

INSIDE THE ULTIMA ONLINE GOLD DEMO - THE PACKET COMMUNICATION – PART 2

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 learned programming in BASIC, and then I switched to C and from there to assembler. I think that's the most logical order and I believe every one should learn to program like that. BASIC because it's easy, C because it's a language where other languages are based upon (C++, Java, C#, ...) and at the same time you have to stay focused to avoid buffer overruns (which I consider to be an important skill). Assembler will teach you how a program works and it doesn't open a door or a window, it opens a **gate** to the inner workings of your computer.

SMALL RECAPULATION

In part 1 of The Packet Communication we learned where and how the sending and receiving of the packets is done (at the transfer level, not at the game level). Today's goal is to insert code to enable logging of these packets.

It's my decision to use the log format of the Razor application. Razor is a macro utility for free shards. This log format is clear text so files will become big quickly. I could have chosen to create my own binary format which will be easier to implement but this means that yet another log format would have been born. That my dear reader I wanted to avoid!

WHERE TO START THE PATCH

My first idea was to modify the constructor of the MemoryTransferOC object, however; this object is initialized before the critical section is entered. If we have two log files, one for client to server and one for server to client, this wouldn't be much of a problem. That is however a silly idea.

Remember, we have to keep thread-safety into mind so it's better to insert our code between the EnterCriticalSection and LeaveCriticalSection calls.

```
004E72B5 mov     ecx, [ebp+THIS_MemoryTransfer20]
004E72B8 push   ecx
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 loc_4E72DC:
004E72DC mov     ecx, [ebp+THIS_MemoryTransfer20] ; CODE XREF: FUNC_MemoryTransfer_Write+85↑j
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:
004E72F1 mov     eax, [ebp+THIS_MemoryTransfer20] ; CODE XREF: FUNC_MemoryTransfer_Write+99↑j
004E72F4 push   eax
004E72F5 call   ds:LeaveCriticalSection
004E72FB mov     eax, [ebp+ARG_BytesToSend]
```

Do you see place to insert code? I didn't.

We can redesign the whole function and optimize it correctly to gain space. However, I took a different approach. I replaced the call to LeaveCriticalSection with a call to inserted code plus I added code to make sure we can use the fastcall.

WHAT IS FASTCALL

Fastcall is a calling convention like any other ☺. When using fastcall you are passing variables through registers and not through the stack (which is faster, hence fastcall). Extra reading: http://en.wikipedia.org/wiki/X86_calling_conventions#fastcall.

WHAT TO INSERT

The code we are adding in this patch is really big compared with previous patches so I decided to code and test this in C. Remember, modern C compilers support 64-bit time_t values, but due to its age the uodemo is limited to 32-bit. Luckily you can tell modern C compilers to use the old-style time_t standard.

WARNING! With this patch I could be introducing the [Year 2038 bug](#) in the demo because of using 32-bit time_t!

The actual C code is attached at the end of this document!

THE PATCH ENTRY

The screenshot below shows the modified FUNC_MemoryTransfer_Write function, notice the added “XCHG EAX, EDX”:

```
004E72C2 83 7A 1C 00      cmp     [edx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], 0
004E72C6 75 14           jnz    short loc_4E72DC
004E72C8 8B 45 E4       mov     eax, [ebp+THIS_MemoryTransfer20]
004E72CB 8B 4D F0       mov     ecx, [ebp+UAR_MemoryTransfer0C_2]
004E72CE 89 48 1C       mov     [eax+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], ecx
004E72D1 8B 55 E4       mov     edx, [ebp+THIS_MemoryTransfer20]
004E72D4 8B 45 F0       mov     eax, [ebp+UAR_MemoryTransfer0C_2]
004E72D7 89 42 18       mov     [edx+struct_MemoryTransfer20.MemoryTransfer0C_FirstInLinkedList], eax
004E72DA EB 16         jmp     short loc_4E72F2
004E72DC
004E72DC
loc_4E72DC:
004E72DC 8B 4D E4       mov     ecx, [ebp+THIS_MemoryTransfer20] ; CODE XREF: FUNC_MemoryTransfer_Write+85↑j
004E72DF 8B 51 1C       mov     edx, [ecx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList]
004E72E2 8B 45 F0       mov     eax, [ebp+UAR_MemoryTransfer0C_2]
004E72E5 89 42 08       mov     [edx+struct_MemoryTransfer0C.NextMemoryTransfer0C], eax
004E72E8 8B 4D E4       mov     ecx, [ebp+THIS_MemoryTransfer20]
004E72EB 8B 55 F0       mov     edx, [ebp+UAR_MemoryTransfer0C_2]
004E72EE 89 51 1C       mov     [ecx+struct_MemoryTransfer20.MemoryTransfer0C_LinkedList], edx
004E72F1 92           xchg   eax, edx
004E72F2
004E72F2
loc_4E72F2:
004E72F2 8B 55 E4       mov     edx, [ebp+THIS_MemoryTransfer20] ; CODE XREF: FUNC_MemoryTransfer_Write+99↑j
004E72F5 52           push   edx
004E72F6 E8 27 5D 10 00 call   FUNC_LoggerEntry_Patch
004E72FB 8B 45 0C       mov     eax, [ebp+ARG_BytesToSend]
```

