Final Project Report
Gunnar Yonker

MiTeC Exe:

Potential malicious imports that I can check for to find the main function are CopyFileA and IsDebuggerPresent, because usually these are functions are not typically in a program. CreateFileA and _chdir are also possibly malicious. As I went through the analysis the LoadLibraryA function was also used to then additionally load the WriteFile and ReadFile functions for malicious purposes.

Name	Ordinal	Address	Delaye
CloseHandle	134	0x0000368A	
CopyFileA	168	0x000036C8	
CreateFileA	195	0x00003628	
FindClose	373	0x00003636	
FindFirstFileA	377	0x00003642	
FindNextFileA	394	0x00003654	
FreeLibrary	427	0x00003698	
GetCurrentProcess	535	0x00003B00	
GetCurrentProcessId	536	0x00003A28	
GetCurrentThreadId	540	0x00003A3E	
GetModuleHandleW	632	0x00003AEC	
GetProcAddress	686	0x000036A6	
GetStartupInfoW	720	0x00003ABE	
GetSystemTimeAsFileTime	745	0x00003A54	
InitializeSListHead	867	0x00003A6E	
IsDebuggerPresent	895	0x00003676	
IsProcessorFeaturePresent	902	0x00003AD0	
LoadLibraryA	961	0x000036B8	
QueryPerformanceCounter	1101	0x00003A0E	
SetFilePointer	1314	0x00003664	
SetUnhandledExceptionFilter	1389	0x00003AA0	
TerminateProcess	1420	0x00003B14	
UnhandledExceptionFilter	1453	0x00003A84	

Ghidra:

I did my primary analysis in OllyDbg, but I did use Ghidra to help in locating the main function.

Final Project Report
Gunnar Yonker

This is the start of the main function that I then looked for in OllyDbg:

```
                    undefined4 __stdcall main(void)
                      assume FS_OFFSET = 0xffdff000
        undefined4      EAX:4          <RETURN>
        undefined4      Stack[-0x8]:4 local_8                    XREF[1]:     00401156(W)
                    main                                       XREF[1]:     FUN_0040133c:00401431(c)
   00401144 55          PUSH    EBP
   00401145 8b ec       MOV     EBP,ESP
   00401147 51          PUSH    ECX
   00401148 53          PUSH    EBX
   00401149 56          PUSH    ESI
   0040114a 57          PUSH    EDI
   0040114b 68 38 40    PUSH    s_Kernel32.dll_00404038                  = "Kernel32.dll"
            40 00
   00401150 ff 15 24 30 CALL    dword ptr [->KERNEL32.DLL::LoadLibraryA]  = 000036b8
            40 00
   00401156 89 45 fc    MOV     dword ptr [EBP + local_8],EAX
   00401159 66 60       PUSHA
   0040115b b9 00 00    MOV     ECX,0x0
            00 00
```

OllyDbg:

For the structure of this analysis, I will do the first part of the analysis of the program before the two different branches are taken depending on if a debugger is present or not. Then I will analyze the path of if a debugger is present and then the path of if a debugger is not present.

The program starts out pushing some values to the registers and then with a call to LoadLibraryA using the parameter “Kernel32.dll”. This call will load the given library and return a value that is the handle to the module if successful, otherwise the return value is null. In this case, the value is then moved from the eax register to EBP-4.

```
00B9114B | . 68 3840B900      PUSH    sampleX.00B94038          FileName = "Kernel32.dll"
00B91150 | . FF15 2430B900    CALL    DWORD PTR DS:[<&KERNEL32.LoadLibraryA>]  LoadLibraryA
00B91156 | . 8945 FC          MOV     DWORD PTR SS:[EBP-4], EAX  KERNEL32.77710000
EAX=77710000 (KERNEL32.77710000)
Stack SS:[010FF868]=00000000
```

Then a loop is entered where 0 is put into the eax register and then incremented each loop until it reaches the hex value of 10. Each iteration of this loop is modifying the path value for the upcoming _chdir call which at the first loop is “F=2Xvhuv2Sxeolf2” and I believe that this loop is decrypting this string one byte at a time by subtracting a hex value of 3, with the ultimate end being the destined path.

```
CMP     ECX, 10
JE      SHORT sampleX.00B91177
LEA     ESI, DWORD PTR DS:[ECX+B94008]
MOV     AX, WORD PTR DS:[ESI]
SUB     AX, 3
MOV     BYTE PTR DS:[ESI], AL
INC     ECX
JMP     SHORT sampleX.00B91160
```

