Some prerequisite words: *The networking code is aggressively optimized and this post will be extremely difficult to comprehend without decent knowledge of assembly and following along in a debugger. Below is my analysis of how a chat packet is constructed within the game, which was analyzed on a running instance with OllyDbg. Also, the knowledge contained in post isn't necessarily needed for being able to forge packets across an established in-game connection.*

The process begins by trying to find out where the game grabs the chat from. Logically, to build a chat packet, there needs to be some chat. This should either be the starting point or near to the starting point of building the chat packet to send. Techniques on finding out this starting point are situational. For example, if the chat were entered directly from some input box, it might be wise to breakpoint calls to GetDlgItemText or something similar and follow from there. For this, I chose the approach of setting a breakpoint on sendto and following the call stack backwards. The packet can be fully inspected on the call to sendto as shown below:

Address	Hex dump	ASCII
0F9AAC5C	3A 68 BC 13 00 00 00 00 43 02 84 00 06 00 00 00	:h╝‼....C☻ä.♠...
0F9AAC6C	D2 07 00 00 01 DC 59 59 4E 4E 4E 4E 4E 4E 6D 00	╥•..☺▄YYNNNNNNm.
0F9AAC7C	0F 00 00 00 01 48 45 4C 4C 4F 20 57 4F 52 4C 44	☼...☺HELLO WORLD
0F9AAC8C	21 21 21 27 00 00 00 00 00 00 00 00 CF B0 42 57	!!!'........╧░BW
0F9AAC9C	00 00 00 80 D3 00 9A 0F 01 00 00 00 80 D2 A1 0F	...Ç╙.Ü☼☺...Ç╥í☼
0F9AACAC	61 00 70 00 69 00 2D 00 6D 00 73 00 2D 00 77 00	a.p.i.-.m.s.-.w.

As an aside, this is also a good place to test for things like data checksums or even possible exploits in the protocol. Checking for data checksums is pretty easy — if one of those unknown fields in the packet is a checksum, then modifying the data but keeping the checksum the same will make the receiving client report an error and/or not display the chat. Since it was possible to modify the text and still have it appear on the other side, you can conclude that no data checksum is present (or the receiving code doesn't check it). Additionally, modifying the length of the chat also allows the chat to get through. This is more interesting because there actually is a checksum on the length appended to the packets. Having looked over the recvfrom code, which won't be discussed in this post, I will just say that the checksum is not checked by the game. There are other things to check for like overflows which can be invoked by setting the chat length to 0xFF and sending chat greater than 0xFF in size to see if the game parses the packet correctly or not. Overall, I didn't find anything too interesting that didn't solely cause my own instance (the one sending the packets) of the game to crash.

Back on topic however, the call stack at sendto is shown below:

Address	Stack	Procedure	Called from	Frame
0018D128	6E2A61FA	<JMP.&WS2_32.#20>	dpwsockx.6E2A61F5	0018D178
0018D17C	6E2A7965	dpwsockx.6E2A60DD	dpwsockx.6E2A7960	0018D178
0018D190	6E2E540D	Includes dpwsockx.6E2A7965	DPLAYX.6E2E540A	0018D18C
0018D1C4	6E2E5ADB	DPLAYX.6E2E5265	DPLAYX.6E2E5AD6	0018D1C0
0018D1F8	6E2E6552	DPLAYX.6E2E59F4	DPLAYX.6E2E654D	0018D1F4
0018DA38	6E2DCBBF	DPLAYX.6E2E6333	DPLAYX.6E2DCBBA	0018DA34