The EAX will contain a pointer to the MemoryTransfer0C object, remember, that object contains the pointer to the packet data and contains the packet size.

This is the called FUNC_LoggerEntry function (I added __Patch so I know during the debugging sessions that it's a patch that is being called):

```
005ED022          FUNC_LoggerEntry__Patch proc near      ; CODE XREF: FUNC_MemoryTransfer_Write+B5f
005ED022
005ED022          ARG_EIPofCaller_FarAway= dword ptr  2Ch
005ED022
005ED022  8B 48 04      mov     ecx, [eax+struct_MemoryTransfer0C.MemorySize]
005ED025  8B 10        mov     edx, [eax+struct_MemoryTransfer0C.MemoryBlock]
005ED027  8B 44 24 2C   mov     eax, [esp+ARG_EIPofCaller_FarAway]
005ED02B  FF 15 60 AC 9A 00  call  ds:GLOBAL_LogFunction
005ED031  FF 25 E0 55 9A 00  jmp    ds:LeaveCriticalSection
005ED031          FUNC_LoggerEntry__Patch endp
```

I will clarify that function a bit; it looks odd if you don't know assembler that well.

We know from the previous screenshot that EAX is used to pass the pointer to the MemoryTransfer0C object, so the memory size and memory block (packet) is placed into respectively ECX and EDX (fastcall at work again).

Then ARG_EIPofCaller_FarAway is put into EAX. This EIP is **not** the address of the FUNC_MemoryTransfer_Write. It is the EIP of where FUNC_MemoryTransfer will return to. That EIP will tell us whether it is the server code that is calling the write function or whether it is the client code that is calling. This is important so the logger knows it should record Server->Client or Client->Server. You cannot code ARG_EIPofCaller_FarAway in C without the help of extra variables or assembler. This is the power of assembler at its best because we have direct access to the stack.

GLOBAL_LogFunction is a pointer to the log function (stored in memory). The idea is: the first time the function is called "LOGFILE" is looked up in the environment strings, if not found we will replace GLOBAL_LogFunction with a call to a "nullsub". If found we will replace GLOBAL_LogFunction with a call to the log function. That way we don't have to check every time whether or not the log file has been opened. It's a simply but effective speed optimization technique.

