This is a guide to how I unpacked an FSG executable. There are easier ways, such as finding OEP manually or by using [Joe Stewart's OllyBonE](). In this case, I'm not so interested in the un-packing algorithm per se, but more of what memory regions are utilized in the process. The binary being analyzed is an executable discussed in a post on [Spyware Warrior Forum]().

First it uses an xchg instruction to swap esp with the dword at offset 4094E8h. Also see below the value here is 4094CCh. This effectively points the program's stack pointer into the program's own segment where it has hard-coded additional data. Note: the second xchg restores the original stack…but leaves esi and edi in tact (read on to see why this is significant).

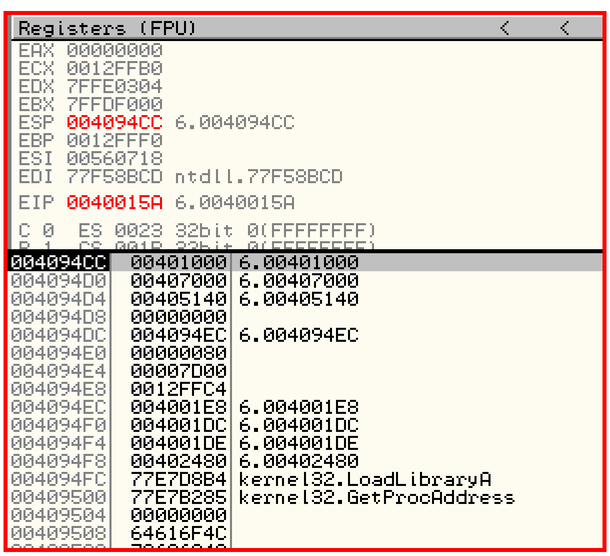
```
HEADER:00400154 start           proc near
HEADER:00400154                 xchg    esp, ds:off_4094E8
HEADER:0040015A                 popa
HEADER:0040015B                 xchg    eax, esp

seg002:004094E8 off_4094E8      dd offset off_4094CC    ; DATA XREF:
startw
```

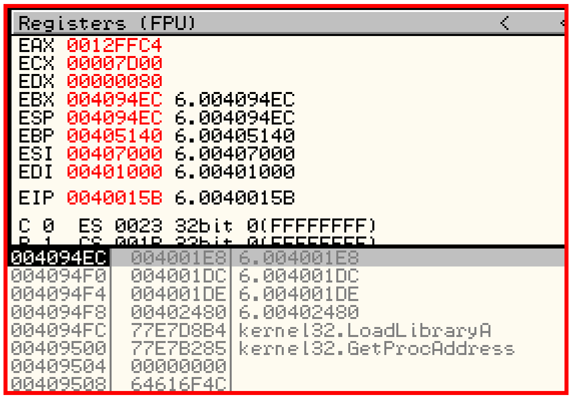
Take a look at what is at 4094CCh in the binary:

```
seg002:004094CC off_4094CC      dd offset dword_401000  ; DATA XREF:
seg002:off_4094E8o
seg002:004094D0                 dd offset unk_407000
seg002:004094D4                 dd offset dword_401000+4140h
```

Notice how after the xchg instruction, these values are on the top of the program's stack, as well as a bunch of other junk



Now take a look at the registers, esi and edi in particular, after the popa:



So, edi (401000h) and esi (407000h) are the addresses of seg001 and seg002 sections, respectively. See how much the file is packed by focusing on the “dd 1800h dup” instruction inside seg001.

```
seg001:00401000 ; Segment type: Pure code
seg001:00401000 ; Segment permissions: Read/Write
seg001:00401000 seg001          segment para public 'BSS' use32
seg001:00401000                 assume cs:seg001
seg001:00401000                 ;org 401000h
seg001:00401000                 assume es:nothing, ss:nothing,
ds:seg001, fs:nothing, gs:nothing
seg001:00401000 dword_401000    dd 1800h dup(?)         ; DATA XREF:
seg002:off_4094CCo
```

When unpacked, we can expect seg001 to be 1800h (6144 dec) bytes long. Seg002 virtual size is 3600h (12288 dec) – exactly twice the size of seg001. Looking a little further into the start code, you can see data taken from esi and moved into edi – this happens pretty much right after the popa above and then enters a loop where esi and edi increment while the bytes are read from esi, processed, and written to edi.

```
HEADER:0040015D                 movsb
HEADER:0040015E                 mov     dh, 80h
```

It follows this pattern until seg001 contains all the DLL names and the names of the exports in those DLLs that it wants to use, then calls LoadLibrary() and GetProcAddress() to resolve them and stores the function addresses to the remaining vacant space in seg001.

It loops and resolves these functions:

wsprintfA
InternetCloseHandle
InternetGetConnectedState
InternetOpenA
InternetOpenUrlA
InternetQueryDataAvailable
InternetReadFile

CloseHandle
CopyFileA
CreateEventA
CreateFileA
CreateMutexA
CreateThread
DeleteFileA
EnterCriticalSection
ExitProcess
ExitThread
FreeConsole
GetLastError
GetModuleFileNameA
GetModuleHandleA
GetSystemDirectoryA
GetThreadContext
GetTickCount
GetVersionExA
GlobalAlloc
GlobalFree
InitializeCriticalSection
LeaveCriticalSection
LoadLibraryA
ReleaseMutex
ResumeThread
SetEvent
SetFileAttributesA
SetThreadContext
Sleep
TerminateProcess
VirtualAllocEx
WaitForSingleObject
WaitForSingleObjectEx
WinExec
WriteFile
WriteProcessMemory
lstrcatA
lstrcpyA
lstrlenA
WSACleanup
WSAStartup
__WSAFDIsSet
accept
closesocket
connect
gethostbyname

gethostname
getsockname
htons
inet_ntoa
listen
ntohs
recvfrom
select
sendto
socket
URLDownloadToCacheFileA
URLDownloadToFileA
RegCloseKey
RegOpenKeyExA
RegQueryValueExA
RegSetValueExA

Also to do this, it loads these DLLs (and a few others not listed too):

user32.dll
wininet.dll
ws2_32.dll
urlmon.dll
advapi32.dll

To further unpack without wasting tons of time figuring out when the resolution loop finishes, locate one of the exports that would reasonably be called before any of the real malicious behavior starts (WSAStartup, InitializeCriticalSection, and WinExec are good choices). Set a breakpoint at the beginning of each function inside the DLL that exports them using OllyDbg and let the program run. You should be able to catch it red-handed doing bad things.