytes the information buffer requires to fit the information, if it is not large enough the function fails, and the jump condition falls through. Finally, the program gets ready to process the information by zeroing out local_130.

```
00401d5e 51              PUSH        param_1
00401d5f 68 00 40        PUSH        0x4000
         00 00
00401d64 50              PUSH        EAX
00401d65 6a 01           PUSH        0x1
00401d67 ff b5 d0        PUSH        dword ptr [EBP + local_134]
         fe ff ff
00401d6d ff 15 00        CALL        dword ptr [->ADVAPI32.DLL::GetTokenInformation]  = 00009a7c
         70 40 00
00401d73 85 c0           TEST        EAX,EAX
00401d75 74 71           JZ          LAB_00401de8
00401d77 83 a5 d4        AND         dword ptr [EBP + local_130],0x0
         fe ff ff 00
```

Next comes processing the information. After local_130 is zeroed, the address is stored into EAX. EAX is then pushed onto the stack. After EAX is pushed, the previously saved memory space from malloc is saved into EAX and then pushed as a pointer to the stack, then FUN_00401821 is called.

```
00401d7e 8d 85 d4        LEA         EAX=>local_130,[EBP + 0xfffffed4]
         fe ff ff
00401d84 50              PUSH        EAX
00401d85 8b 85 cc        MOV         EAX,dword ptr [EBP + local_138]
         fe ff ff
00401d8b ff 30           PUSH        dword ptr [EAX]
00401d8d 8b cb           MOV         param_1,EBX
00401d8f e8 8d fa        CALL        FUN_00401821                                  int FUN_00401821(int pa
         ff ff
```

FUN_00401821 reveals a lot of functionality in the first couple of lines. The first thing it does is load Advapi32.dll which is used for interacting with user accounts, permissions, and the registry. This is a huge red flag, because the calling function passed in a buffer that contains user account information, meaning it passed that buffer off to a function that can process it.

```
00401826 68 9c 73        PUSH        s_Advapi32.dll_0040739c                       = "Advapi32.dll"
         40 00
0040182b 33 ff           XOR         EDI,EDI
0040182d ff 15 3c        CALL        dword ptr [->KERNEL32.DLL::LoadLibraryA]      = 000098ec
         70 40 00
```

Converting Security Identifier to String

After LoadLibrary is called, the library handle is stored in ESI. The next comparison checks if the library was fetched successfully; by comparing the ESI register which is supposed to hold the library handle to the zeroes out EDI register, the program can quickly tell if the library handle is nonzero, which makes the jump fall through.

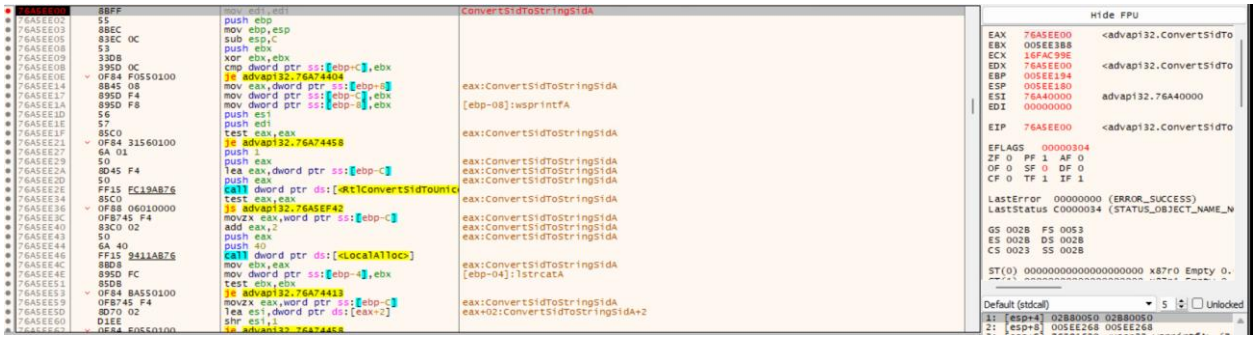
The function performs a validation check on *param_2*, which holds the base memory address of the buffer containing the retrieved account information. By comparing *param_2* to EDI (which was previously zeroed via XOR), the program verifies that the pointer is not NULL. If the pointer is valid (non-zero), the conditional jump is not taken, and the execution falls through to the next instruction. This safety check ensures the malware does not attempt to process a null memory address, which would result in an application crash.