THE LOGGING

This is the first part of the actual log function. Remember GLOBAL_LogHandle is not checked against NULL because this function can only be called when it has been set.

```

005ED0B6          FUNC_LoggerLogPacket__Patch proc near    ; DATA XREF: FUNC_LoggerInit__Patch+6C↑o
005ED0B6          var_E4= dword ptr -0E4h
005ED0B6          var_DA= dword ptr -0DAh
005ED0B6          var_78= byte ptr -78h
005ED0B6          Dest= byte ptr -38h
005ED0B6          SystemTime= _SYSTEMTIME ptr -32h
005ED0B6          var_1C= byte ptr -1Ch
005ED0B6          arg_8= dword ptr 0Ch
005ED0B6          arg_10= dword ptr 14h
005ED0B6          arg_14= dword ptr 18h
005ED0B6          arg_18= dword ptr 1Ch
005ED0B6          or      ecx, ecx
005ED0B8 09 C9          jnz     short LOCAL_Continue
005ED0BA 75 01          retn
005ED0BB          ; -----|-----
005ED0BB          LOCAL_Continue:                          ; CODE XREF: FUNC_LoggerLogPacket__Patch+2T↑j
005ED0BB          pusha
005ED0BC 89 E5          mov     ebp, esp
005ED0BE 83 EC 78      sub     esp, 78h
005ED0C1 BF F9 5F 60 00 mov     edi, offset a_Server                ; "Server"
005ED0C6 81 7D 1C E3 3F 51 00 cmp     [ebp+arg_18], offset loc_513FE3
005ED0CD BE F2 5F 60 00 mov     esi, offset a_Client                ; "Client"
005ED0D2 75 02          jnz     short LOCAL_LogDirectionOK
005ED0D4 87 F7          xchg   esi, edi
005ED0D6          LOCAL_LogDirectionOK:                      ; CODE XREF: FUNC_LoggerLogPacket__Patch+1C↑j
005ED0D6          mov     eax, ds:GLOBAL_LogHandle
005ED0DB 8D 4D CE      lea    ecx, [ebp+SystemTime]
005ED0DE 93           xchg   eax, ebx
005ED0DF 51           push  ecx                                  ; lpSystemTime
005ED0E0 FF 15 A8 55 9A 00 call   ds:GetLocalTime
005ED0E6 8B 45 14      mov     eax, [ebp+arg_10]
005ED0E9 FF 75 18      push   [ebp+arg_14]
005ED0EC 0F B6 10      movzx  edx, byte ptr [eax]
005ED0EF 93           xchg   eax, ebx
005ED0F0 52           push  edx
005ED0F1 0F B7 4D DC  movzx  ecx, [ebp+SystemTime.uMilliseconds]
005ED0F5 56           push  esi
005ED0F6 57           push  edi
005ED0F7 51           push  ecx
005ED0F8 97           xchg   eax, edi
005ED0F9 0F B7 4D DA  movzx  ecx, [ebp+SystemTime.uSecond]
005ED0FD 0F B7 55 D6  movzx  edx, [ebp+SystemTime.uHour]
005ED101 51           push  ecx
005ED102 FF 75 D8      push  dword ptr [ebp+SystemTime.uMinute]
005ED105 52           push  edx
005ED106 68 A0 5F 60 00 push  offset a_LogPacketHeader            ; Format
005ED10B 57           push  edi                                  ; File
005ED10C E8 8F CC EF FF call   _fprintf
005ED111 BE D5 5F 60 00 mov     esi, offset a_S                    ; Format
005ED116 68 68 AC 9A 00 push  offset a_LogText1                  ; "
005ED11B 56           push  esi                                  ; Format
005ED11C 57           push  edi                                  ; File
005ED11D E8 7E CC EF FF call   _fprintf
005ED122 68 A0 AC 9A 00 push  offset a_LogText2                  ; "
005ED127 56           push  esi                                  ; Format
005ED128 57           push  edi                                  ; File
005ED129 E8 72 CC EF FF call   _fprintf
005ED12E 31 C0        xor     eax, eax
005ED130 89 45 0C      mov     [ebp+arg_8], eax
005ED133          LOCAL_MainLoop:                          ; CODE XREF: FUNC_LoggerLogPacket__Patch+12F↑j

```

In the first stage, EDI and ESI are used to point a string either “Server” or “Client”. Depending on the EAX register (ARG_EIPofCaller_FarAway) they are swapped or not.

Instead of the C API “time(NULL)”, this time the Windows API “GetLocalTime” is used to obtain the current time. Why? Because GetLocalTime returns the current millisecond whereas the C API doesn’t return this value at all. In a later stage, EDI is used to contain the log file handle (GLOBAL_LogHandle) and ESI points to a__S__ (which is “%s\r\n”).