The goal is to find out where the packet is built, so the best place to look is the furthest back. Checking DPLAYX.6E2E6333 reveals the following data at [ESP+14]:

```
Address   Hex dump                                         ASCII
029EFCA0  43 02 98 02 06 00 00 00 D2 07 00 00 01 DC 59 59  C☻ÿ☻♠...╥•..☺▄YY
029EFCB0  4E 4E 4E 4E 4E 4E 31 00 0F 00 00 00 01 48 45 4C  NNNNNN1.☼...☺HEL
029EFCC0  4C 4F 20 57 4F 52 4C 44 21 21 21 21 00 00 00 00  LO WORLD!!!!....
029EFCD0  00 FC 9E 02 00 00 00 00 41 00 00 00 31 00 00 00  .ⁿ₧☻....A...1...
029EFCE0  40 23 31 50 6C 61 79 65 72 32 3A 20 48 45 4C 4C  @#1Player2: HELL
029EFCF0  4F 20 57 4F 52 4C 44 21 21 21 21 00 40 FC 9E 02  O WORLD!!!!.@ⁿ₧☻
029EFD00  09 00 00 00 00 00 00 1E 31 00 00 00 31 00 00 00  .......▲1...1...
029EFD10  00 00 00 00 00 00 00 10 FF FF FF FF 00 00 00 00  .......►    ....
```

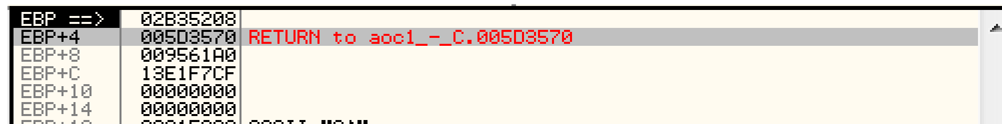
which shows a chat packet containing the header. This means that it is necessary to check even further back. The next step is to look at the call from DPLAYX.6E2DCBBA. The call is shown below:

```
6E2DCBAC    6A 01          PUSH 1
6E2DCBAE    57             PUSH EDI
6E2DCBAF    53             PUSH EBX
6E2DCBB0    FF75 14        PUSH DWORD PTR SS:[EBP+14]
6E2DCBB3    FF75 F8        PUSH DWORD PTR SS:[EBP-8]
6E2DCBB6    FF75 1C        PUSH DWORD PTR SS:[EBP+1C]
6E2DCBB9    50             PUSH EAX
6E2DCBBA    E8 74970000    CALL DPLAYX.6E2E6333
```

Setting a breakpoint on this shows that the fully built packet is stored in EBX. Tracing backwards up the function, the chat is loaded into EBX by

```
6E2DCB1E    8B5D 18        MOV EBX,DWORD PTR SS:[EBP+18]
```

which means it's necessary to go even further backwards since the packet is still fully built at this point. OllyDbg isn't too great at keeping track of call stacks when the function isn't called directly. Setting a breakpoint at the top of the function and checking the call stack showed that nothing called it, which definitely is not correct. The easiest approach is to inspect the stack manually for the point of return. When EBP is set up, the stack looks like the following:



That address leads to the following function, with the call at 0x005D356D:

```
005D3540  /$ 8B91 10470000  MOV EDX,DWORD PTR DS:[ECX+4710]
005D3546  |. 33C0           XOR EAX,EAX
005D3548  |. 85D2           TEST EDX,EDX
005D354A  |. 75 24          JNZ SHORT aoc1_-_C.005D3570
005D354C  |. 8B5424 14      MOV EDX,DWORD PTR SS:[ESP+14]
005D3550  |. A1 84027900    MOV EAX,DWORD PTR DS:[790284]
005D3555  |. 52             PUSH EDX
005D3556  |. 8B5424 14      MOV EDX,DWORD PTR SS:[ESP+14]
005D355A  |. 52             PUSH EDX
005D355B  |. 8B5424 14      MOV EDX,DWORD PTR SS:[ESP+14]
005D355F  |. 8B08           MOV ECX,DWORD PTR DS:[EAX]
005D3561  |. 52             PUSH EDX
005D3562  |. 8B5424 14      MOV EDX,DWORD PTR SS:[ESP+14]
005D3566  |. 52             PUSH EDX
005D3567  |. 8B5424 14      MOV EDX,DWORD PTR SS:[ESP+14]
005D356B  |. 52             PUSH EDX
005D356C  |. 50             PUSH EAX
005D356D  |. FF51 68        CALL DWORD PTR DS:[ECX+68]
```

```
005D3570      C2 1400          RETN 14
```

