INSIDE THE ULTIMA ONLINE GOLD DEMO
- THE PACKET COMMUNICATION – PART 1

GOAL

It's our goal to get a deep understanding of how the Ultima Online Gold Demo works. This demo is a representation of the rule set from the Ultima Online Second Age Era.

There is proof that some people have already reversed this demo partially or as a whole, however so far no tools or knowledge has been published. This project is to overcome does shortcomings.

URL's with some proof for this:
http://www.runuo.com/forums/general-discussion/94767-help-m-files.html
http://azaroth.org/2008/12/31/your-topic/ (posting by Faust)

If we understand the demo there is a big chance we can alter the demo and even create our own demo. By default mounting horses is not possible in the demo, but what if we can alter the demo and unlock horses; can we then see how horses behaved during T2A?

This demo is 10 years old and I do not understand no one published his/her work. Maybe that DMCA thing is in the way?

UTILITIES USED

IDA Pro, a very professional utility, definitely worth buying, Standard version is affordable.
HxD, a very neat hex editor and above all, it's free
Explorer Suite, it did the job for this project but the tool can be improved

ABOUT ME

I'm just a guy who loves the Ultima universe and knows a bit assembler. Why not combine the two? ☺ I really enjoy programming and I've done things in many languages. But there is one language which I really dislike and I never ever want to use it again, let that monster be named Java. Not sure why I don't like it but it must be the first experiences I had with it, Java 1.0, I remember it being slow and I had no integrated editor for it. First impressions count and it's probably my fault that I can't get over that first experience.

THEORY

If this demo is based on a real server and a real client than the protocol used to send data between the client and server will be the same. The Ultima Online Protocol has been well reversed and used to create custom servers and even custom clients (http://en.wikipedia.org/wiki/Ultima_Online_shard_emulation).

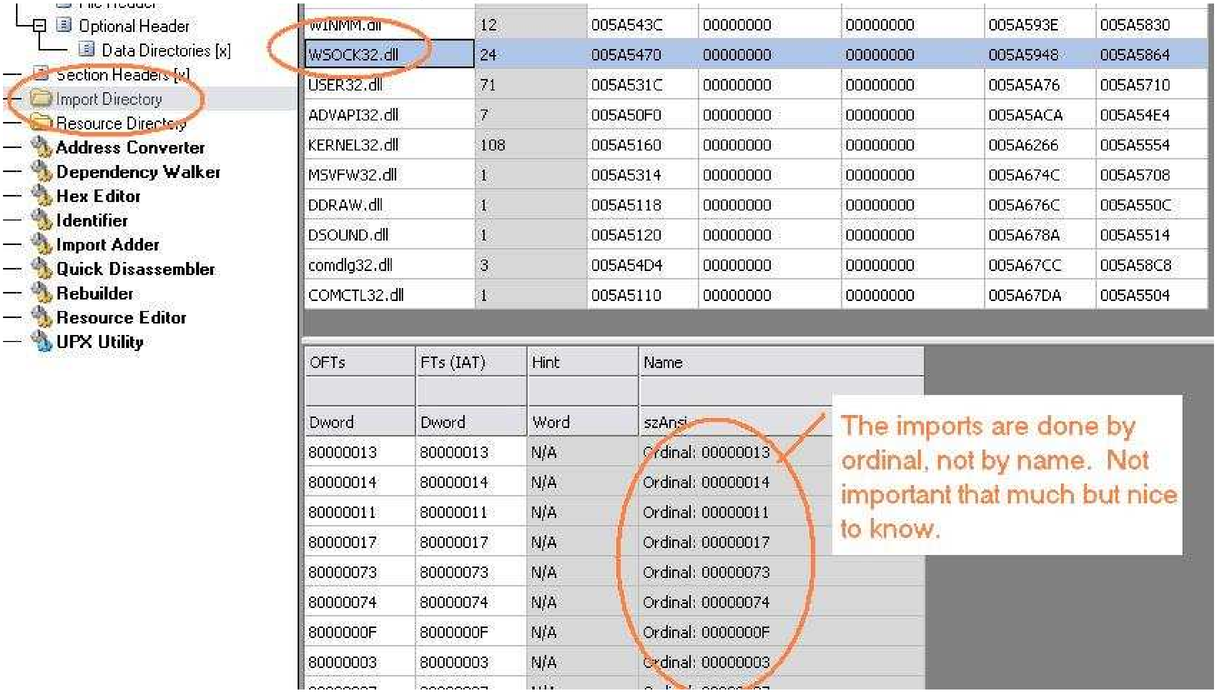
So, if we can find out how the communication is done we could implement a patch to peek at the data communication and this will prove or disprove that the communication is equal (or similar) to the already reversed protocol.

It's important to note that the communication between the official client and the official server is being encrypted and that the protocol itself has seen some improvements over the years.

HOW TO LOOK

Now, where do you start to find such code? If you open the UoDemo with the Explorer Suite you can look at the import table, you will find the WSOCKS API is being imported, however when you run the demo and you use the "netstat" command to look for active sockets you will find that UoDemo.EXE is not actively using the WinSock API. This suggests that the API is there because of the original code but that another method is used to send the data between client and server.

Screenshot of UoDemo.exe using Explorer Suite III:



WINMM.dll	12	005A543C	00000000	00000000	005A593E	005A5830
WSOCK32.dll	24	005A5470	00000000	00000000	005A5948	005A5864
USER32.dll	71	005A531C	00000000	00000000	005A5A76	005A5710
ADVAPI32.dll	7	005A50F0	00000000	00000000	005A5ACA	005A54E4
KERNEL32.dll	108	005A5160	00000000	00000000	005A6266	005A5554
MSVFW32.dll	1	005A5314	00000000	00000000	005A674C	005A5708
DDRAW.dll	1	005A5118	00000000	00000000	005A676C	005A550C
DSOUND.dll	1	005A5120	00000000	00000000	005A678A	005A5514
comdlg32.dll	3	005A54D4	00000000	00000000	005A67CC	005A58C8
COMCTL32.dll	1	005A5110	00000000	00000000	005A67DA	005A5504