This big screenshot shows the second part of the logger function:

```

005ED133 LOCAL_MainLoop: ; CODE XREF: FUNC_LoggerLogPacket__Patch+12F↓j
005ED133 56 push esi
005ED134 57 push edi
005ED135 6A 08 push 8 ; Dest
005ED137 8B 7D 18 mov edi, [ebp+arg_14]
005ED13A 58 pop eax
005ED13B 50 push eax
005ED13C 39 C7 cmp edi, eax
005ED13E 8D 75 C8 lea esi, [ebp+Dest]
005ED141 76 01 jbe short LOCAL_Ready1
005ED143 97 xchg eax, edi
005ED144
005ED144 LOCAL_Ready1: ; CODE XREF: FUNC_LoggerLogPacket__Patch+8B↑j
005ED144 53 push ebx
005ED145
005ED145 LOCAL_Loop1: ; CODE XREF: FUNC_LoggerLogPacket__Patch+A6↓j
005ED145 0F B6 03 movzx eax, byte ptr [ebx]
005ED148 50 push eax
005ED149 68 EC 5F 60 00 push offset a_X_ ; "%02X "
005ED14E 56 push esi ; Dest
005ED14F E8 2C B6 EF FF call _sprintf
005ED154 83 C4 0C add esp, 0Ch
005ED157 43 inc ebx
005ED158 83 C6 03 add esi, 3
005ED15B 4F dec edi
005ED15C 75 E7 jnz short LOCAL_Loop1
005ED15E 5B pop ebx
005ED15F 59 pop ecx
005ED160 8D 75 E4 lea esi, [ebp+var_1C]
005ED163 8D 04 09 lea eax, [ecx+ecx]
005ED166 8B 7D 18 mov edi, [ebp+arg_14]
005ED169 50 push eax
005ED16A 39 C7 cmp edi, eax
005ED16C 8B 26 mov [esi], ah
005ED16E 76 01 jbe short LOCAL_Ready2
005ED170 97 xchg eax, edi
005ED171
005ED171 LOCAL_Ready2: ; CODE XREF: FUNC_LoggerLogPacket__Patch+B8↑j
005ED171 29 CF sub edi, ecx
005ED173 76 1D jbe short LOCAL_SkipLoop2
005ED175 53 push ebx
005ED176 01 CB add ebx, ecx
005ED178
005ED178 LOCAL_Loop2: ; CODE XREF: FUNC_LoggerLogPacket__Patch+D9↓j
005ED178 0F B6 03 movzx eax, byte ptr [ebx]
005ED17B 50 push eax
005ED17C 68 EC 5F 60 00 push offset a_X_ ; "%02X "
005ED181 56 push esi ; Dest
005ED182 E8 F9 B5 EF FF call _sprintf
005ED187 83 C4 0C add esp, 0Ch
005ED18A 43 inc ebx
005ED18B 83 C6 03 add esi, 3
005ED18E 4F dec edi
005ED18F 75 E7 jnz short LOCAL_Loop2
005ED191 5B pop ebx
005ED192
005ED192 LOCAL_SkipLoop2: ; CODE XREF: FUNC_LoggerLogPacket__Patch+BD↑j
005ED192 5E pop esi
005ED193 5F pop edi
005ED194 8D 45 E4 lea eax, [ebp+var_1C]
005ED197 8D 4D C8 lea ecx, [ebp+Dest]
005ED19A 50 push eax
005ED19B 51 push ecx
005ED19C 8D 45 88 lea eax, [ebp+var_78]
005ED19F FF 75 0C push [ebp+arg_8]
005ED1A2 68 DA 5F 60 00 push offset a_LogPacketData ; "%04X %-25s%-26s"
005ED1A7 50 push eax ; Dest
005ED1A8 E8 D3 B5 EF FF call _sprintf
005ED1AD 83 C4 14 add esp, 14h
005ED1B0 8D 54 05 88 lea edx, [ebp+eax+var_78]
005ED1B4 8B 4D 18 mov ecx, [ebp+arg_14]
005ED1B7 39 F1 cmp ecx, esi
005ED1B9 76 02 jbe short LOCAL_Ready3andLoop3
005ED1BB 89 F1 mov ecx, esi
005ED1BD
005ED1BD LOCAL_Ready3andLoop3: ; CODE XREF: FUNC_LoggerLogPacket__Patch+103↑j
005ED1BD ; FUNC_LoggerLogPacket__Patch+112↓j
005ED1BD 8A 03 mov al, [ebx]
005ED1BF E8 31 01 F0 FF call FUNC_LoggerFixCharacter__Patch
005ED1C4 8B 02 mov [edx], al
005ED1C6 43 inc ebx
005ED1C7 42 inc edx ; File
005ED1C8 E2 F3 loop LOCAL_Ready3andLoop3
005ED1CA 8B 0A mov [edx], cl
005ED1CC 8D 45 88 lea eax, [ebp+var_78]
005ED1CF 50 push eax
005ED1D0 68 D5 5F 60 00 push offset a_S_ ; "%s\r\n"
005ED1D5 57 push edi ; File
005ED1D6 E8 C5 CB EF FF call _fprintf
005ED1DB 83 C4 0C add esp, 0Ch
005ED1DE 01 75 0C add [ebp+arg_8], esi
005ED1E1 29 75 18 sub [ebp+arg_14], esi
005ED1E4 5E pop esi
005ED1E5 0F 87 48 FF FF FF ja LOCAL_MainLoop ; Format