This is beginning to lead in the right direction since the call is made directly from the game instead of an additional library. Additionally, this function is **only** called when chat is to be sent. All other ones prior to this were called when **any** packet was to be sent. The chat is loaded into EDX at the following instruction:

```
005D355A   |. 52               PUSH EDX
```

The packet is still fully built at this point, but the search is almost over. The problem and approach are still the same, but now its been heavily isolated. The call stack from this point should contain functions that deal with collecting data and constructing the appropriate packet, instead of networking functions in the DirectPlay and Winsock libraries. Taking the same approach as before and setting a breakpoint at the top of the function shows the following call stack:

Address	Stack	Procedure	Called from	Frame
0018DA8C	005D7E7E	aoc1_-_C.005D3540	aoc1_-_C.005D7E79	
0018DAD4	005D69CF	aoc1_-_C.005D7BC0	aoc1_-_C.005D69CA	
0018DC10	005D6798	aoc1_-_C.005D67A0	aoc1_-_C.005D6793	
0018DC30	004FDF44	aoc1_-_C.005D6720	aoc1_-_C.004FDF3F	
0018DF98	0054E5CE	Includes aoc1_-_C.004FDF44	aoc1_-_C.0054E5C8	
0018DFE8	0054DCA9	aoc1_-_C.0054E4B0	aoc1_-_C.0054DCA4	
0018E000	752462FA	Includes aoc1_-_C.0054DCA9	USER32.752462F7	
0018E02C	75246D3A	? USER32.752462D7	USER32.75246D35	
0018E0A4	752477C4	? USER32.75246C83	USER32.752477BF	0018E0A0
0018E104	75247BCA	? USER32.752476D7	USER32.75247BC5	0018E100

Looking at 0x005DC6720, one of the further functions down the call stack, begins to show some promise. When this function is entered, the chat string is held in the EAX register. It can be modified and the changes carry on through to the sendto function. This means that it's not a function working on a temporary copy of the buffer, but that it holds the "real" one that will be written into the chat packet. This seems like a good starting point since there is no sign of any packet header being built. Additionally, inspecting the other functions up the call stack from 0x005DC720 shows that some of them display debug messages dealing with packets and chat headers. This is also a good sign that the real reversing should begin here.

What I personally prefer doing is following the code flow in OllyDbg then highlighting the executed code path in IDA Pro. The assembly listings from IDA will be shown in the code blocks to follow. The function starts at 0x005D6720:

```
.text:005D6720 ; int __stdcall send_normal(char *Str)
.text:005D6720 send_normal     proc near               ; CODE XREF:
sub_4FD360+BDFp
.text:005D6720                                         ; sub_5A6C90+154p ...
.text:005D6720
.text:005D6720 broadcast       = dword ptr -0Ch
.text:005D6720 var_8           = dword ptr -8
.text:005D6720 var_4           = word ptr -4
.text:005D6720 Str             = dword ptr  4
.text:005D6720
.text:005D6720                 sub     esp, 0Ch
.text:005D6723                 mov     eax, 59595959h
.text:005D6728                 push    esi
.text:005D6729                 mov     [esp+10h+broadcast], eax
```

```
.text:005D672D                  mov     esi, ecx
.text:005D672F                  mov     [esp+10h+var_8], eax
.text:005D6733                  mov     [esp+10h+var_4], ax
.text:005D6738                  mov     eax, [esi+10E4h] ; number of other
players in game
.text:005D673E                  test    eax, eax
.text:005D6740                  jnz     short loc_5D6780
```