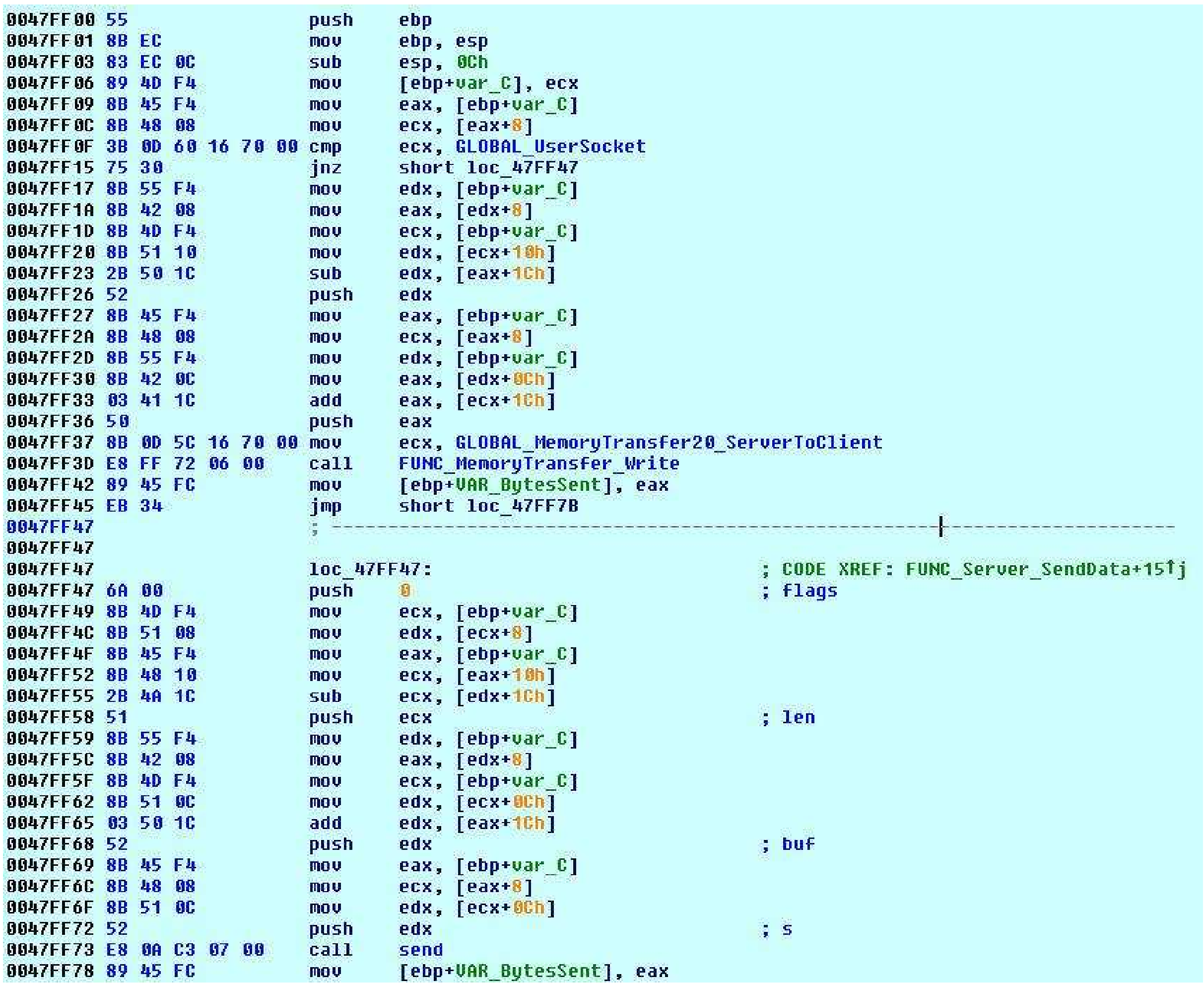
OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
80000013	80000013	N/A	Ordinal: 00000013
80000014	80000014	N/A	Ordinal: 00000014
80000011	80000011	N/A	Ordinal: 00000011
80000017	80000017	N/A	Ordinal: 00000017
80000073	80000073	N/A	Ordinal: 00000073
80000074	80000074	N/A	Ordinal: 00000074
8000000F	8000000F	N/A	Ordinal: 0000000F
80000003	80000003	N/A	Ordinal: 00000003

The imports are done by ordinal, not by name. Not important that much but nice to know.

There are many methods a programmer can use to send data internally. The most logical choice is by using memory. If threads are used then we should find references to critical sections. If you don't know what critical sections are then stop reading now and study first. I'm serious; every programmer must now what critical sections are. http://en.wikipedia.org/wiki/Critical_sections

SEND USING SOCKET VERSUS SEND USING MEMORY

I'm not going to show you how I located the code but it isn't that hard to find once you know what to look for. On the following screenshot you can see that the UoDemo supports sending to a socket and also sending to a memory object. However, the "send" call (at the bottom) is never reached. Dream question: can we add code that will make it reach this point?