For example, the first iteration of the loop takes the first Byte “F” and replaces it with the AL register that contains the ASCII value “C”. This is because the original hex value is 46, then it subtracts 3 and the result

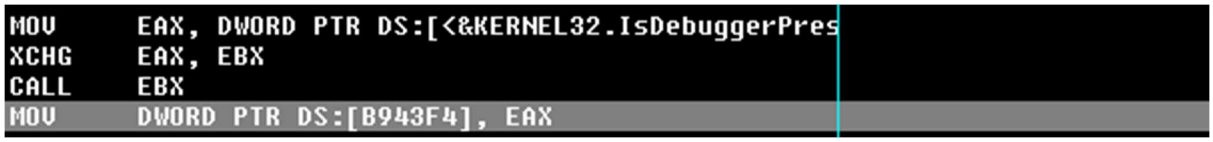
is a hex value of 43 which is ASCII “C”. So the path value after the first loop is “C=2Xvhuv2Sxeolf2” and the address in 00B94009 is now “=2Xvhuv2Sxeolf2”.

The next iteration then subtracts the hex value of 3 from 3D, ASCII “=”, and it becomes hex value 3A, ASCII “:”. This byte is then replaced in the path variable, so it becomes “C:2Xvhuv2Sxeolf2”.

This process of subtracting a hex value of 3 on each byte continues until the eax register has a hex value of 10 and then the jump inside of the loop is taken to exit the loop. The path variable at the end of the loop iterations becomes “C:/Users/Public/”.

This is the parameter that is used in the call to _chdir which uses the chdir command to change the directory of the cmd window into the path given, which is C:/Users/Public/”. Then that path value is popped off of the stack.

Next is the call to IsDebuggerPresent, which the program tries to hide by moving the call itself into the eax register, then moving it from the eax register to the ebx register, then finally making the call to ebx which is a call to IsDebuggerPresent. This call with return a nonzero value if there is a debugger present or return a zero value if there is not a debugger present. This value is then stored, in this case it is stored as a global variable in DS:[B943F4]. OllyDbg does not detect this call because it is hidden in the ebx register.



The next call is to GetProcAddress with the parameters for the function name, ReadFile, and the handle to the DLL module that it is contained in which is the value that was returned from the LoadLibraryA call. This returns the address to the exported function if successful or returns a null value if it fails. In this program, the value is then stored in DS:[B943F0] which is KERNEL32.ReadFile.

Another call to GetProcAddress is made with the same handle to the DLL module from LoadLibraryA, but this time for the function WriteFile. The return value is stored in DS:[B943EC] which is KERNEL32.WriteFile.

```
PUSH    sampleX.00B94054                        ProcNameOrOrdinal = "WriteFile"
PUSH    DWORD PTR SS:[EBP-4]                    hModule
CALL    DWORD PTR DS:[<&KERNEL32.GetProcAddress>]  GetProcAddress
MOV     DWORD PTR DS:[B943EC], EAX              KERNEL32.WriteFile
```

Now the program starts to compare the values that were stored to see if these calls were successful or not. The first value checked is the value returned from GetProcAddress for ReadFile, jump is taken if it was unsuccessful. The second value checked is the value returned from GetProcAddress for WriteFile, jump is taken if it was successful. Then the third value checked if the value from IsDebuggerPresent which depending on the value, changes the course of the program. If a debugger is present the program takes a path involving a .jpg search, if there is no debugger present the program takes a path involving a *.d* search.

```
CMP     DWORD PTR DS:[B943F0], 0
JE      SHORT sampleX.00B911CD
CMP     DWORD PTR DS:[B943EC], 0
JNZ     SHORT sampleX.00B911D2
OR      EAX, FFFFFFFF
JMP     SHORT sampleX.00B911F2
CMP     DWORD PTR DS:[B943F4], 1
JNZ     SHORT sampleX.00B911E2
CALL    sampleX.00B91000
```

First, I will analyze the path for if a debugger is present, then the jump over this call is not taken, and the call to this function is made.