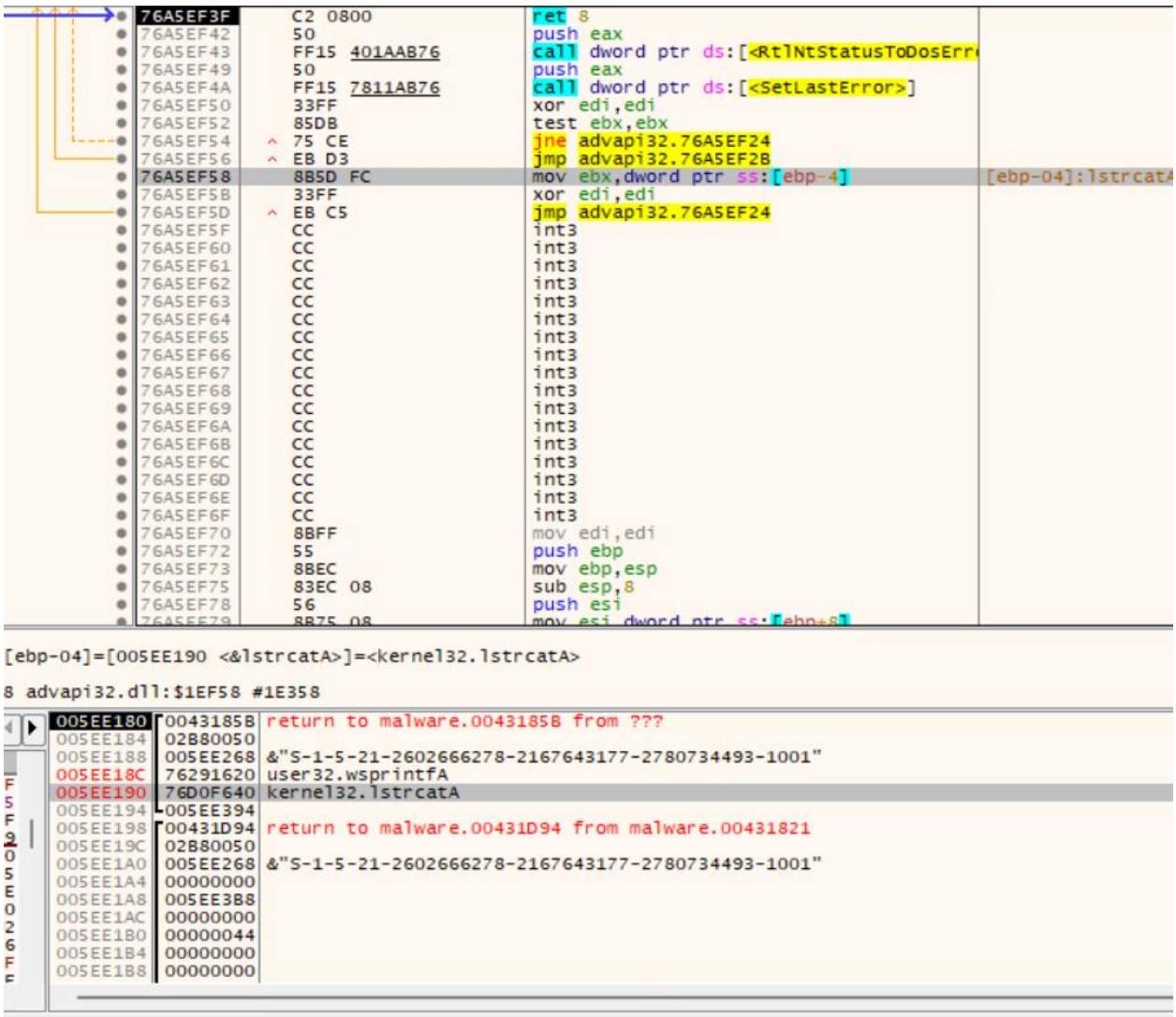
The function then dynamically fetches the ConvertSidToStringSidA function by passing in the handle to Advapi.dll and the string “ConvertSidToStringSidA” to GetProcAddress, which returns the address of the ConvertSidToStringSidA function so it can be used. The address is then compared to EDI which is the zero register to verify that EAX has successfully stored the memory address of ConvertSidToStringSidA.

```
00401833 8b f0           MOV       ESI,EAX
00401835 3b f7           CMP       ESI,EDI
00401837 74 2b           JZ        LAB_00401864
00401839 39 7d 0c        CMP       dword ptr [EBP + param_2],EDI
0040183c 74 26           JZ        LAB_00401864
0040183e 39 7d 08        CMP       dword ptr [EBP + param_1],EDI
00401841 74 21           JZ        LAB_00401864
00401843 68 84 73        PUSH      s_ConvertSidToStringSidA_00407384              = "ConvertSidToStringSi
         40 00
00401848 56              PUSH      ESI
00401849 ff 15 38        CALL      dword ptr [->KERNEL32.DLL::GetProcAddress]     = 000098da
         70 40 00
0040184f 3b c7           CMP       EAX,EDI
00401851 74 0a           JZ        LAB_0040185d
```

The account information buffer and empty buffer passed into this function are being carried and passed into ConvertSidToStringSidA which was dynamically loaded into EAX.
In the ConvertSidToStringSidA function, it converts the security identifier of the account into a string format and stores it into the empty buffer passed in as param_1. After, the return value for ConvertSidToStringSidA is stored in EDI, and the program frees the library that was loaded previously to attempt to hide its footprint. The function returns the value 0x8 which is the access identifier for TOKEN_QUERY.

```
00401853 ff 75 0c        PUSH        dword ptr [EBP + param_2]
00401856 ff 75 08        PUSH        dword ptr [EBP + param_1]
00401859 ff d0           CALL        EAX
0040185b 8b f8           MOV         EDI,EAX

                     LAB_0040185d                                    XREF[1]:     00401851(j)
0040185d 56              PUSH        ESI
0040185e ff 15 34        CALL        dword ptr [->KERNEL32.DLL::FreeLibrary]          = 000098cc
         70 40 00

                     LAB_00401864                                    XREF[3]:     00401837(j), 0040183c(j),
                                                                                  00401841(j)
00401864 8b c7           MOV         EAX,EDI
00401866 5f              POP         EDI
00401867 5e              POP         ESI
00401868 5d              POP         EBP
00401869 c2 08 00        RET         0x8
```





After FUN_00401821 has stores the SID string into EAX, FUN_00401b8c checks that EAX is not zero, and checks that the buffer containing the user SID string is nonzero. If these both succeed it would mean that local_130 successfully has the user SID string stored and is ready to proceed.