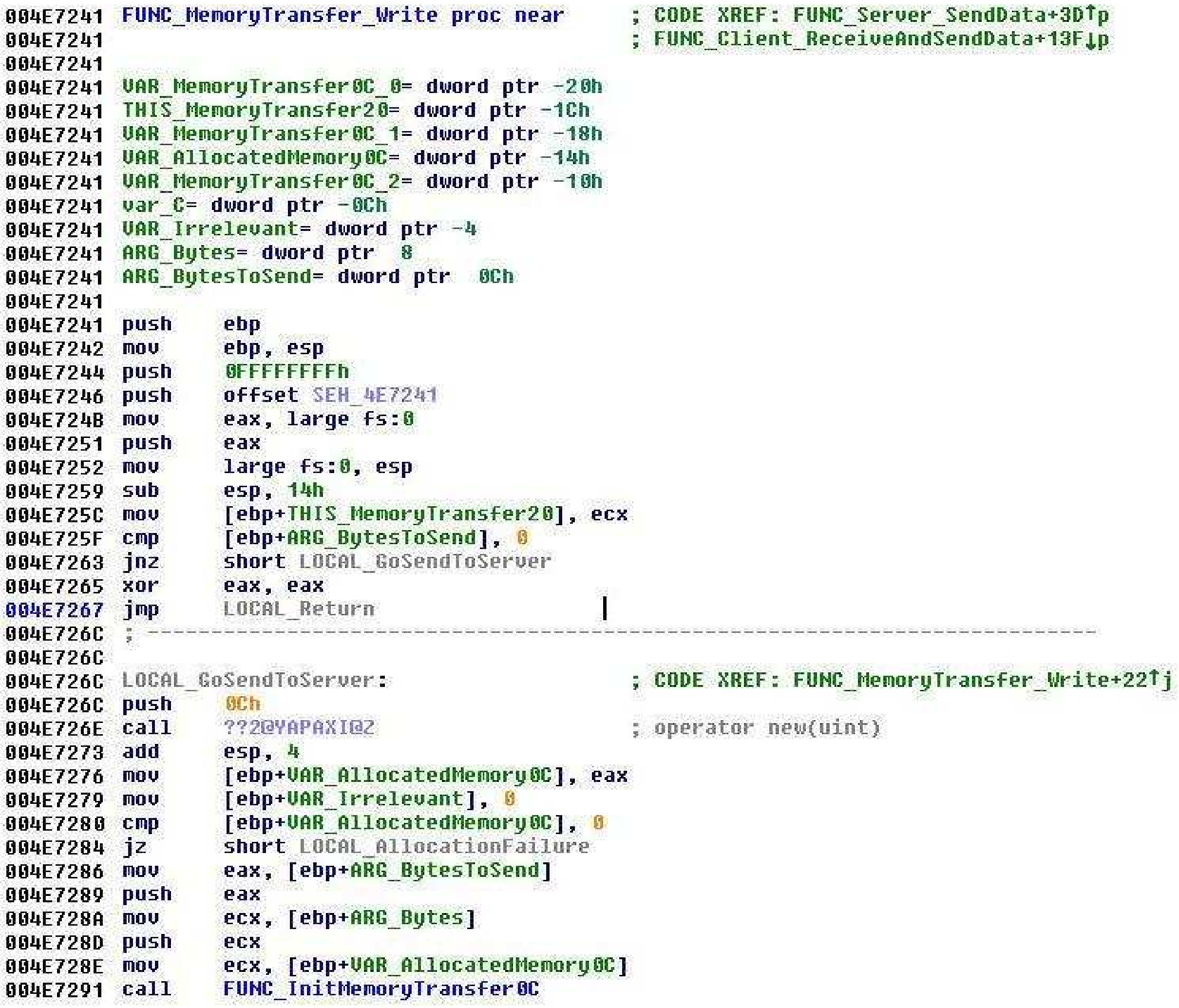


```
0047FF00 55                push    ebp
0047FF01 8B EC             mov     ebp, esp
0047FF03 83 EC 0C          sub     esp, 0Ch
0047FF06 89 4D F4          mov     [ebp+var_C], ecx
0047FF09 8B 45 F4          mov     eax, [ebp+var_C]
0047FF0C 8B 48 08          mov     ecx, [eax+8]
0047FF0F 3B 0D 60 16 70 00 cmp     ecx, GLOBAL_UserSocket
0047FF15 75 30             jnz     short loc_47FF47
0047FF17 8B 55 F4          mov     edx, [ebp+var_C]
0047FF1A 8B 42 08          mov     eax, [edx+8]
0047FF1D 8B 4D F4          mov     ecx, [ebp+var_C]
0047FF20 8B 51 10          mov     edx, [ecx+10h]
0047FF23 2B 50 1C          sub     edx, [eax+1Ch]
0047FF26 52                push    edx
0047FF27 8B 45 F4          mov     eax, [ebp+var_C]
0047FF2A 8B 48 08          mov     ecx, [eax+8]
0047FF2D 8B 55 F4          mov     edx, [ebp+var_C]
0047FF30 8B 42 0C          mov     eax, [edx+0Ch]
0047FF33 03 41 1C          add     eax, [ecx+1Ch]
0047FF36 50                push    eax
0047FF37 8B 0D 5C 16 70 00 mov     ecx, GLOBAL_MemoryTransfer20_ServerToClient
0047FF3D E8 FF 72 06 00    call    FUNC_MemoryTransfer_Write
0047FF42 89 45 FC          mov     [ebp+VAR_BytesSent], eax
0047FF45 EB 34             jmp     short loc_47FF7B
0047FF47                   ; ---------------------------------------------------------------------------
0047FF47
0047FF47                   loc_47FF47:                             ; CODE XREF: FUNC_Server_SendData+15↑j
0047FF47 6A 00             push    0                               ; flags
0047FF49 8B 4D F4          mov     ecx, [ebp+var_C]
0047FF4C 8B 51 08          mov     edx, [ecx+8]
0047FF4F 8B 45 F4          mov     eax, [ebp+var_C]
0047FF52 8B 48 10          mov     ecx, [eax+10h]
0047FF55 2B 4A 1C          sub     ecx, [edx+1Ch]
0047FF58 51                push    ecx                             ; len
0047FF59 8B 55 F4          mov     edx, [ebp+var_C]
0047FF5C 8B 42 08          mov     eax, [edx+8]
0047FF5F 8B 4D F4          mov     ecx, [ebp+var_C]
0047FF62 8B 51 0C          mov     edx, [ecx+0Ch]
0047FF65 03 50 1C          add     edx, [eax+1Ch]
0047FF68 52                push    edx                             ; buf
0047FF69 8B 45 F4          mov     eax, [ebp+var_C]
0047FF6C 8B 48 08          mov     ecx, [eax+8]
0047FF6F 8B 51 0C          mov     edx, [ecx+0Ch]
0047FF72 52                push    edx                             ; s
0047FF73 E8 0A C3 07 00    call    send
0047FF78 89 45 FC          mov     [ebp+VAR_BytesSent], eax
```

To get a deeper understanding we are going to explore the FUNC_MemoryTransfer_Write and related functions (logic implies you cannot read what has not been written).

FUNC_MemoryTransfer_Write

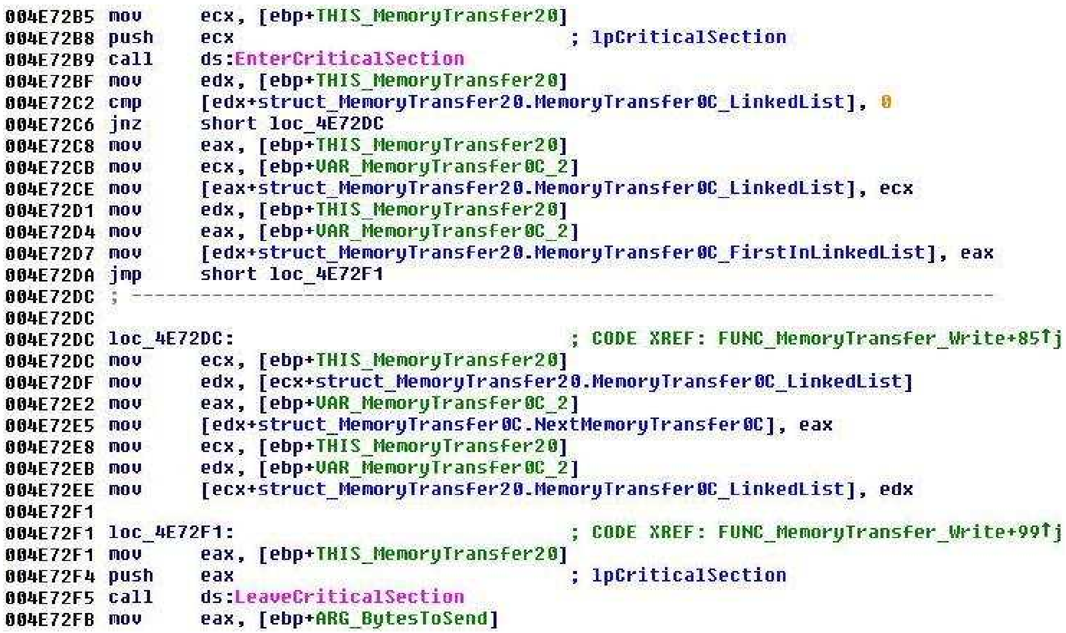
The Write function, after analysis, is pretty straightforward:



```
004E7241 FUNC_MemoryTransfer_Write proc near     ; CODE XREF: FUNC_Server_SendData+3D↑p
004E7241                                         ; FUNC_Client_ReceiveAndSendData+13F↓p
004E7241
004E7241 VAR_MemoryTransfer0C_0= dword ptr -20h
004E7241 THIS_MemoryTransfer20= dword ptr -1Ch
004E7241 VAR_MemoryTransfer0C_1= dword ptr -18h
004E7241 VAR_AllocatedMemory0C= dword ptr -14h
004E7241 VAR_MemoryTransfer0C_2= dword ptr -10h
004E7241 var_C= dword ptr -0Ch
004E7241 VAR_Irrelevant= dword ptr -4
004E7241 ARG_Bytes= dword ptr  8
004E7241 ARG_BytesToSend= dword ptr  0Ch
004E7241
004E7241 push    ebp
004E7242 mov     ebp, esp
004E7244 push    0FFFFFFFFh
004E7246 push    offset SEH_4E7241
004E724B mov     eax, large fs:0
004E7251 push    eax
004E7252 mov     large fs:0, esp
004E7259 sub     esp, 14h
004E725C mov     [ebp+THIS_MemoryTransfer20], ecx
004E725F cmp     [ebp+ARG_BytesToSend], 0
004E7263 jnz     short LOCAL_GoSendToServer
004E7265 xor     eax, eax
004E7267 jmp     LOCAL_Return
004E726C ; ---------------------------------------------------------------------------
004E726C
004E726C LOCAL_GoSendToServer:                   ; CODE XREF: FUNC_MemoryTransfer_Write+22↑j
004E726C push    0Ch
004E726E call    ??2@YAPAXI@Z                    ; operator new(uint)
004E7273 add     esp, 4
004E7276 mov     [ebp+VAR_AllocatedMemory0C], eax
004E7279 mov     [ebp+VAR_Irrelevant], 0
004E7280 cmp     [ebp+VAR_AllocatedMemory0C], 0
004E7284 jz      short LOCAL_AllocationFailure
004E7286 mov     eax, [ebp+ARG_BytesToSend]
004E7289 push    eax
004E728A mov     ecx, [ebp+ARG_Bytes]
004E728D push    ecx
004E728E mov     ecx, [ebp+VAR_AllocatedMemory0C]
004E7291 call    FUNC_InitMemoryTransfer0C
```