```


FUNC_LoggerFixCharacter_Patch

The main loop contained code to replace non-standard ASCII strings with a dot, same as Razor is doing. However, I needed more bytes so I had to place that code elsewhere and I replaced it with a single call.

Before:

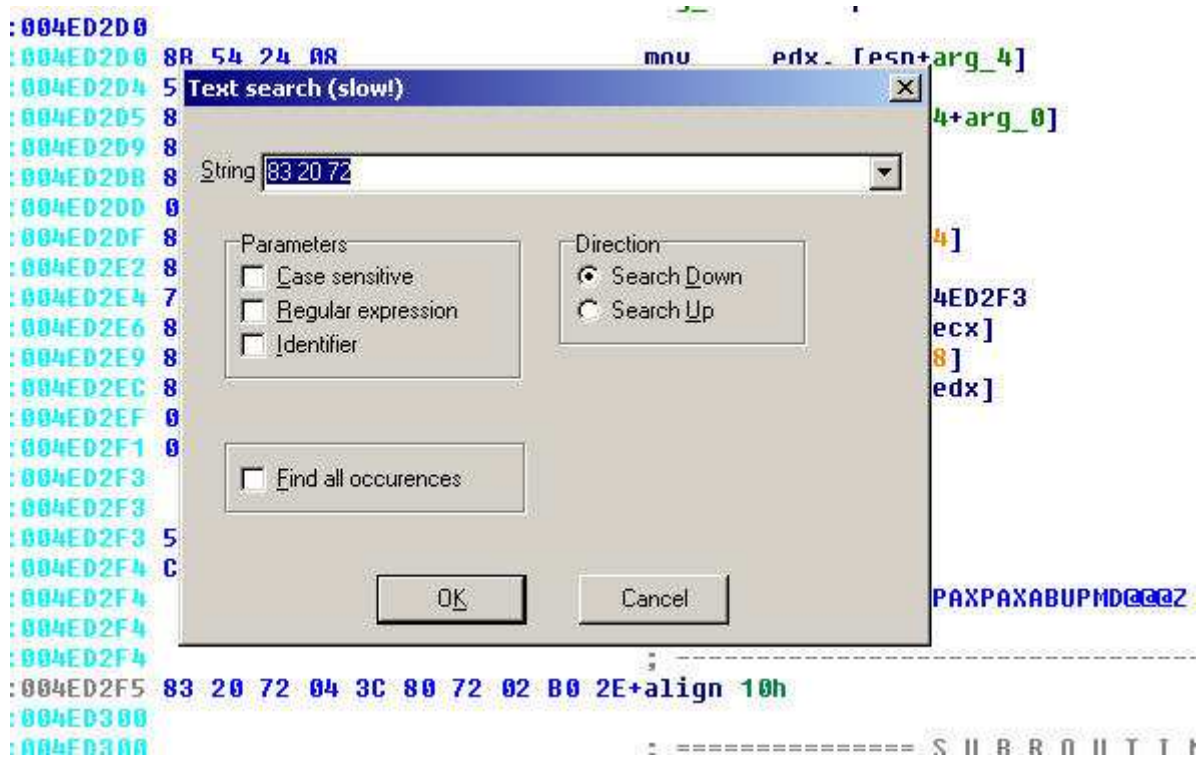
```
89 F1      mov     ecx, esi
          loc_5ED1AA:      ; CODE XREF: .text:005ED1BA↓j
8A 03      mov     al, [ebx]
          loc_5ED1AC:      ; CODE XREF: .text:005ED1A6↑j
3C 20      cmp     al, 20h ; ' '
72 04      jb     short loc_5ED1B4
3C 80      cmp     al, 80h ; '0'
72 02      jb     short loc_5ED1B6
          loc_5ED1B4:      ; CODE XREF: .text:005ED1AE↑j
B0 2E      mov     al, 2Eh ; '.'
          loc_5ED1B6:      ; CODE XREF: .text:005ED1B2↑j
43        inc     ebx
88 02      mov     [edx], al
42        inc     edx
E2 EE      loop   loc_5ED1AA
88 0A      mov     [edx], cl
```