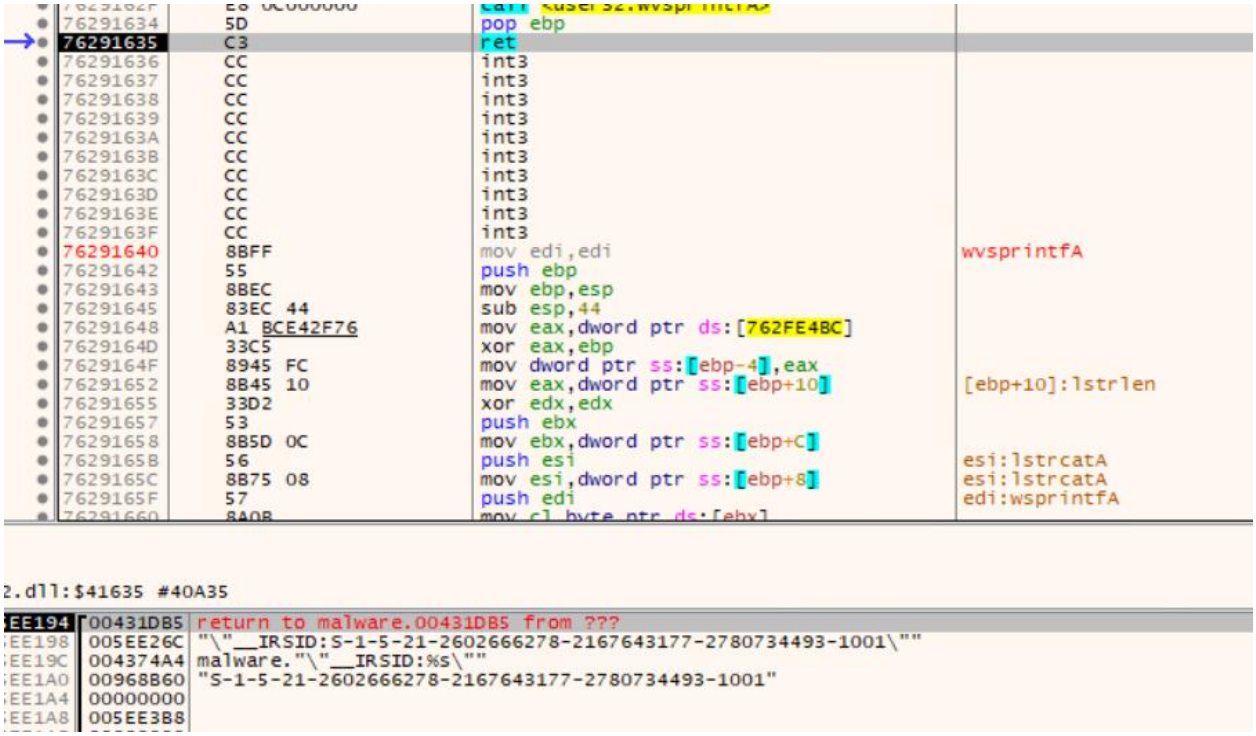
After the function has deemed it is ready to continue, the user SID string is pushed onto the stack as local_130. EAX is loaded with the address of 292 byte character array which is the maximum length of a Windows SID string. Parameter two is then passed as the string “__IRSID:%s”, and parameter three is the SID string which is being passed in as EAX. After the parameters are all pushed, wsprintfA is called which stores the formatted output “__IRSID:(account security identifier)” into the 292 byte buffer passed into wsprintfA.

```
00401d8f e8 8d fa ff ff     CALL        FUN_00401821                                  int FUN_00401821(int pa
00401d94 85 c0              TEST        EAX,EAX
00401d96 74 44              JZ          LAB_00401ddc
00401d98 83 bd d4 fe ff ff  CMP         dword ptr [EBP + local_130],0x0
00401d9f 74 3b              JZ          LAB_00401ddc
00401da1 ff b5 d4 fe ff ff  PUSH        dword ptr [EBP + local_130]
00401da7 8d 85 d8 fe ff ff  LEA         EAX=>local_12c,[EBP + 0xfffffed8]
00401dad 68 a4 74 40 00     PUSH        s_"__IRSID:%s"_004074a4                       = "\"__IRSID:%s\""
00401db2 50                 PUSH        EAX
00401db3 ff d7              CALL        EDI=>USER32.DLL::wsprintfA
```

```
76291620    8BFF        mov edi,edi                                 wsprintfA
76291622    55          push ebp
76291623    8BEC        mov ebp,esp
76291625    8D45 10     lea eax,dword ptr ss:[ebp+10]               [ebp+10]:lstr
76291628    50          push eax                                    eax:"\"__IRTS
76291629    FF75 0C     push dword ptr ss:[ebp+C]
7629162C    FF75 08     push dword ptr ss:[ebp+8]
7629162F    E8 0C000000 call <user32.wvsprintfA>
76291634    5D          pop ebp
76291635    C3          ret
76291636    CC          int3
76291637    CC          int3
76291638    CC          int3
76291639    CC          int3
7629163A    CC          int3
7629163B    CC          int3
7629163C    CC          int3
7629163D    CC          int3
7629163E    CC          int3
7629163F    CC          int3
76291640    8BFF        mov edi,edi                                 wvsprintfA
76291642    55          push ebp
76291643    8BEC        mov ebp,esp
76291645    83EC 44     sub esp,44
76291648    A1 BCE42F76 mov eax,dword ptr ds:[762FE4BC]             eax:"\"__IRTS
7629164D    33C5        xor eax,ebp
7629164F    8945 FC     mov dword ptr ss:[ebp-4],eax
76291652    8B45 10     mov eax,dword ptr ss:[ebp+10]               [ebp+10]:lstr
76291655    33D2        xor edx,edx
76291657    53          push ebx
76291658    8B5D 0C     mov ebx,dword ptr ss:[ebp+C]
7629165B    56          push esi                                    esi:lstrcatA
7629165C    8B75 08     mov esi,dword ptr ss:[ebp+8]                esi:lstrcatA
7629165F    57          push edi                                    edi:wsprintfA
76291660    8A0B        mov cl,byte ptr ds:[ebx]
```

```
tfA>

r32.dll:$41620 #40A20 <wsprintfA>
```

```
005EE194  00431DB5  return to malware.00431DB5 from ???
005EE198  005EE26C  "\"__IRTSS:0\""
005EE19C  004374A4  malware."\"__IRSID:%s\""
005EE1A0  00968B60  "S-1-5-21-2602666278-2167643177-2780734493-1001"
005EE1A4  00000000
005EE1A8  005EE3B8
```