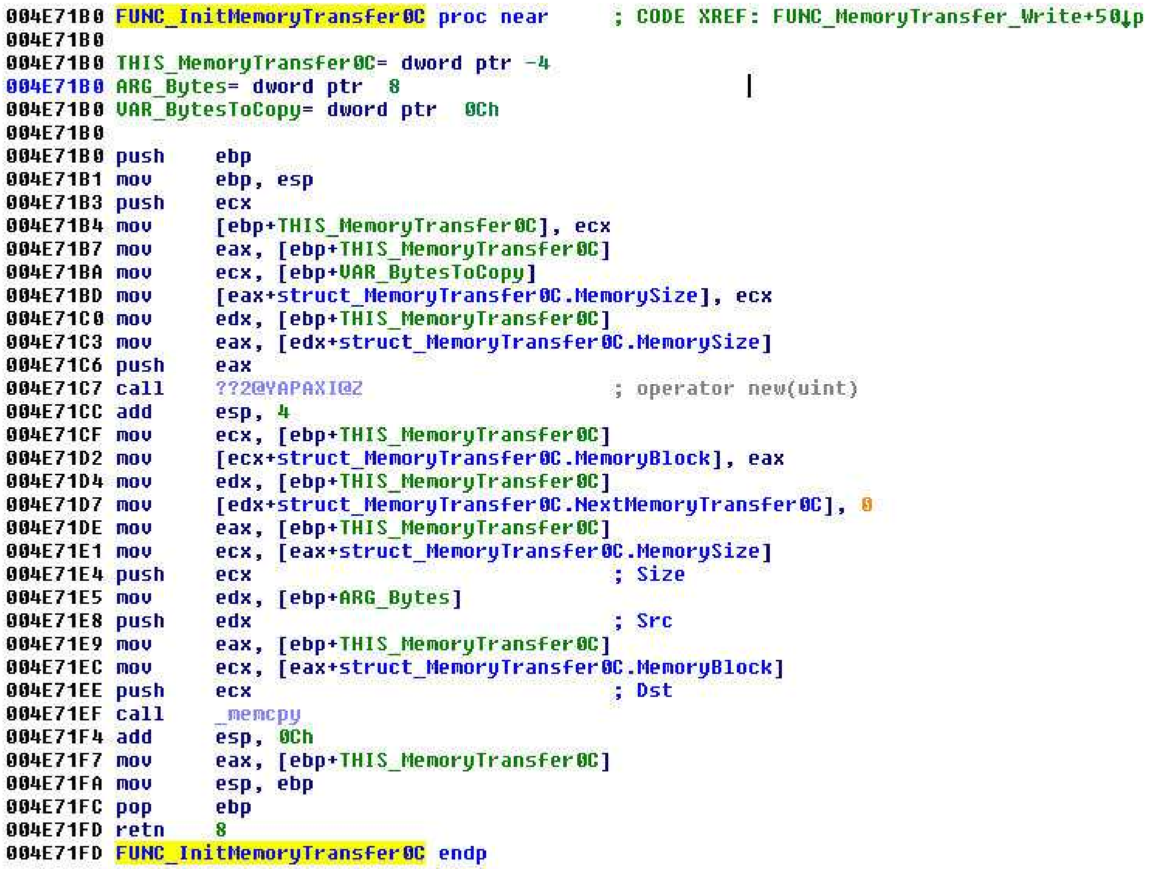
If zero bytes are being sent, the function returns without doing anything. The data being sent is placed into another object, which I named MemoryTransfer0C.

It's the second part of the function that is more interesting, the critical section is entered, the newly created MemoryTransfer0C object is added to a linked list and then the critical section is left.



```
004E72B5 mov     ecx, [ebp+THIS_MemoryTransfer20]
004E72B8 push    ecx                              ; lpCriticalSection
004E72B9 call    ds:EnterCriticalSection
004E72BF mov     edx, [ebp+THIS_MemoryTransfer20]
004E72C2 cmp     [edx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], 0
004E72C6 jnz     short loc_4E72DC
004E72C8 mov     eax, [ebp+THIS_MemoryTransfer20]
004E72CB mov     ecx, [ebp+VAR_MemoryTransfer0C_2]
004E72CE mov     [eax+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], ecx
004E72D1 mov     edx, [ebp+THIS_MemoryTransfer20]
004E72D4 mov     eax, [ebp+VAR_MemoryTransfer0C_2]
004E72D7 mov     [edx+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList], eax
004E72DA jmp     short loc_4E72F1
004E72DC ; ---------------------------------------------------------------------------
004E72DC
004E72DC loc_4E72DC:                              ; CODE XREF: FUNC_MemoryTransfer_Write+85↑j
004E72DC mov     ecx, [ebp+THIS_MemoryTransfer20]
004E72DF mov     edx, [ecx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList]
004E72E2 mov     eax, [ebp+VAR_MemoryTransfer0C_2]
004E72E5 mov     [edx+struct_MemoryTransfer0C.NextMemoryTransfer0C], eax
004E72E8 mov     ecx, [ebp+THIS_MemoryTransfer20]
004E72EB mov     edx, [ebp+VAR_MemoryTransfer0C_2]
004E72EE mov     [ecx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], edx
004E72F1
004E72F1 loc_4E72F1:                              ; CODE XREF: FUNC_MemoryTransfer_Write+99↑j
004E72F1 mov     eax, [ebp+THIS_MemoryTransfer20]
004E72F4 push    eax                              ; lpCriticalSection
004E72F5 call    ds:LeaveCriticalSection
004E72FB mov     eax, [ebp+ARG_BytesToSend]
```