The first thing to notice is that no stack frame is set up. This is the beginning of a lot of incoming optimized code. Everything function to come will not set up a stack frame and will work directly relative to ESP. The next thing to notice is that the value 0×59595959 is moved into EAX, which is then moved into [ESP+0x4]. Why 0×59595959? Looking back at the previous example packets, there were eight bytes devoted to who can see the packet. These were set to either Y or N depending on whether the target player is supposed to display the message or not. 0×59 happens to be the ASCII code for 'Y', so the beginning of this function sets up to send a message to all players, i.e. the broadcast field in the packet will be YYYYYYYY (or 0×59 0×59 … 0×59). [ESI+0x10E4] holds the number of other players in the game. This is moved into EAX and checked for zero. Assuming there is at least one additional player, the code jumps to 0x005D6780. The case where no other player is in game won't be discussed in-depth except that in that case no packets are sent and the text is only displayed on the screen. Continuing on at 0x005D6780 is the following block:

```
.text:005D6780 loc_5D6780:                                 ; CODE XREF:
send_normal+20j
.text:005D6780                  mov     eax, [esp+14h]
.text:005D6784                  mov     edx, [esi+10E0h]
.text:005D678A                  lea     ecx, [esp+10h+broadcast]
.text:005D678E                  push    eax             ; chat string
.text:005D678F                  push    ecx             ; broadcast audience
.text:005D6790                  push    edx             ; player_index
.text:005D6791                  mov     ecx, esi
.text:005D6793                  call    sub_5D67A0
.text:005D6798                  pop     esi
.text:005D6799                  add     esp, 0Ch
.text:005D679C                  retn    4
.text:005D679C send_normal      endp
```

Looking at this is just a matter of stepping over in OllyDbg. The function at 0x005D67A0 is called with the player index, the broadcast audience, and the chat message as parameters. This function is a bit longer and more complicated. The first block is shown below:

```
.text:005D67A0 ; int __stdcall sub_5D67A0(int player_index, int chat_message,
char *Str)
.text:005D67A0 sub_5D67A0       proc near               ; CODE XREF:
.text:004A8961p
.text:005D67A0                                          ; sub_4A8970+5Bp ...
.text:005D67A0
.text:005D67A0 var_114          = byte ptr -114h
.text:005D67A0 var_113          = byte ptr -113h
.text:005D67A0 chat_length      = dword ptr -108h
.text:005D67A0 var_104          = byte ptr -104h
.text:005D67A0 Dest             = byte ptr -103h
```

```
.text:005D67A0 var_4           = byte ptr -4
.text:005D67A0 player_index    = dword ptr  4
.text:005D67A0 chat_message    = dword ptr  8
.text:005D67A0 Str             = dword ptr  0Ch
.text:005D67A0
.text:005D67A0                 sub     esp, 114h
.text:005D67A6                 xor     eax, eax
.text:005D67A8                 push    ebx
.text:005D67A9                 push    ebp
.text:005D67AA                 mov     ebp, ecx
.text:005D67AC                 push    esi
.text:005D67AD                 mov     ecx, [esp+120h+player_index]
.text:005D67B4                 push    edi
.text:005D67B5                 mov     ax, [ebp+12DCh] ; maximum number of
players
.text:005D67BC                 cmp     ecx, eax
.text:005D67BE                 jbe     short loc_5D67D2
```

This code will be tricky to go through because things will be referenced through EBP and ESP. The important immediate thing to note is that EBP takes the value of ECX, which had ESI moved into it prior to the function call. As usual, no stack frame is set up. All that this code does is compare the index of the player sending the chat (stored in [EBP+0x12DC]) with the maximum number of players allowed in the game. The error condition is that the player index is greater than the maximum number allowed (e.g. player 9 is trying to send a message in an 8 player game). Assuming no error, the jump to 0x005D67D2 is taken. This is a pretty small block which performs an unusual check:

```
.text:005D67D2 loc_5D67D2:                             ; CODE XREF:
sub_5D67A0+1Ej
.text:005D67D2                 mov     ebx, [esp+124h+Str]
.text:005D67D9                 push    ebx             ; Str
.text:005D67DA                 call    _atoi
.text:005D67DF                 mov     esi, 1
.text:005D67E4                 add     esp, 4
.text:005D67E7                 cmp     [ebp+12DCh], si ; [ebp+12DCh] holds
number of players
.text:005D67EE                 mov     edi, eax
.text:005D67F0                 jb      loc_5D6883
```