After:

```
89 F1      mov     ecx, esi
          LOCAL_Ready3andLoop3:
8A 03      mov     al, [ebx]
E8 31 01 F0 FF  call   FUNC_LoggerFixCharacter_Patch
88 02      mov     [edx], al
43        inc     ebx
42        inc     edx
E2 F3      loop   LOCAL_Ready3andLoop3
88 0A      mov     [edx], cl
```

I searched the EXE for empty space to place the code sequence “3c 20 72 04 3c 80 72 02 b0 2e c3”. I used the search string “c3 90 90 90 90 90 90 90 90 90 90 90 55”. This is a long NOP sequence (alignment) between a RET and a PUSH.

After editing the EXE I used IDA Pro's text search function to locate the modified code (I had no clue where exactly I placed the function):



After removing the alignment directive and documenting:

```

004ED2F5
004ED2F5 3C 20
004ED2F7 72 04
004ED2F9 3C 80
004ED2FB 72 02
004ED2FD
004ED2FD
004ED2FD B0 2E
004ED2FF
004ED2FF
004ED2FF C3
004ED2FF
004ED2FF

FUNC_LoggerFixCharacter proc near
cmp     al, 20h ; ' '
jb      short LOCAL_Change
cmp     al, 80h ; 'G'
jb      short LOCAL_ChangeDone

LOCAL_Change:
mov     al, 2Eh ; 'Z'

LOCAL_ChangeDone:
retn
FUNC_LoggerFixCharacter endp

```

Looks better, right?

THE C TEST CODE

```
#define _USE_32BIT_TIME_T
#include <windows.h>
#include <stdio.h>
#include <locale.h>
#include <time.h>

#pragma warning( disable : 4996 )

static FILE *logfile = NULL;

enum DIRECTION
{
    RazorToServer, ClientToServer, ServerToClient
};

void LogPacket(DIRECTION PacketDirection, size_t PacketLength, const BYTE *PacketData)
{
    const char *LogText1 = "      0 1 2 3 4 5 6 7 8 9 A B C D E F";
    const char *LogText2 = "      - - - - - - - - - - - - - - - -";

    char LogLine[100];
    char Log8_1[8 * 3 + 1], Log8_2[8 * 3 + 1];

    // We must have an open logfile
    if(logfile == NULL)
        return;

    // Do not log empty packets
    if(PacketLength == 0)
        return;

    // Part 1
    {
        // Get the current time as string
        time_t t = time(NULL);
        GetLocalTime((LPSYSTEMTIME) LogLine); // (use LogLine also for SYSTEMTIME)

        strftime(LogLine, sizeof(LogLine), "%X", localtime(&t));

        // Verify the packet direction
        char *LogDirection1, *LogDirection2;
        switch(PacketDirection)
        {
            case ClientToServer: LogDirection1 = "Client";
                                LogDirection2 = "Server";
                                break;
            case ServerToClient: LogDirection1 = "Server";
                                LogDirection2 = "Client";
                                break;
            default: return;
        }

        fprintf(logfile, "%s.%04u: %s -> %s 0x%02X (Length: %u)\n%s\n%s\n", LogLine,
            ((LPSYSTEMTIME) LogLine)->wMilliseconds, LogDirection1, LogDirection2, PacketData[0],
            PacketLength, LogText1, LogText2);
    }

    // Part 2
    unsigned int perl6counter = 0;
    do
    {
        register unsigned int Log8len, MaxCounter;

        Log8len = 0;
        MaxCounter = PacketLength < 8 ? PacketLength : 8;
        for(register unsigned int Counter = 0; Counter < MaxCounter; Counter++)
        {
            sprintf(Log8_1 + Log8len, "%02X ", PacketData[Counter]); // TEST
            Log8len += 3;
        }
    }
}
```