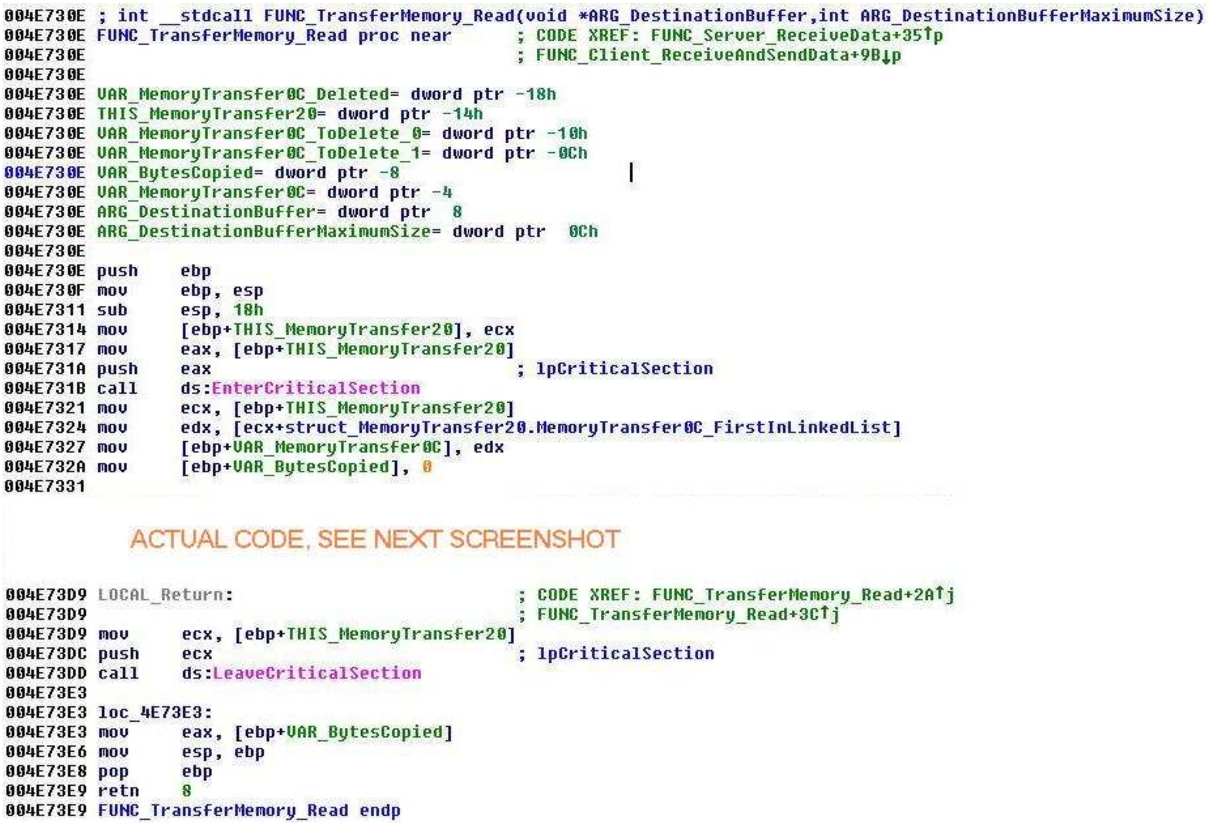
So, let's take a look inside the MemoryTransfer0C constructor. This function, again, is easy to follow, packet size is stored (ARG_Bytes) and a duplicate of the packet is made and stored (new+ memcpy). Plus the linked list pointer is set to NULL.



```
004E71B0 FUNC_InitMemoryTransfer0C proc near      ; CODE XREF: FUNC_MemoryTransfer_Write+50↓p
004E71B0
004E71B0 THIS_MemoryTransfer0C= dword ptr -4
004E71B0 ARG_Bytes= dword ptr  8
004E71B0 VAR_BytesToCopy= dword ptr  0Ch
004E71B0
004E71B0 push    ebp
004E71B1 mov     ebp, esp
004E71B3 push    ecx
004E71B4 mov     [ebp+THIS_MemoryTransfer0C], ecx
004E71B7 mov     eax, [ebp+THIS_MemoryTransfer0C]
004E71BA mov     ecx, [ebp+VAR_BytesToCopy]
004E71BD mov     [eax+struct_MemoryTransfer0C.MemorySize], ecx
004E71C0 mov     edx, [ebp+THIS_MemoryTransfer0C]
004E71C3 mov     eax, [edx+struct_MemoryTransfer0C.MemorySize]
004E71C6 push    eax
004E71C7 call    ??2@YAPAXI@Z                     ; operator new(uint)
004E71CC add     esp, 4
004E71CF mov     ecx, [ebp+THIS_MemoryTransfer0C]
004E71D2 mov     [ecx+struct_MemoryTransfer0C.MemoryBlock], eax
004E71D4 mov     edx, [ebp+THIS_MemoryTransfer0C]
004E71D7 mov     [edx+struct_MemoryTransfer0C.NextMemoryTransfer0C], 0
004E71DE mov     eax, [ebp+THIS_MemoryTransfer0C]
004E71E1 mov     ecx, [eax+struct_MemoryTransfer0C.MemorySize]
004E71E4 push    ecx                              ; Size
004E71E5 mov     edx, [ebp+ARG_Bytes]
004E71E8 push    edx                              ; Src
004E71E9 mov     eax, [ebp+THIS_MemoryTransfer0C]
004E71EC mov     ecx, [eax+struct_MemoryTransfer0C.MemoryBlock]
004E71EE push    ecx                              ; Dst
004E71EF call    _memcpy
004E71F4 add     esp, 0Ch
004E71F7 mov     eax, [ebp+THIS_MemoryTransfer0C]
004E71FA mov     esp, ebp
004E71FC pop     ebp
004E71FD retn    8
004E71FD FUNC_InitMemoryTransfer0C endp
```

FUNC_MemoryTransfer_Read

The reading of the packets turned out to be a bit more different than writing then. I was expecting to find code that would take the first MemoryTransfer0C object and unlink it from the linked list (inside a critical section). Boy; was I wrong.