After the program captures the SID, the program adds a space to a larger buffer (DAT_004074e0) with the first lstrcatA call. This large buffer will be exfiltrated from the targets system to the attackers system. After it adds the space, it saves the SID string address to EAX and then pushes EAX, followed by the larger buffer to add the formatted SID string to a formatted buffer containing other user information.

```
00401dad 68 a4 74        PUSH      s_"__IRSID:%s"_004074a4                  = "\"__IRSID:%s\""
         40 00
00401db2 50              PUSH      EAX
00401db3 ff d7           CALL      EDI=>USER32.DLL::wsprintfA
00401db5 83 c4 0c        ADD       ESP,0xc
00401db8 68 e0 74        PUSH      DAT_004074e0                             = 20h
         40 00
00401dbd 8d bb 08        LEA       EDI,[EBX + 0x808]
         08 00 00
00401dc3 57              PUSH      EDI
00401dc4 ff d6           CALL      ESI=>KERNEL32.DLL::lstrcatA
00401dc6 8d 85 d8        LEA       EAX=>local_12c,[EBP + 0xfffffed8]
         fe ff ff
00401dcc 50              PUSH      EAX
00401dcd 57              PUSH      EDI
00401dce ff d6           CALL      ESI=>KERNEL32.DLL::lstrcatA
```

Capturing Other User Information

The program captures four other formatted user information strings and adds them to the large buffer to be exfiltrated. This is why a single space is being added before adding the information string. The four other fields are the follwing:

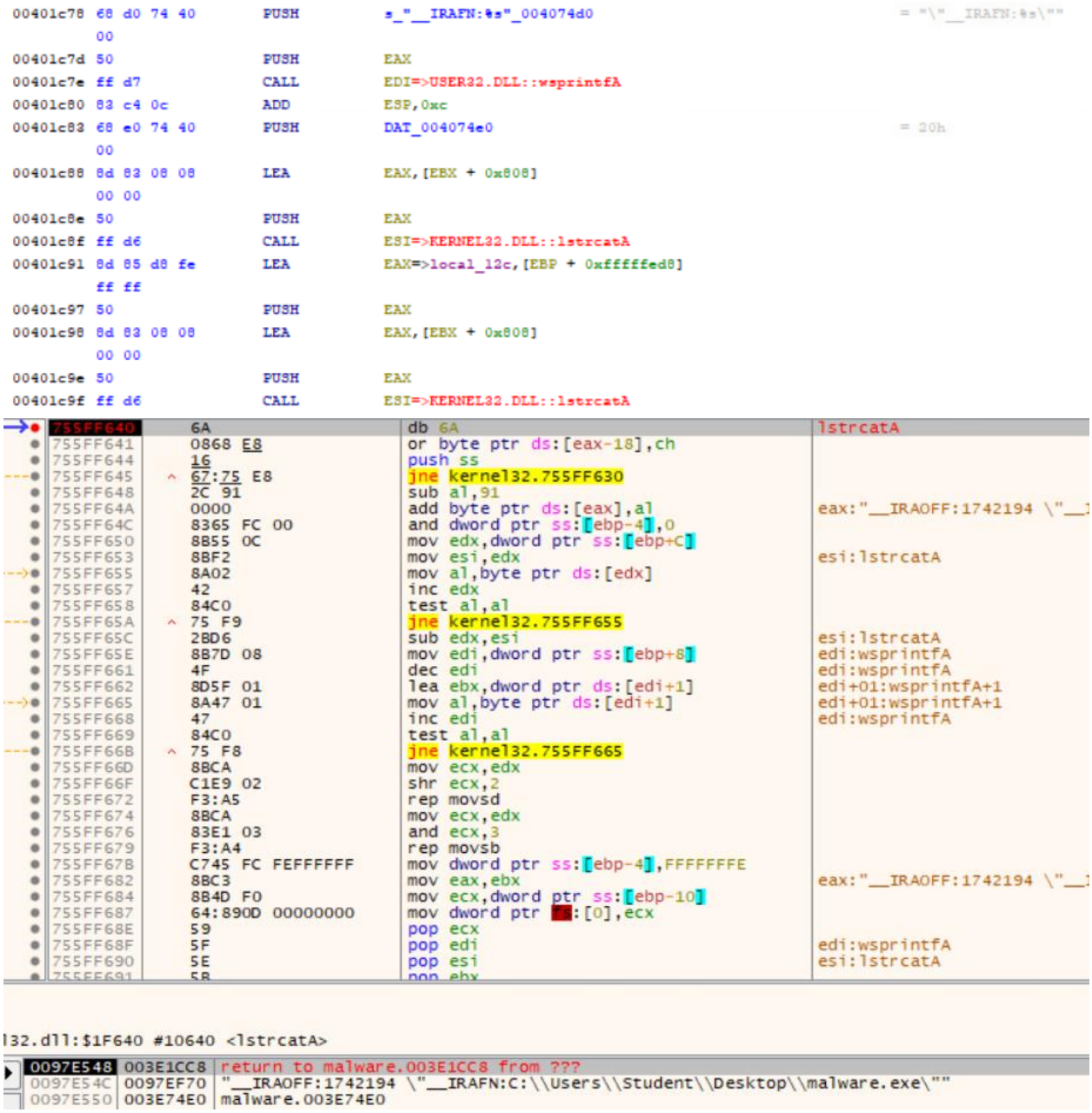
1.) The users active offline flag (how long the machine has been idle for), saved as "__IRAOFF:(users offline flag stored as a string)".

```
00401c13 68 e4 74 40 00     PUSH        s___IRAOFF:%I64u_004074e4                    = "__IRAOFF:%I64u"
00401c18 50                 PUSH        EAX
00401c19 89 b5 20 fe ff     MOV         dword ptr [EBP + local_1e4],ESI
         ff
00401c1f 89 b5 50 fe ff     MOV         dword ptr [EBP + local_1b4],ESI
         ff
00401c25 89 b5 24 fe ff     MOV         dword ptr [EBP + local_1e0],ESI
         ff
00401c2b 89 b5 48 fe ff     MOV         dword ptr [EBP + local_1bc],ESI
         ff
00401c31 ff d7              CALL        EDI->USER32.DLL::wsprintfA
00401c33 83 c4 34           ADD         ESP,0x34
00401c36 8d 83 08 08 00     LEA         EAX,[EBX + 0x808]
         00
00401c3c 50                 PUSH        EAX
00401c3d ff 15 50 70 40     CALL        dword ptr [->KERNEL32.DLL::lstrlenA]         = 00009836
         00
00401c43 8b 35 30 70 40     MOV         ESI,dword ptr [->KERNEL32.DLL::lstrcatA]     = 000098c0
         00
00401c49 85 c0              TEST        EAX,EAX
00401c4b 74 0e              JZ          LAB_00401c5b
00401c4d 68 e0 74 40 00     PUSH        DAT_004074e0                                 = 20h
00401c52 8d 83 08 08 00     LEA         EAX,[EBX + 0x808]
         00
00401c58 50                 PUSH        EAX
00401c59 ff d6              CALL        ESI->KERNEL32.DLL::lstrcatA

                        LAB_00401c5b                                    XREF[1]:    00401c4b(j)
00401c5b 8d 85 d8 fe ff     LEA         EAX=>local_12c,[EBP + 0xfffffed8]
         ff
00401c61 50                 PUSH        EAX
00401c62 8d 83 08 08 00     LEA         EAX,[EBX + 0x808]
         00
00401c68 50                 PUSH        EAX
00401c69 ff d6              CALL        ESI->KERNEL32.DLL::lstrcatA
```

```
755FF640    6A              db 6A                       lstrcatA
755FF641    0868 E8         or byte ptr ds:[eax-18],ch
755FF644    16              push ss
755FF645  ^ 67:75 E8        jne kernel32.755FF630
755FF648    2C 91           sub al,91
755FF64A    0000            add byte ptr ds:[eax],al    eax:"__IRAOFF:1742194 "
755FF64C    8365 FC 00      and dword ptr ss:[ebp-4],0
755FF650    8B55 0C         mov edx,dword ptr ss:[ebp+C]
755FF653    8BF2            mov esi,edx                 esi:lstrcatA
755FF655    8A02            mov al,byte ptr ds:[edx]
755FF657    42              inc edx
755FF658    84C0            test al,al
755FF65A  ^ 75 F9           jne kernel32.755FF655
755FF65C    2BD6            sub edx,esi                 esi:lstrcatA
755FF65E    8B7D 08         mov edi,dword ptr ss:[ebp+8] edi:wsprintfA
755FF661    4F              dec edi                     edi:wsprintfA
755FF662    8D5F 01         lea ebx,dword ptr ds:[edi+1] edi+01:wsprintfA+1
755FF665    8A47 01         mov al,byte ptr ds:[edi+1]  edi+01:wsprintfA+1
755FF668    47              inc edi                     edi:wsprintfA
755FF669    84C0            test al,al
755FF66B  ^ 75 F8           jne kernel32.755FF665
755FF66D    8BCA            mov ecx,edx
755FF66F    C1E9 02         shr ecx,2
755FF672    F3:A5           rep movsd
755FF674    8BCA            mov ecx,edx
755FF676    83E1 03         and ecx,3
755FF679    F3:A4           rep movsb
755FF67B    C745 FC FEFFFFFF mov dword ptr ss:[ebp-4],FFFFFFFE
755FF682    8BC3            mov eax,ebx                 eax:"__IRAOFF:1742194 "
755FF684    8B4D F0         mov ecx,dword ptr ss:[ebp-10]
755FF687    64:890D 00000000 mov dword ptr fs:[0],ecx
755FF68E    59              pop ecx
755FF68F    5F              pop edi                     edi:wsprintfA
755FF690    5E              pop esi                     esi:lstrcatA
755FF691    5B              pop ebx
```

```
.dll:$1F640 #10640 <lstrcatA>
0098DE7C  003E1CA1  return to malware.003E1CA1 from ???
0098DE80  0098E8A4  "__IRAOFF:1742194 "
0098DE84  0098DF50  "\"__IRAFN:C:\\Users\\Student\\Desktop\\malware.exe\""
0098DE88  00000000
```

2.) The users active flag name (the username/computer name of the machine), saved as "__IRAFN:(users active flag name stored as a string)".