For this path, the first call is FindFirstFileA which takes the parameters lpFileName, “*.jpg”, and a pointer to a structure that will receive the information about the found file which in this case looks like EBP-254. The value returned will be a search handle to be used in further calls if successful, or if unsuccessful the return will be ERROR_FILE_NOT_FOUND.

```
PUSH    EAX                                     pFindFileData
PUSH    sampleX.00B9401C                        FileName = "*.jpg"
CALL    DWORD PTR DS:[<&KERNEL32.FindFirstFileA>]  FindFirstFileA
MOV     DWORD PTR SS:[EBP-4], EAX
```

Next a call to CopyFileA is made with a few parameters. First the parameter lpExistingFileName which is the name of an existing file and the name being used is “sampleX.exe”, the name of the program. The parameter lpNewFileName, which is the name of the new file, in this case the program is using the name of the file that was found in the previous call. The last parameter is bFailIfExists which is set to True, this means that if the new file name already exists the function will fail. I thought that this was interesting because how it is set up it will always fail because it is using the name of the file that was already found, so the file exists. I think the intent was to have the malware program propagate into pretending it was an existing jpg image on the victim’s system so that it could potentially be executed at a later date, but this would require the bFailIfExists parameter to be False so that the file could be overwritten.

Final Project Report
Gunnar Yonker

```
PUSH    1                                       ┌FailIfExists = TRUE
LEA     EAX, DWORD PTR SS:[EBP-228]             │
PUSH    EAX                                     │NewFileName
PUSH    sampleX.00B94024                        │ExistingFileName = "sampleX.exe"
CALL    DWORD PTR DS:[<&KERNEL32.CopyFileA>]    └CopyFileA
```

Then the program clears the eax register, increments it by 1, and returns to the main function.

Before discussing the end of the main function, I will analyze the second path of the program that will be taken if no debugger is present.

If no debugger is present, the jump is taken over the previous call path and a new call path is taken.

```
CMP     DWORD PTR DS:[B943F4], 1
JNZ     SHORT sampleX.00B911E2
CALL    sampleX.00B91000                        jpg malware function
JMP     SHORT sampleX.00B911E7
CALL    sampleX.00B91037                        call to malware purpose, succes in loading and no debugger present
```

Similar to the other path, the first call FindFirstFileA where the FileName parameter is “*.d*” and the pointer to where to store the information about a found file is EBP-264.

```
LEA     EAX, DWORD PTR SS:[EBP-264]             ┌pFindFileData
PUSH    EAX                                     │FileName = "*.d*"
PUSH    sampleX.00B94030                        └FindFirstFileA
CALL    DWORD PTR DS:[<&KERNEL32.FindFirstFileA>]
```

It is then checked if a file was found or not before proceeding, if a file was not found the jump is not taken and the eax register is cleared before jumping out of this function. If a file was found a jump into a loop is made.

```
MOV     DWORD PTR SS:[EBP-C], EAX
CMP     DWORD PTR SS:[EBP-C], -1
JNZ     SHORT sampleX.00B91069
XOR     EAX, EAX
JMP     sampleX.00B9113F
```

The first call is this loop is CreateFileA with a few different parameters. The FileName, which is used from the FindFirstFileA call, dwDesiredAccess which is GENERIC_READ|GENERIC_WRITE, dwShareMode which is 0 and means that other processes are prevented from accessing or deleting the file until the handle is closed, lpSecurityAttributes is null which sets a default security descriptor, dwCreationDisposition is OPEN_EXISTING which means it will only open a file that exists otherwise produce an error, dwFlagsandAttributes is NORMAL which is the common default, and the final parameter hTemplateFile is null and can be ignored because this is an existing file. The return value is a handle to the specified file if successful, otherwise an error is returned.

```
┌PUSH    0                                       ┌hTemplateFile = NULL
│PUSH    80                                      │Attributes = NORMAL
│PUSH    3                                       │Mode = OPEN_EXISTING
│PUSH    0                                       │pSecurity = NULL
│PUSH    0                                       │ShareMode = 0
│PUSH    C0000000                                │Access = GENERIC_READ|GENERIC_WRITE
│LEA     EAX, DWORD PTR SS:[EBP-238]             │
│PUSH    EAX                                     │FileName
│CALL    DWORD PTR DS:[<&KERNEL32.CreateFileA>]  └CreateFileA
```

Final Project Report
Gunnar Yonker

The next call made is ReadFile which was obfuscated in the code, but looking at the variable that is being called to we can notice that before that it is the variable that was returned from GetProcAddress for ReadFile. This call is going to use the file handle returned from CreateFileA to read through the existing file that has been found and now opened.

```
00B9108B  . 6A 00          PUSH    0                                lpoverlapped value
00B9108D  . 8D45 F0        LEA     EAX, DWORD PTR SS:[EBP-10]       number of bytes actually read
00B91090  . 50             PUSH    EAX
00B91091  . 68 00100000    PUSH    1000                             bytes to read
00B91096  . 8D85 9CEDFFFF  LEA     EAX, DWORD PTR SS:[EBP-1264]     where buffer is located
00B9109C  . 50             PUSH    EAX                              buffer
00B9109D  . FF75 F8        PUSH    DWORD PTR SS:[EBP-8]             File handle
00B910A0  . FF15 F043B900  CALL    DWORD PTR DS:[B943F0]            ReadFile
00B910A6  . 8365 FC 00     AND     DWORD PTR SS:[EBP-4], 0
DS:[00B943F0]=77733B30 (KERNEL32.ReadFile), JMP to KERNELBA.ReadFile
```