The purpose of the call to atoi is to check whether a taunt has been entered. Taunts are represented in chat purely by numbers, so atoi will return non-zero if that is the case. The code then continues to check whether more than one player is in the game. This is to determine whether it’s necessary to build a packet or not. Assuming more than one player is in the game, the jump to 0x005D6883 is not taken. A loop is then entered. The important parts of the loop are reproduced below. These are the instructions that are executed when more than one player is to receive some chat.

```
.text:005D67F6 loc_5D67F6:                             ; CODE XREF:
sub_5D67A0+DDj
.text:005D67F6                 push    esi             ; begin loop to see
who is allowed to see message
.text:005D67F7                 mov     ecx, ebp
```

```
.text:005D67F9                 call    can_player_see
.text:005D67FE                 test    eax, eax
.text:005D6800                 jnz     short loc_5D680E ; player allowed to
see message
.text:005D6802                 push    esi             ; player number
.text:005D6803                 mov     ecx, ebp
.text:005D6805                 call    sub_5D9720
.text:005D680A                 test    eax, eax
.text:005D680C                 jz      short loc_5D686C
.text:005D680E
.text:005D680E loc_5D680E:                             ; CODE XREF:
sub_5D67A0+60j
.text:005D680E                 mov     ecx, [esp+124h+chat_message] ; player
allowed to see message
.text:005D6815                 cmp     byte ptr [esi+ecx], 59h
.text:005D6819                 jnz     short loc_5D686C
.text:005D681B                 mov     edx, dword_7912A0 ; jump not taken,
player allowed to see message
.text:005D6821                 mov     [esp+esi+124h+var_113], 59h ; allowed
flag
.text:005D6826                 mov     eax, [edx+424h]
.text:005D682C                 test    eax, eax
.text:005D682E                 jz      short loc_5D6871
.text:005D6871
.text:005D6871 loc_5D6871:                             ; CODE XREF:
sub_5D67A0+8Ej
.text:005D6871                                         ; sub_5D67A0+98j ...
.text:005D6871                 xor     eax, eax
.text:005D6873                 inc     esi             ; see if next player
is allowed to see message
.text:005D6874                 mov     ax, [ebp+12DCh] ; maximum number of
players in game
.text:005D687B                 cmp     esi, eax
.text:005D687D                 jbe     loc_5D67F6      ; begin loop to see
who is allowed to see message
.text:005D6883
.text:005D6883 loc_5D6883:                             ; CODE XREF:
sub_5D67A0+50j
.text:005D6883                 mov     eax, [ebp+10E4h] ; total number of
players in game
.text:005D6889                 test    eax, eax
.text:005D688B                 jnz     loc_5D6915
```

The can_player_see function at 0x005D96E0 just returns 1 or 0 depending on whether the player can see the message. What this loop basically does is decide who is going to see this message. The YYY…Y buffer that was passed in to this function gets modified here according to which player can see the message. The following two instructions set the appropriate byte:

```
.text:005D6821                 mov     [esp+esi+124h+var_113], 59h ; allowed
flag
```

or

```
.text:005D686C                 mov     [esp+esi+124h+var_113], 4Eh ; not
allowed flag
```