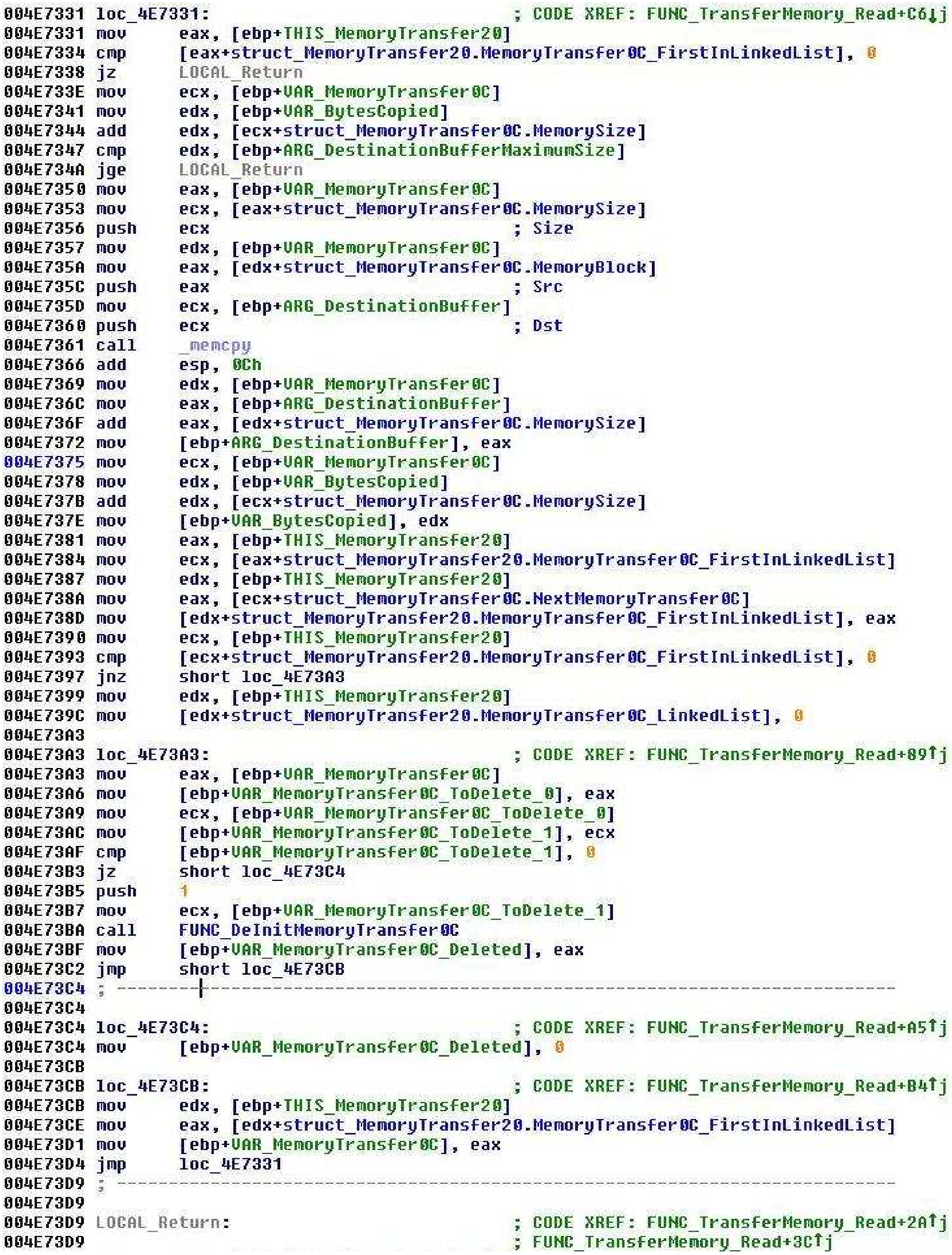
```
004E730E ; int __stdcall FUNC_TransferMemory_Read(void *ARG_DestinationBuffer,int ARG_DestinationBufferMaximumSize)
004E730E FUNC_TransferMemory_Read proc near       ; CODE XREF: FUNC_Server_ReceiveData+35↑p
004E730E                                          ; FUNC_Client_ReceiveAndSendData+9B↓p
004E730E
004E730E VAR_MemoryTransfer0C_Deleted= dword ptr -18h
004E730E THIS_MemoryTransfer20= dword ptr -14h
004E730E VAR_MemoryTransfer0C_ToDelete_0= dword ptr -10h
004E730E VAR_MemoryTransfer0C_ToDelete_1= dword ptr -0Ch
004E730E VAR_BytesCopied= dword ptr -8
004E730E VAR_MemoryTransfer0C= dword ptr -4
004E730E ARG_DestinationBuffer= dword ptr  8
004E730E ARG_DestinationBufferMaximumSize= dword ptr  0Ch
004E730E
004E730E push    ebp
004E730F mov     ebp, esp
004E7311 sub     esp, 18h
004E7314 mov     [ebp+THIS_MemoryTransfer20], ecx
004E7317 mov     eax, [ebp+THIS_MemoryTransfer20]
004E731A push    eax                              ; lpCriticalSection
004E731B call    ds:EnterCriticalSection
004E7321 mov     ecx, [ebp+THIS_MemoryTransfer20]
004E7324 mov     edx, [ecx+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList]
004E7327 mov     [ebp+VAR_MemoryTransfer0C], edx
004E732A mov     [ebp+VAR_BytesCopied], 0
004E7331

          ACTUAL CODE, SEE NEXT SCREENSHOT

004E73D9 LOCAL_Return:                            ; CODE XREF: FUNC_TransferMemory_Read+2A↑j
004E73D9                                          ; FUNC_TransferMemory_Read+3C↑j
004E73D9 mov     ecx, [ebp+THIS_MemoryTransfer20]
004E73DC push    ecx                              ; lpCriticalSection
004E73DD call    ds:LeaveCriticalSection
004E73E3
004E73E3 loc_4E73E3:
004E73E3 mov     eax, [ebp+VAR_BytesCopied]
004E73E6 mov     esp, ebp
004E73E8 pop     ebp
004E73E9 retn    8
004E73E9 FUNC_TransferMemory_Read endp
```

For clarity sake I split the screenshots in two parts, the first one (seen above) shows the function entry and exit points. At entry the critical section is entered and at exit the critical section is left. Basic stuff actually.