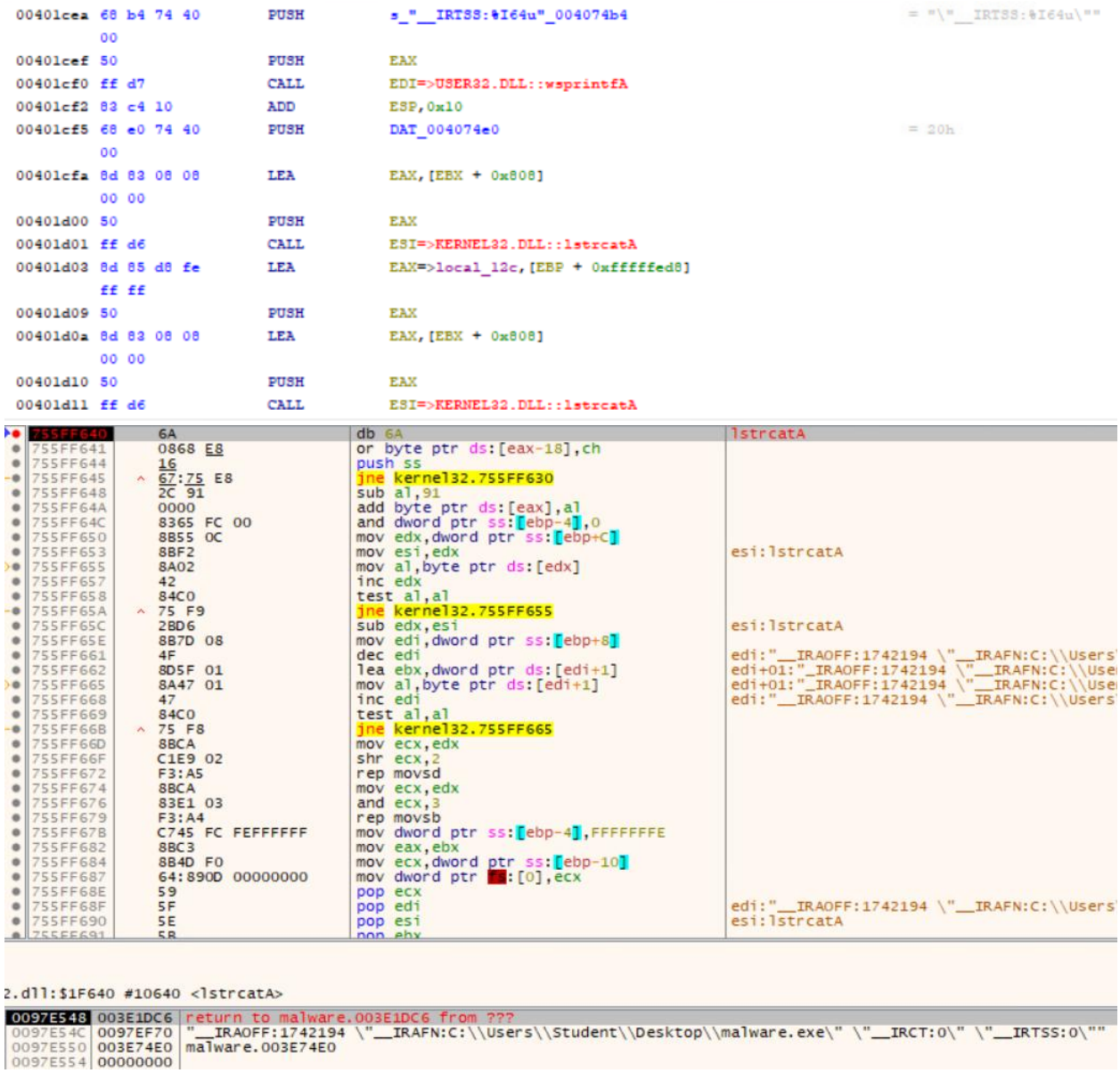
3.) The users connection type (such as speed/method), saved as "__IRCT:(users connection type stored as a string)".

```
00401caf 68 c4 74 40 00    PUSH    s_"__IRCT:%d"_004074c4                    = "\"__IRCT:%d\""
00401cb4 50                PUSH    EAX
00401cb5 ff d7             CALL    EDI=>USER32.DLL::wsprintfA
00401cb7 83 c4 0c          ADD     ESP,0xc
00401cba 68 e0 74 40 00    PUSH    DAT_004074e0                              = 20h
00401cbf 8d 83 08 08 00 00 LEA     EAX,[EBX + 0x808]
00401cc5 50                PUSH    EAX
00401cc6 ff d6             CALL    ESI=>KERNEL32.DLL::lstrcatA
00401cc8 8d 85 d8 fe ff ff LEA     EAX=>local_12c,[EBP + 0xfffffed8]
00401cce 50                PUSH    EAX
00401ccf 8d 83 08 08 00 00 LEA     EAX,[EBX + 0x808]
00401cd5 50                PUSH    EAX
00401cd6 ff d6             CALL    ESI=>KERNEL32.DLL::lstrcatA
```

```
755FF640   6A              db 6A                          lstrcatA
755FF641   0868 E8         or byte ptr ds:[eax-18],ch
755FF644   16              push ss
755FF645 ^ 67:75 E8        jne kernel32.755FF630
755FF648   2C 91           sub al,91
755FF64A   0000            add byte ptr ds:[eax],al       eax:"__IRAOFF:1742194 \"__I
755FF64C   8365 FC 00      and dword ptr ss:[ebp-4],0
755FF650   8B55 0C         mov edx,dword ptr ss:[ebp+C]
755FF653   8BF2            mov esi,edx                    esi:lstrcatA
755FF655   8A02            mov al,byte ptr ds:[edx]
755FF657   42              inc edx
755FF658   84C0            test al,al
755FF65A ^ 75 F9           jne kernel32.755FF655
755FF65C   2BD6            sub edx,esi                    esi:lstrcatA
755FF65E   8B7D 08         mov edi,dword ptr ss:[ebp+8]   edi:wsprintfA
755FF661   4F              dec edi                        edi:wsprintfA
755FF662   8D5F 01         lea ebx,dword ptr ds:[edi+1]   edi+01:wsprintfA+1
755FF665   8A47 01         mov al,byte ptr ds:[edi+1]     edi+01:wsprintfA+1
755FF668   47              inc edi                        edi:wsprintfA
755FF669   84C0            test al,al
755FF66B ^ 75 F8           jne kernel32.755FF665
755FF66D   8BCA            mov ecx,edx
755FF66F   C1E9 02         shr ecx,2
755FF672   F3:A5           rep movsd
755FF674   8BCA            mov ecx,edx
755FF676   83E1 03         and ecx,3
755FF679   F3:A4           rep movsb
755FF67B   C745 FC FEFFFFFF mov dword ptr ss:[ebp-4],FFFFFFFE
755FF682   8BC3            mov eax,ebx                    eax:"__IRAOFF:1742194 \"__I
755FF684   8B4D F0         mov ecx,dword ptr ss:[ebp-10]
755FF687   64:890D 00000000 mov dword ptr fs:[0],ecx
755FF68E   59              pop ecx
755FF68F   5F              pop edi                        edi:wsprintfA
755FF690   5E              pop esi                        esi:lstrcatA
755FF691   5B              pop ebx
```

```
dll:$1F640 #10640 <lstrcatA>
0097E548  003E1D03  return to malware.003E1D03 from ???
0097E54C  0097EF70  "__IRAOFF:1742194 \"__IRAFN:C:\\Users\\Student\\Desktop\\malware.exe\" \"__IRCT:0\""
0097E550  003E74E0  malware.003E74E0
0097E554  00000000
0097E558  0097E768
```