If the file is not empty, then another loop is entered. There are no direct calls made in this loop, but this loop has a large impact on what is done to the file itself. I am not sure what is specifically going on and how the encryption is taking place, but it seems that this loop reads one byte at a time and performs a bitwise XOR operation on that byte before storing it back into the buffer at the same position and moving onto the next byte. Essentially, this loop encrypts the contents of the file one byte at a time. Once there are no more bytes detected the loop then exits and continues to the next call.

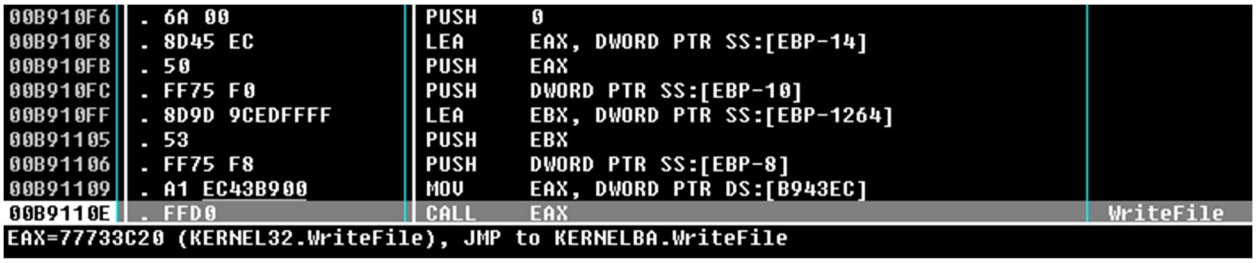
```
MOV     EAX, DWORD PTR SS:[EBP-4]          loop if file is not empty
INC     EAX
MOV     DWORD PTR SS:[EBP-4], EAX
MOV     EAX, DWORD PTR SS:[EBP-4]
CMP     EAX, DWORD PTR SS:[EBP-10]
JNB     SHORT sampleX.00B910E5
MOV     EAX, DWORD PTR SS:[EBP-4]
MOVSX   ECX, BYTE PTR SS:[EBP+EAX-1264]
MOV     EAX, DWORD PTR SS:[EBP-4]
XOR     EDX, EDX
PUSH    8
POP     ESI
DIV     ESI
MOVSX   EAX, BYTE PTR DS:[EDX+B94000]
XOR     ECX, EAX
MOV     EAX, DWORD PTR SS:[EBP-4]
MOV     BYTE PTR SS:[EBP+EAX-1264], CL
JMP     SHORT sampleX.00B910AC
```

After the loop the next call made is SetFilePointer which uses the following parameters: hFile which is the handle to the file from the previous call, lDistanceToMove set to 0, lpDistanceToMoveHigh which is set to null, and the dwMoveMethod which is set to FILE_BEGIN so that it starts at the beginning of the file. The return value will be the low-order DWORD of the new file pointer, otherwise if it fails the return is an error value. This sets the file pointer to the beginning of the file that is being read.

```
PUSH    0                                  Origin = FILE_BEGIN
PUSH    0                                  pOffsetHi = NULL
PUSH    0                                  OffsetLo = 0
PUSH    DWORD PTR SS:[EBP-8]               hFile
CALL    DWORD PTR DS:[<&KERNEL32.SetFilePointer>]   SetFilePointer
```

There is another call here that has been hidden much like the ReadFile call that was previously used. This time it is the WriteFile call that was returned from the GetProcAddress for the WriteFile function. The parameters are pushed and then KERNEL32.WriteFile is moved to the eax register and called from the

eax register. WriteFile takes the handle to the file, the pointer to the buffer containing the data that is going to be written which in this case is the buffer that now contains the encrypted data from the loop, the number of bytes to be written, the pointer to the variable that receives the bytes. The return value upon success is nonzero, and it is zero if it fails. This call is writing the buffer contents that were encrypted from the previous loop into the file that is open. This call is overwriting what was originally in the file with the encrypted contents of the buffer.