Turn to the next page to view the second screenshot displaying the actual code.



```
004E7331 loc_4E7331:                             ; CODE XREF: FUNC_TransferMemory_Read+C6↓j
004E7331 mov     eax, [ebp+THIS_MemoryTransfer20]
004E7334 cmp     [eax+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList], 0
004E7338 jz      LOCAL_Return
004E733E mov     ecx, [ebp+VAR_MemoryTransfer0C]
004E7341 mov     edx, [ebp+VAR_BytesCopied]
004E7344 add     edx, [ecx+struct_MemoryTransfer0C.MemorySize]
004E7347 cmp     edx, [ebp+ARG_DestinationBufferMaximumSize]
004E734A jge     LOCAL_Return
004E7350 mov     eax, [ebp+VAR_MemoryTransfer0C]
004E7353 mov     ecx, [eax+struct_MemoryTransfer0C.MemorySize]
004E7356 push    ecx                             ; Size
004E7357 mov     edx, [ebp+VAR_MemoryTransfer0C]
004E735A mov     eax, [edx+struct_MemoryTransfer0C.MemoryBlock]
004E735C push    eax                             ; Src
004E735D mov     ecx, [ebp+ARG_DestinationBuffer]
004E7360 push    ecx                             ; Dst
004E7361 call    _memcpy
004E7366 add     esp, 0Ch
004E7369 mov     edx, [ebp+VAR_MemoryTransfer0C]
004E736C mov     eax, [ebp+ARG_DestinationBuffer]
004E736F add     eax, [edx+struct_MemoryTransfer0C.MemorySize]
004E7372 mov     [ebp+ARG_DestinationBuffer], eax
004E7375 mov     ecx, [ebp+VAR_MemoryTransfer0C]
004E7378 mov     edx, [ebp+VAR_BytesCopied]
004E737B add     edx, [ecx+struct_MemoryTransfer0C.MemorySize]
004E737E mov     [ebp+VAR_BytesCopied], edx
004E7381 mov     eax, [ebp+THIS_MemoryTransfer20]
004E7384 mov     ecx, [eax+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList]
004E7387 mov     edx, [ebp+THIS_MemoryTransfer20]
004E738A mov     eax, [ecx+struct_MemoryTransfer0C.NextMemoryTransfer0C]
004E738D mov     [edx+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList], eax
004E7390 mov     ecx, [ebp+THIS_MemoryTransfer20]
004E7393 cmp     [ecx+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList], 0
004E7397 jnz     short loc_4E73A3
004E7399 mov     edx, [ebp+THIS_MemoryTransfer20]
004E739C mov     [edx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], 0
004E73A3
004E73A3 loc_4E73A3:                             ; CODE XREF: FUNC_TransferMemory_Read+89↑j
004E73A3 mov     eax, [ebp+VAR_MemoryTransfer0C]
004E73A6 mov     [ebp+VAR_MemoryTransfer0C_ToDelete_0], eax
004E73A9 mov     ecx, [ebp+VAR_MemoryTransfer0C_ToDelete_0]
004E73AC mov     [ebp+VAR_MemoryTransfer0C_ToDelete_1], ecx
004E73AF cmp     [ebp+VAR_MemoryTransfer0C_ToDelete_1], 0
004E73B3 jz      short loc_4E73C4
004E73B5 push    1
004E73B7 mov     ecx, [ebp+VAR_MemoryTransfer0C_ToDelete_1]
004E73BA call    FUNC_DeInitMemoryTransfer0C
004E73BF mov     [ebp+VAR_MemoryTransfer0C_Deleted], eax
004E73C2 jmp     short loc_4E73CB
004E73C4 ; ---------------------------------------------------------------------------
004E73C4
004E73C4 loc_4E73C4:                             ; CODE XREF: FUNC_TransferMemory_Read+A5↑j
004E73C4 mov     [ebp+VAR_MemoryTransfer0C_Deleted], 0
004E73CB
004E73CB loc_4E73CB:                             ; CODE XREF: FUNC_TransferMemory_Read+B4↑j
004E73CB mov     edx, [ebp+THIS_MemoryTransfer20]
004E73CE mov     eax, [edx+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList]
004E73D1 mov     [ebp+VAR_MemoryTransfer0C], eax
004E73D4 jmp     loc_4E7331
004E73D9 ; ---------------------------------------------------------------------------
004E73D9
004E73D9 LOCAL_Return:                           ; CODE XREF: FUNC_TransferMemory_Read+2A↑j
004E73D9                                         ; FUNC_TransferMemory_Read+3C↑j
```

What going on is: the function is called with an address of a target buffer and a size of that target buffer. Than as long as there is room inside that target buffer, the packets are copied to the target buffer and removed from the linked list (and the MemoryTransfer0C object is deleted).

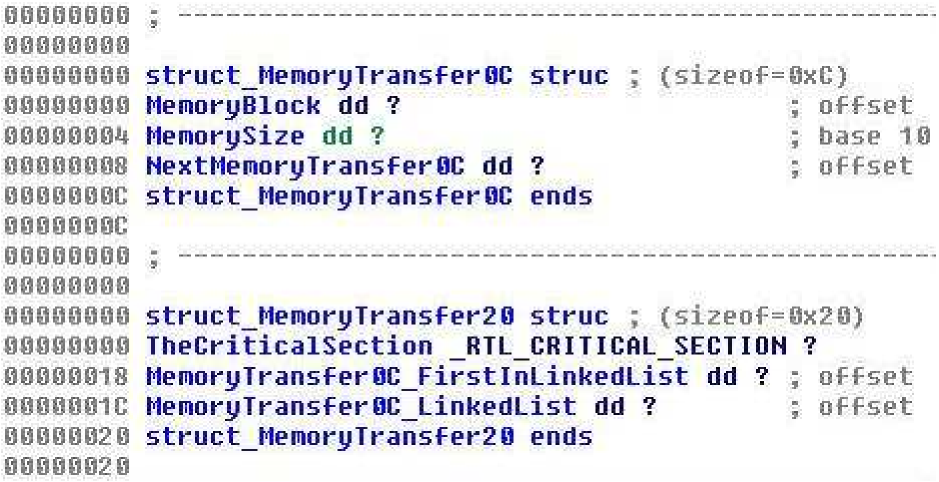
CONCLUSION

Basically:

- the TransferMemory_Write function writes packet per packet
- the TransferMemory_Read function reads as many packets as possible