ABOUT OPTIMAZATION TECHNIQUES

I had lot of problems with fitting the code inside the binary. So I had to relook how I implemented to log function and I rewrote it a few times before all the bytes I needed fitted in.

Remember, I optimized for size, not for speed!

Let's take a look at some techniques I used:

- 1) Replacing "MOV reg32, 0XXXXXXXX" with "PUSH 0xXX / POP reg32"

For example, "MOV EAX, 8" is assembled to "B8 08 00 00 00" (5 bytes)

Now, "PUSH 8 / POP EAX" becomes "6A 08 58" (3 bytes) (but slower!)

- 2) Not cleaning up the stack after every call

This is actually a technique also used by modern compilers, because it's faster and uses less code bytes. I only clean-up the stack inside the MainLoop because inside there the stack is actually important and used with care.

- 3) locale(LC_TIME, "C")

By default, this becomes:

```
...
PUSH offset a__C      68 XX XX XX XX
PUSH 5 ; LC_TIME      6A 05
CALL locale           E8 XX XX XX XX
ADD ESP, 4            83 C4 04
...
a__C DB 'C', 0        43 00
→ 5+2+5+3+2 = 17 bytes
```

I implemented this with:

```
...
PUSH 'C'              6A 43
PUSH ESP              54
PUSH 5 ; LC_TIME      6A 05
CALL locale           E8 XX XX XX XX
...
→ 2+1+2+5 = 10 bytes
```

The trick is that "C" is a string consisting of 1 byte and 1 0-terminator. If you push 'C' (43 hex) on the stack then the value is zero extended on the stack and the stack will actually contain "43 00 00 00". Then ESP is pushed because ESP points to the freshly created string on the stack.

The same trick is applied when calling “localtime(&t)”. Instead of placing the return-value of time(NULL) into a variable and then calling localtime with a reference to that variable, localtime is called directly after pushing EAX and pushing ESP (which points to the pushed EAX). I hope I could explain?

- 4) I also replaced comparisons against a constant with a comparison against a register, this takes one byte less in memory but you have to make sure the register contains the right constant (without adding extra code to put a constant into the register!).
- 5) CMP equals SUB

This is something you should know, the CMP instruction is actually a SUB instruction but with that difference the target register isn't changed. That's why you are seeing Jxx instructions right after a SUB instruction.

```
mainloop: ...
    CMP EAX, 16
    JBE endloop    ; Jump if Below or Equal
    SUB EAX, 16
    JMP mainloop
endloop: ...
```

becomes:

```
mainloop: ...
    SUB EAX, 16
    JA mainloop    ; Jump if Above (=JNBE) (Jump if Not Below or Equal)
endloop: ...
```

NOTE: also the TEST instruction is actually an AND instruction (not used here)

- 6) Remember that the C API and Windows API preserve the EBX, EDI, ESI and EBP registers. Make use of that fact! Never ever trust the contents of ECX or EDX after calling an API. Know that most functions put their return values into EAX and sometimes into the EDX:EAX pair.

Let's take a closer look at the following code inside the main loop:

```
005ED15F 59          pop     ecx
005ED160 8D 75 E4     lea    esi, [ebp+var_10]
005ED163 8D 04 09     lea    eax, [ecx+ecx]
005ED166 8B 7D 18     mov    edi, [ebp+arg_14]
005ED169 50          push   eax
005ED16A 39 C7       cmp    edi, eax
005ED16C 88 26       mov    [esi], ah
005ED16E 76 01       jbe    short LOCAL_Ready2
005ED170 97          xchg   eax, edi
005ED171
005ED171          LOCAL_Ready2:
005ED171 29 CF       sub    edi, ecx
005ED173 76 1D       jbe    short LOCAL_SkipLoop2
005ED175 53          push   ebx
005ED176 01 CB       add    ebx, ecx
005ED178
005ED178          LOCAL_Loop2:
```

If you are looking at that code to understand it, I think you're in hell.