4.) The users terminal services session (such as local or remote desktop), saved as "__IRTSS:(users terminal services session stores as a string)".

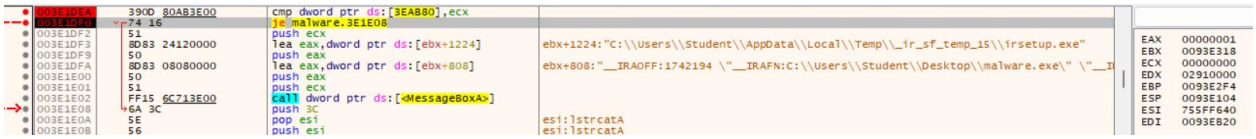


The Exfiltration Process

After the program adds the final piece to the large formatted buffer which is stored in EDI, it begins the exfiltration process to the command and control server. First, the program checks if it is running in debug mode or not by checking if DAT_0040ab80 is zero. DAT_0040ab80 being zero would mean that the program is not being ran in debug mode.

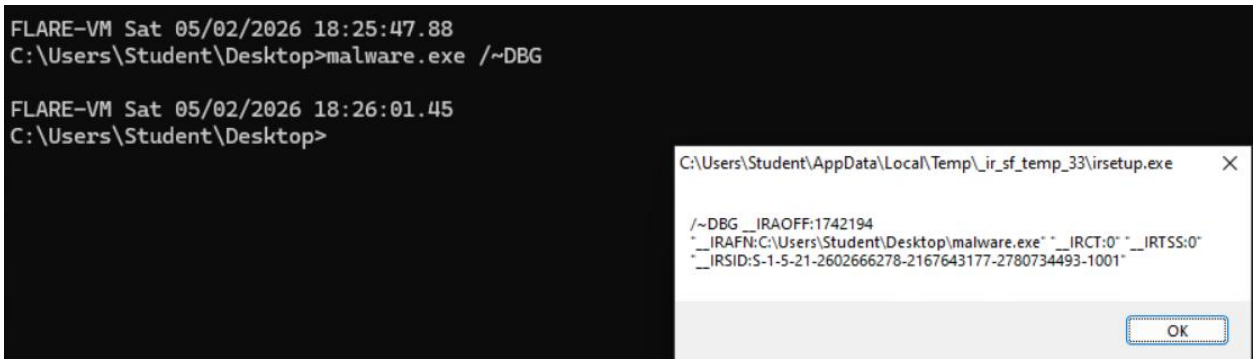
```
00401de8 33 c9           XOR        param_1,param_1
00401dea 39 0d 80        CMP        dword ptr [DAT_0040ab80],param_1
         ab 40 00
00401df0 74 16           JZ         LAB_00401e08
```

in the case below, ECX is zero, which would mean it is not being ran in debug mode and it does take the jump to LAB_00401e08.