THE STRUCTURES INVOLVED

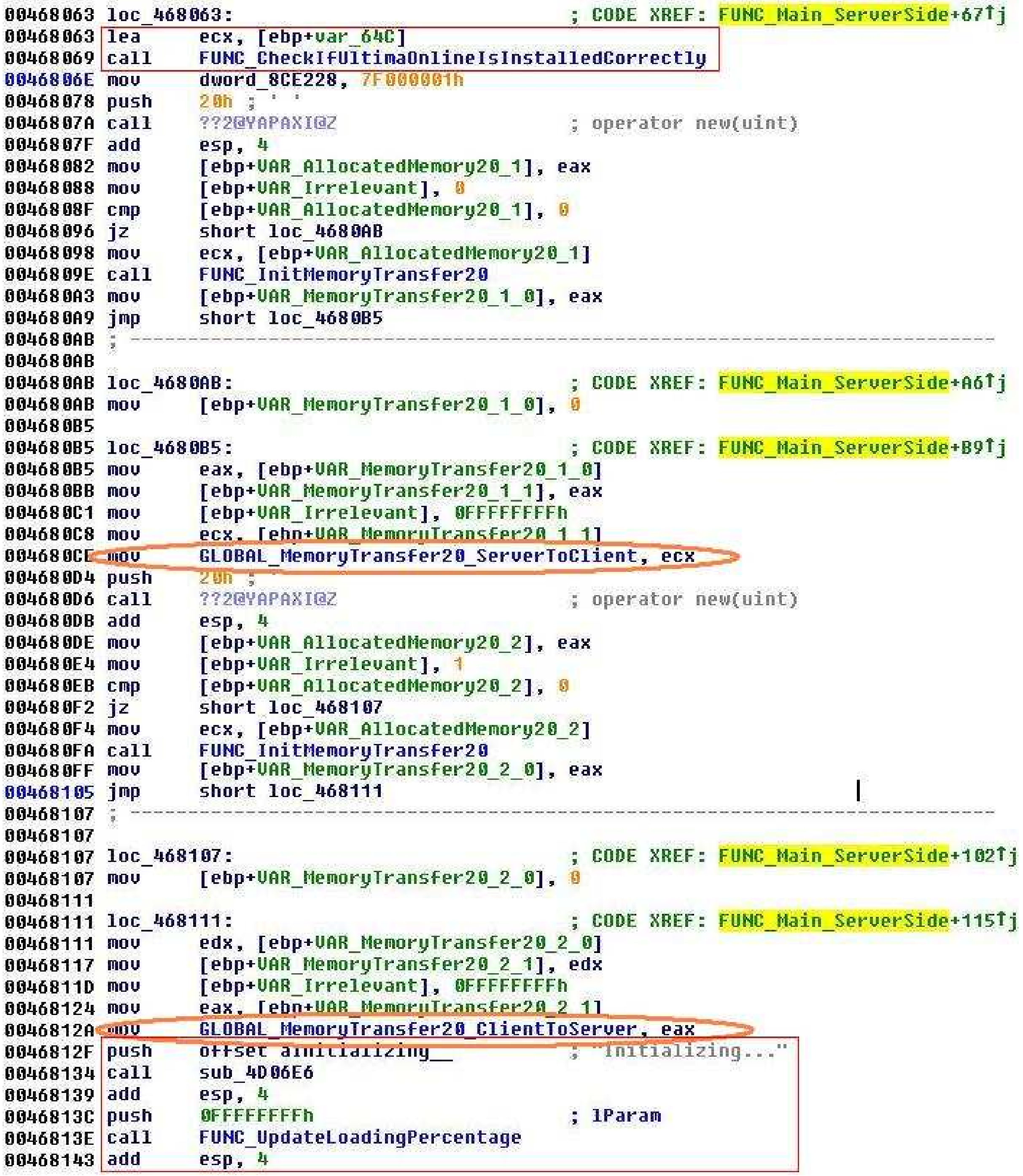
If you are going to do this yourself in IDA Pro, which I encourage, here are the structures I unraveled:



```
00000000 ; ---------------------------------------------------------------
00000000
00000000 struct_MemoryTransfer0C struc ; (sizeof=0xC)
00000000 MemoryBlock dd ?                           ; offset
00000004 MemorySize dd ?                            ; base 10
00000008 NextMemoryTransfer0C dd ?                  ; offset
0000000C struct_MemoryTransfer0C ends
0000000C
00000000 ; ---------------------------------------------------------------
00000000
00000000 struct_MemoryTransfer20 struc ; (sizeof=0x20)
00000000 TheCriticalSection _RTL_CRITICAL_SECTION ?
00000018 MemoryTransfer0C_FirstInLinkedList dd ? ; offset
0000001C MemoryTransfer0C_LinkedList dd ?           ; offset
00000020 struct_MemoryTransfer20 ends
00000020
```

2 object instances involved

During initialization the demo initializes two MemoryTransfer20 objects, one for communication from the Client to the Server, another one for communication from the Server to the Client.



```
00468063 loc_468063:                            ; CODE XREF: FUNC_Main_ServerSide+67↑j
00468063 lea     ecx, [ebp+var_64C]
00468069 call    FUNC_CheckIfUltimaOnlineIsInstalledCorrectly
0046806E mov     dword_8CE228, 7F000001h
00468078 push    20h ; ' '
0046807A call    ??2@YAPAXI@Z                   ; operator new(uint)
0046807F add     esp, 4
00468082 mov     [ebp+VAR_AllocatedMemory20_1], eax
00468088 mov     [ebp+VAR_Irrelevant], 0
0046808F cmp     [ebp+VAR_AllocatedMemory20_1], 0
00468096 jz      short loc_4680AB
00468098 mov     ecx, [ebp+VAR_AllocatedMemory20_1]
0046809E call    FUNC_InitMemoryTransfer20
004680A3 mov     [ebp+VAR_MemoryTransfer20_1_0], eax
004680A9 jmp     short loc_4680B5
004680AB ; ---------------------------------------------------------------------------
004680AB
004680AB loc_4680AB:                            ; CODE XREF: FUNC_Main_ServerSide+A6↑j
004680AB mov     [ebp+VAR_MemoryTransfer20_1_0], 0
004680B5
004680B5 loc_4680B5:                            ; CODE XREF: FUNC_Main_ServerSide+B9↑j
004680B5 mov     eax, [ebp+VAR_MemoryTransfer20_1_0]
004680BB mov     [ebp+VAR_MemoryTransfer20_1_1], eax
004680C1 mov     [ebp+VAR_Irrelevant], 0FFFFFFFFh
004680C8 mov     ecx, [ebp+VAR_MemoryTransfer20_1_1]
004680CE mov     GLOBAL_MemoryTransfer20_ServerToClient, ecx
004680D4 push    20h ;
004680D6 call    ??2@YAPAXI@Z                   ; operator new(uint)
004680DB add     esp, 4
004680DE mov     [ebp+VAR_AllocatedMemory20_2], eax
004680E4 mov     [ebp+VAR_Irrelevant], 1
004680EB cmp     [ebp+VAR_AllocatedMemory20_2], 0
004680F2 jz      short loc_468107
004680F4 mov     ecx, [ebp+VAR_AllocatedMemory20_2]
004680FA call    FUNC_InitMemoryTransfer20
004680FF mov     [ebp+VAR_MemoryTransfer20_2_0], eax
00468105 jmp     short loc_468111
00468107 ; ---------------------------------------------------------------------------
00468107
00468107 loc_468107:                            ; CODE XREF: FUNC_Main_ServerSide+102↑j
00468107 mov     [ebp+VAR_MemoryTransfer20_2_0], 0
00468111
00468111 loc_468111:                            ; CODE XREF: FUNC_Main_ServerSide+115↑j
00468111 mov     edx, [ebp+VAR_MemoryTransfer20_2_0]
00468117 mov     [ebp+VAR_MemoryTransfer20_2_1], edx
0046811D mov     [ebp+VAR_Irrelevant], 0FFFFFFFFh
00468124 mov     eax, [ebp+VAR_MemoryTransfer20_2_1]
0046812A mov     GLOBAL_MemoryTransfer20_ClientToServer, eax
0046812F push    offset aInitializing__         ; "Initializing..."
00468134 call    sub_4D06E6
00468139 add     esp, 4
0046813C push    0FFFFFFFFh                     ; lParam
0046813E call    FUNC_UpdateLoadingPercentage
00468143 add     esp, 4
```

Using IDA Pro’s cross reference function you can find out where the objects are used.

This concludes part 1. Really; do this cross reference stuff yourself ☺ or wait for future “Inside The Ultima Online Demo” publishes.