The next call is CloseHandle which takes the handle to the object that is currently open, in this case the file handle that is open, and then closes it. This means that the file can now be opened by the user or other processes. The return value is nonzero on success, on failure the return value is 0.

A call to FindNextFileA is made using the search handle and a pointer to the structure to receive the information. This call is used to see if there is another file in the current directory that has the “*.d*” search parameter. If there is another file found, then the pointer location will contain the information about that file. A zero is returned if there are no more files found. Additionally, if another file is found, a jump back to the initial loop with CreateFileA is made, then the encryption loop is taken where the file contents will be encrypted and then overwritten, leading to another FindNextFileA call. Once no more files are found, then the next call is made.

If there are no more files found, then a call to FindClose is made with the search handle, this call closes the search handle. If successful it returns a nonzero value, and upon failure returns a zero. After this call, a return to the main function is made after clearing the eax register, incrementing it by 1, and popping off a few values into the registers.

This is the final part of the analysis where both paths rejoin, the debugger present and the no debugger present paths will return to this part of the main function.

A call to FreeLibrary is made using the only parameter hLibModule which is the handle to the loaded library module, in this program it is the Kernel32.dll library that was loaded.

The eax register is cleared and a few more values are popped off the stack before the program gets to the next and final call.

A call to exit is made with the status parameter being 0, this means that everything was successfully executed, and the program terminates.

Conclusion:

After analysis, this program does indeed seem to be a malicious malware program that if taking place on your system can cause substantial damage. The program starts out with a LoadLibraryA call that loads the Kernel32.dll module. Next the program enters a loop to decrypt a stored variable value, in ASCII “"F=3Xvhuv2Sxeolf2", by subtracting a hex value of 3 from each byte before replacing it in the string. The loop increments the eax register for each iteration of the loop until the eax register hex value is 10, resulting in the string being decrypted, represented in ASCII as "C:/Users/Public/" which is used for the next call. The next call is _chdir and uses the path mentioned above to change the directory of the cmd window into the "C:/Users/Public/" directory. The first anti-debugging technique is used here where a call to IsDebuggerPresent is made to see if the program is being executed within a debugger. The value isn’t used right away so it is first stored as two additional calls are made. Both calls are GetProcAddress, one for ReadFile and the other for WriteFile, both are then stored as global variables. This call loads the ReadFile and WriteFile functions from the loaded library to be used later. Now, the values are checked to see if they are successful otherwise jumps are taken accordingly. The most important jump distinction here is if the debugger call return value is 0 or 1.

If the debugger value is 1, meaning there is a debugger present, a jump to a different function is taken. A call to FindFirstFileA is used to look for any .jpg files in this directory, if one is found then a call to CopyFileA is made which attempts to copy the “sampleX.exe” file and name it as the .jpg file that was found. However, the flag for overwriting is TRUE so this call ultimately will fail every time it runs. I think that the initial intention of this was to find a .jpg file, copy the malware executable and rename it as the found .jpg file so that the infected system would potentially try to open that .jpg image later causing the malware to run again. I believe it was supposed to be a propagation technique for if a debugger was detected, but that the flag may have been incorrectly set.

If the debugger value is 0, meaning there is no debugger present, a different call is made to a function where the true purpose of the malware is seen. A call to FindFirstFileA is made to look for any files that contain “*.d*” and if there is a file found then a call to CreateFileA is made. An important parameter in this call is that it is opened with GENERIC_READ|GENERIC_WRITE access and share mode of 0 so no other process can open it while the handle is open. Another obfuscation technique is used here where the call to ReadFile is hidden by calling the global variable location of ReadFile that was loaded so that it is not directly detected by the debugger. The ReadFile call is made, where the contents of the file are read and stored into a buffer. This buffer is then used in an encryption loop where the contents of the buffer are encrypted byte by byte using what I think is an XOR bitwise encryption algorithm before being replaced in the same spot of the buffer. So, the original contents in the buffer are

now encrypted. A call is made to SetFilePointer to set the starting point at the beginning of the file that is opened. Then the same obfuscation technique that was used for ReadFile is used to hide the use of the WriteFile call. This is a very important call because it now overwrites the opened file with the contents of the buffer used from the encryption loop, so the contents of the file are overwritten with the encrypted data. The handle is then closed using CloseHandle and FindNextFileA is used to see if there is another file in the directory. If there is, then the loop restarts and encrypts the content. If there is no other file found, then the search handle is closed using FindClose and the program returns to the main function.