The loop continues for eight iterations, the maximum number of players in the game. Once this loop is done, the part of the packet which will hold who can see the message is ready. Once the loop exits, the following code blocks are executed:

```
.text:005D6883 loc_5D6883:                                   ; CODE XREF:
sub_5D67A0+50j
.text:005D6883                 mov     eax, [ebp+10E4h] ; total number of
players in game
.text:005D6889                 test    eax, eax
.text:005D688B                 jnz     loc_5D6915
.text:005D6915 ; -------------------------------------------------------------
---------------
.text:005D6915
.text:005D6915 loc_5D6915:                                   ; CODE XREF:
sub_5D67A0+EBj
.text:005D6915                                               ; sub_5D67A0+164j
.text:005D6915                 mov     ecx, [ebp+10E0h]
.text:005D691B                 cmp     [esp+ecx+124h+var_113], 59h
.text:005D6920                 jnz     short loc_5D6957 ; EDI holds chat
string
.text:005D6922                 mov     eax, [esp+124h+player_index]
.text:005D6929                 mov     edx, [ebp+12CCh] ; username sending
chat
.text:005D692F                 mov     ecx, [ebp+18h]
.text:005D6932                 push    0
.text:005D6934                 push    0
.text:005D6936                 push    eax
.text:005D6937                 shl     eax, 7
.text:005D693A                 add     edx, eax          ; EDX holds username
.text:005D693C                 push    ebx               ; EBX holds chat
.text:005D693D                 push    edx
.text:005D693E                 call    sub_5E2780
.text:005D6943                 mov     eax, dword_78BF34
.text:005D6948                 push    ebx
.text:005D6949                 push    offset aLocalChatAddS ; "Local chat
add: %s"
.text:005D694E                 push    eax
.text:005D694F                 call    nullsub_1         ; debug function?
.text:005D6954                 add     esp, 0Ch
.text:005D6957
.text:005D6957 loc_5D6957:                                   ; CODE XREF:
sub_5D67A0+180j
.text:005D6957                 mov     edi, ebx          ; EDI holds chat
string
.text:005D6959                 or      ecx, 0FFFFFFFFh
.text:005D695C                 xor     eax, eax
.text:005D695E                 repne scasb               ; Calculates length of
string
.text:005D6960                 not     ecx
.text:005D6962                 dec     ecx               ; ECX holds number of
characters
.text:005D6963                 cmp     ecx, 0FFh
.text:005D6969                 jbe     short loc_5D6972 ; Calculates length
of string again
```

There isn't anything too special about this block. There is a call to 0x005E2780, which is a huge and complicated function. Fortunately, it doesn't do anything related to modifying the chat or building a packet — so it doesn't have to be analyzed. Other than that there is nothing going on except for calculating the length of the string. This is compared against 0xFF, which is the maximum number of characters allowed. Normal execution continues into the following blocks:

```
.text:005D6972 ; ---------------------------------------------------------------
---------------
.text:005D6972
.text:005D6972 loc_5D6972:                             ; CODE XREF:
sub_5D67A0+1C9j
.text:005D6972                 mov     edi, ebx        ; Calculates length of
string again
.text:005D6974                 or      ecx, 0FFFFFFFFh
.text:005D6977                 xor     eax, eax
.text:005D6979                 repne scasb
.text:005D697B                 not     ecx
.text:005D697D                 dec     ecx
.text:005D697E
.text:005D697E loc_5D697E:                             ; CODE XREF:
sub_5D67A0+1D0j
.text:005D697E                 mov     dl, byte ptr [esp+124h+player_index]
.text:005D6985                 mov     [esp+124h+chat_length], ecx ; Chat
length added to buffer
.text:005D6989                 inc     ecx
.text:005D698A                 lea     eax, [esp+124h+Dest]
.text:005D698E                 push    ecx             ; Count
.text:005D698F                 push    ebx             ; Source
.text:005D6990                 push    eax             ; Dest
.text:005D6991                 mov     [esp+130h+var_114], dl ; Player number
who sent message
.text:005D6995                 call    _strncpy
.text:005D699A                 mov     ecx, [ebp+1E3Ch] ; 3?
.text:005D69A0                 mov     edx, [esp+130h+chat_length]
.text:005D69A4                 add     esp, 0Ch
.text:005D69A7                 cmp     ecx, 3
.text:005D69AA                 setz    cl              ; CL = 1
.text:005D69AD                 add     edx, 16h
.text:005D69B0                 push    0
.text:005D69B2                 lea     eax, [esp+128h+var_114]
.text:005D69B6                 push    edx             ; Chat length + 0x16
.text:005D69B7                 push    eax             ; Partial packet.
Contains chat flag, chat length, who is allowed to see, and message
.text:005D69B8                 mov     [esp+130h+var_104], cl
.text:005D69BC                 push    43h
.text:005D69BE                 push    0
.text:005D69C0                 mov     ecx, ebp
.text:005D69C2                 mov     [esp+138h+var_4], 0
.text:005D69CA                 call    sub_5D7BC0
.text:005D69CF                 mov     ecx, [ebp+1608h]
.text:005D69D5                 mov     esi, eax
.text:005D69D7                 push    offset aTxchat  ; "TXChat()"
.text:005D69DC                 push    esi
.text:005D69DD                 call    sub_5DF900
.text:005D69E2                 mov     eax, esi
```

```
.text:005D69E4                 pop     edi
.text:005D69E5                 pop     esi
.text:005D69E6                 pop     ebp
.text:005D69E7                 pop     ebx
.text:005D69E8                 add     esp, 114h
.text:005D69EE                 retn    0Ch
.text:005D69EE sub_5D67A0      endp
```