I will help you.

“POP ECX” will put 8 into ECX. It is a constant I pushed on the stack earlier, at address 005ED13B to be exact. ESI points to a string on the stack (Log8_2 in the C Version). EAX is then calculated by adding ECX to itself. $EAX = ECX + ECX = 8 + 8 = 16$. I could use “MOV EAX, 16” (5 bytes) or the “PUSH / POP” trick (also 3 bytes), but I decided to use the LEA version. Then EAX (16) is pushed on the stack **for later use**, it will be popped into ESI at address 005ED192.

So, we know that ECX is 8 and EAX is 16. Then EDI ([EBP+ arg_14] aka PacketSize) is compared against EAX (16). We know that EAX is 16, so AL is 16 and AH is 00. Thus “MOV [ESI], AH” equals “MOV BYTE PTR [ESI], 0” thus meaning “*Log8_2 = 0;” or “Log8_2[0] = 0;”. The “XCHG EAX, EDI” will put the value of 16 into EDI. After this function, when we reach LOCAL_Ready2, the EAX register is no longer used. “SUB EDI, ECX” equals “SUB EDI, 8”, the loop will be skipped if EDI was below or equal to 8.

By the way, EBX is pointing to the packet data.

The readable (but binary longer) version of the code above is:

```
MOV ESI, offset Log8_2
MOV EDI, PacketSize
CMP EDI, 16
MOV BYTE PTR [ESI], 0
JBE LOCAL_Ready2
MOV EDI, 16
LOCAL_Ready2:
CMP EDI, 8
JBE LOCAL_SkipLoop2
SUB EDI, 8
PUSH EBX
ADD EBX, 8
```

WHERE TO ADD THE CODE AND TEXT

Looking for empty space in an executable can be difficult but can also be easy if the executable is compiled by following the rules.

You have to open the EXE with the Explorer Suite and view the section headers.

UoDemo+.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
00000178	00000180	00000184	00000188	0000018C	00000190	00000194	00000198	0000019A	0000019C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	001EC022	00001000	001EC200	00000400	00000000	00000000	0000	0000	60000020
.rdata	00017F68	001EE000	00018000	001EC600	00000000	00000000	0000	0000	40000040
.data	0039ED57	00206000	00037A00	00204600	00000000	00000000	0000	0000	C0000040
.idata	00001826	005A5000	00001A00	0023C000	00000000	00000000	0000	0000	C0000040
UVA_DATA	00003C60	005A7000	00003E00	0023DA00	00000000	00000000	0000	0000	C0000040
.rsrc	00091000	005AB000	00091000	00241800	00000000	00000000	0000	0000	40000040

The Raw Size is the number of bytes a certain section takes inside the file. The Virtual Size is the number of bytes in memory. If the Virtual Size is lower than the Raw Size, then it means there is unused space (if the compiler is following the rules!).

So actually, you can see that have quite some unused space inside the UoDemo executable!

After inserting your code inside that space, do not forget to edit the section headers so it will contain accurate values, which is important if you want to add code later again.

This is a screenshot of the modified section headers:

UoDemo+.exe									
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	001EC200	00001000	001EC200	00000400	00000000	00000000	0000	0000	60000020
.rdata	00018000	001EE000	00018000	001EC600	00000000	00000000	0000	0000	40000040
.data	0039ED57	00206000	00037A00	00204600	00000000	00000000	0000	0000	C0000040
.idata	00001826	005A5000	00001A00	0023C000	00000000	00000000	0000	0000	C0000040
UVA_DATA	00003CD8	005A7000	00003E00	0023DA00	00000000	00000000	0000	0000	C0000040
.rsrc	00091000	005AB000	00091000	00241800	00000000	00000000	0000	0000	40000040

The red squares show the modified virtual sizes.

Note that the “.text” and “.rdata” are now completely full and can no longer grow! If we want to add code in a future patch we either have to either remove code or insert an extra section!