At this point in the program the same outcome takes place regardless of a debugger being present or not, a call to FreeLibrary is made which closes the handle for the Kernel32.dll library that was loaded. The final call is a call to exit with a status code indicating if there was a success with a return value of 0, which this call then terminates the program.

The obfuscation techniques used by this malware were indirect calls to loaded functions such as ReadFile and WriteFile where they were loaded from the library using GetProcAddress and then stored into a global variable that was then called later in the program. The call to IsDebuggerPresent was also hidden by putting it into a register, moving it to a different register, and then calling that register. Both of these techniques made it so that a debugger like OllyDbg would not be able to automatically detect that these calls were being made.

In summary I think that this malware program checks for a debugger to be running, if there is a debugger detected it tries to clone itself into the C:/Users/Public/" directory disguised as a .jpg image that was previously in there so that it could be potentially executed again. If there is no debugger found, it checks for any files with “*.d*” such as a .docx file and then encrypts the contents of that file overwriting what was previously contained in it. It does this until no more files are found and then the program closes all the handles, frees the library, and then terminates. This would leave the victim of the malware program with their files in that directory encrypted, example shown below.

Final Project Report
Gunnar Yonker

```
QF
  
  
"H¤]   
 
 
   
 
 
Q|da}%!ykbr€ 1 É> E¢lg@=^.…ïªKGb~ ÆÐ2^é bò¶3$' ¬® àÂÑ-¡w-
„Ð&ÎÆR Õ£  H\5>WÃàß'= cä¤¢Hªæ‡èê-Ê  3ƒ"÷-c™-  éÇçÆî^ "Å®Ð 7 B³%
Å<‹A
£'¢ @^ìzxJ¯|ógÆ¬-èŽ»‹Ž' Ø¡Ò£~wôí¼ ^GÓä ð^ëÅ
æ ]<à ^J
 E„Ö2¨ 
   
 ^J 
  L¡Y
   
 
 
     
 |a|e"Qxjg}!ebm bn`z/ucf!ykbr ŽÁ É> Dr+ÆÑ{/#@µ ®Ð\1!+Ý
9 œZíÒ  Ë Œ -ÎÂ·è;wwT.·ªü^ß ³)
^U 7æ Ì¿3£½^ x€- þÉù ô)H¾hÛ‰nv™tÛ¬é0? xÒ¾É" ìé7d³7ø ç;@9Þ²}
Î¼'9Ò  æGœÝKr  ŽZ"Œ:¨¡oúdg†±ZBÐ
öKB zó ©N  ðJÏ‰wEò`›òØì ¹&ÞÏŒ€ ^J
 Öü
Î¸
 œ
 ^J 
  L¡Y
   
 
 
    
 |a|e"jel~ckoy rbgãXÀc : ö šú Ä®aI«[4¤© yÓ0Î

< +¹Ý¼÷æ™ZgÜgéì=  0´2ÕÈžn³ñc{|'EÄ  Æi'J© ¼[ ÖÚæøäž.êÏ•£›
0DlêØ¼ÁD ø£53  ‰ù nŒú í"O-U¡"{Ø‹Wk¿  L¦vl¿zi^* õ¦èW›♯v Å»&·²è?
¤¡t²n]Å  Ó/jk'  ž °  $.¼nkp‡°„¬h  ™û¶G˜?®¢åÍ r,¦ÿ¿]Cð·Þœ öx þ
é KÔ é·  ÝWí♯™/¯ZÀ"N¡  ¦Úm Ó( eL å:ªê ÓE ª
€/näãŽ^Šã9lpÀœóìp´ì  Ÿ- ¯¬ãLÚ àŒ/¾
çGr €QÄ ôÍ¶\ÓU¿    Ð )‡
Æ]@"O=Áä,Ø'÷æÎ—¦7Rÿy¬¥c$ƒ ¬w›Cx-ãóÉxõ-¢
ðˆ ÷ù¼èÝ ŠÛ)QF   ž˜•'k
 
C   QF
  
  
"H¤]   
 
 
   
 
 
yasi!dzflksd`m!scb
 O Í;
¾ ÷DR3. 4à£%
ŠÊµ˜ ±… Ñ'îdly"°g6 Ä° 06y< ì*~èËÏâ3'do¿v€  (
0Y-È Wp(î¨—÷Íd¡Œ¾šÍ♯gV♯væê.oá+3öÙhP±G€M™K¤£M6Pðñ Ça
```