If the program is running in debug mode, it creates a popup with that tells the developer the account infromation that the program harvested, and prints it in a message box.

```
00401dea 39 0d 80        CMP         dword ptr [DAT_0040ab80],param_1
         ab 40 00
00401df0 74 16           JZ          LAB_00401e08
00401df2 51              PUSH        param_1
00401df3 8d 83 24        LEA         EAX,[EBX + 0x1224]
         12 00 00
00401df9 50              PUSH        EAX
00401dfa 8d 83 08        LEA         EAX,[EBX + 0x808]
         08 00 00
00401e00 50              PUSH        EAX
00401e01 51              PUSH        param_1
00401e02 ff 15 6c        CALL        dword ptr [->USER32.DLL::MessageBoxA]          = 000099e8
         71 40 00
```



If the program is not running in debug mode, it attempts to launch setup. If it successfully launches setup, it jumps to LAB_00401eb7.

```
00401e73 ff 15 4c        CALL        dword ptr [->SHELL32.DLL::ShellExecuteExA]      = 00009ab4
         71 40 00
00401e79 85 c0           TEST        EAX,EAX
00401e7b 75 3a           JNZ         LAB_00401eb7
```

LAB_00401eb7 is where the command and control loop starts. First, using MsgWaitForMultipleObjects, the program presumably waits for a message back from the server signaling that it has connected. Once it has gotten a reply from the server, it jumps to LAB_00401ef0.

```
                    LAB_00401f05                                    XREF[1]:     00401ebf(j)
00401f05 68 ff 00       PUSH        0xff
         00 00
00401f0a 6a ff          PUSH        -0x1
00401f0c 57             PUSH        EDI
00401f0d 8d 85 c4       LEA         EAX=>local_140,[EBP + 0xfffffec4]
         fe ff ff
00401f13 50             PUSH        EAX
00401f14 56             PUSH        ESI
00401f15 ff 15 70       CALL        dword ptr [->USER32.DLL::MsgWaitForMultipleObj... = 00009a1c
         71 40 00
00401f1b 3b c6          CMP         EAX,ESI
00401f1d 74 d1          JZ          LAB_00401ef0
```

At LAB_00401ef0, the PeekMessageA function is used to read the message from the server. In the PeekMessageA function, local_194, which is being passed in as EAX is used to store the message received. If the message is received, the function jumps to LAB_00401ec1.

```
                    LAB_00401ef0                                    XREF[2]:     00401ed4(j), 00401f1d(j)
00401ef0 56             PUSH        ESI
00401ef1 57             PUSH        EDI
00401ef2 57             PUSH        EDI
00401ef3 57             PUSH        EDI
00401ef4 8d 85 70       LEA         EAX=>local_194,[EBP + 0xfffffe70]
         fe ff ff
00401efa 50             PUSH        EAX
00401efb ff 15 5c       CALL        dword ptr [->USER32.DLL::PeekMessageA]          = 00009a38
         71 40 00
00401f01 85 c0          TEST        EAX,EAX
00401f03 7f bc          JG          LAB_00401ec1
```

Al LAB_00401ec1, the program checks what the message from the server is.

```
                    LAB_00401ec1                                    XREF[1]:     00401f03(j)
00401ec1 83 bd 74       CMP         dword ptr [EBP + local_190],0xf
         fe ff ff 0f
00401ec8 74 0c          JZ          LAB_00401ed6
00401eca 81 bd 74       CMP         dword ptr [EBP + local_190],0x200
         fe ff ff
         00 02 00 00
00401ed4 75 1a          JNZ         LAB_00401ef0
```

The first possibility is a heartbeat from the server, in which case a jump to LAB_00401ed6 will happen. At LAB_00401ed6, the program just sends a message to the server to keep the connection.

```
                    LAB_00401ed6                                    XREF[1]:     00401ec8(j)
00401ed6 8d 85 70       LEA         EAX=>local_194,[EBP + 0xfffffe70]
         fe ff ff
00401edc 50             PUSH        EAX
00401edd ff 15 54       CALL        dword ptr [->USER32.DLL::TranslateMessage]      = 00009a5c
         71 40 00
00401ee3 8d 85 70       LEA         EAX=>local_194,[EBP + 0xfffffe70]
         fe ff ff
00401ee9 50             PUSH        EAX
00401eea ff 15 58       CALL        dword ptr [->USER32.DLL::DispatchMessageA]      = 00009a48
         71 40 00
```

The second possibility is the killswitch from the server. If the server sends 0x200, it is essentially telling the program that it has found what it needs, and to stop running, which is why there is no command after the killswitch being read.

The final possibility is any other message, in this case, the server tells the program to jump to LAB_00401ef0 and keep listening for more messages.

```
                    LAB_00401ef0                                    XREF[2]:     00401ed4(j), 00401fld(j)
00401ef0 56             PUSH        ESI
00401ef1 57             PUSH        EDI
00401ef2 57             PUSH        EDI
00401ef3 57             PUSH        EDI
00401ef4 8d 85 70       LEA         EAX=>local_194,[EBP + 0xfffffe70]
         fe ff ff
00401efa 50             PUSH        EAX
00401efb ff 15 5c       CALL        dword ptr [->USER32.DLL::PeekMessageA]          = 00009a38
         71 40 00
00401f01 85 c0          TEST        EAX,EAX
00401f03 7f bc          JG          LAB_00401ec1
```

Yara Rule for Detection

```
rule gh0stRat {
        strings:
                $debug_flag = "/~DBG"
                $temp_path = "_ir_sf_temp"

                $security_format = "__IRSID:"
                $offline_format = "__IRAOFF:"
                $active_format = "__IRAFN:"
                $conn_format = "__IRCT:"
                $terminal_format = "__IRTSS:"

        condition:
                uint16(0) == 0x5A4D and
                (4 of ($security_format, $offline_format,
                $active_format, $conn_format, $terminal_format))
                and ($debug_flag or $temp_path)
}
```

```
C:\Users\Student\Desktop>yara malware.txt malware.exe
malware malware.exe

FLARE-VM Mon 05/04/2026 16:13:29.42
```