Prior to entering this code block the packet buffer contains only who is allowed to see the message. The actual packet buffer is stored relative to ESP and the data will be written directly there. There are some parts that are difficult to analyze like magic values appearing out of nowhere, i.e., [EBP+0x1E3C] holding the value of 3. These don't affect understanding the code too much unless they play some vital role in how the actual packet will be built (the value is a flag parameter for a function, etc.). With the code above, the the CL register is set to 1 since 3 == 3, and is written into the packet buffer. Fortunately, one beneficial thing about this optimized code is that it's easy to see where the packet is being built. The writes into [ESP+xx] hold the buffer for the packet, which can be verified by inspecting it in the dump with OllyDbg. Checking all places where this occurs and seeing what is being written, the packet will have the following fields before entering the function at 0x005D7BC0: A field set to 0×1 (the CL value being written in), who is allowed to see the chat, the chat message itself, and an additional special value 0xDC. This value is always found in the same field and always has the same value. A view of the packet from the dump is shown below:

```
0018DAD8 00 00 00 00 43 00 00 00 FC DA 18 00 24 00 00 00  ....C...ⁿ┌↑.$...
0018DAE8 00 00 00 00 01 00 00 00 68 49 E5 02 00 00 00 00  ....☺...hIσ☻....
0018DAF8 00 B1 C8 02 01 DC 59 59 4E 4E 4E 4E 4E 4E 3F 00  .▒╚☻☺▄YYNNNNNN?.
0018DB08 0E 00 00 00 01 48 45 4C 4C 4F 20 57 4F 52 4C 44  ♫...☺HELLO WORLD
0018DB18 21 21 21 00 00 00 00 00 00 00 00 00 80 DC 54 00  !!!.........Ç▄T.
0018DB28 38 DB 18 00 54 DB 18 00 30 02 2B 75 51 6D 24 75  8█↑.T█↑.0☻+uQm$u
0018DB38 A0 DB 18 00 91 6D 24 75 51 6D 24 75 C0 20 3D 97  á█↑.æm$uQm$u└ =ù
0018DB48 B0 00 00 00 C0 B2 F3 00 70 FE FF FF 24 00 00 00  ░...└▓≤.p■  $...
0018DB58 01 00 00 00 00 00 00 00 00 00 00 00 70 00 00 00  ☺...........p...
0018DB68 FF FF FF FF FF FF FF FF E9 6C 24 75 91 6D 24 75          Θl$uæm$u
```

The call to 0x005D7BC0 will add the remaining fields of the packet (the two supposed counter values, and fields marked as unknown in the first post). Since this function is pretty big, I won't duplicate the entire thing here, but only relevant parts — this post is meant to be followed along with in a disassembler after all. This function allocates a new block of memory and returns the entire packet to send (excluding the fields added by DirectPlay's networking code). The "second counter" bytes are written in by the instructions listed below:

```
.text:005D7C1A loc_5D7C1A:                             ; CODE XREF: sub_5D7BC0+39j
.text:005D7C1A                 push    0Ch             ; unsigned int
.text:005D7C1C                 call    ??2@YAPAXI@Z    ; operator new(uint)
.text:005D7C21                 mov     edx, [esp+34h+arg_C]
.text:005D7C25                 mov     ecx, [ebp+4714h] ;time stamp
.text:005D7C2B                 mov     edi, eax
.text:005D7C2D                 mov     al, [ebp+1DD0h] ; 1
.text:005D7C33                 add     esp, 4
.text:005D7C36                 add     ecx, edx
.text:005D7C38                 test    al, al
.text:005D7C3A                 mov     [esp+30h+Memory], edi
.text:005D7C3E                 mov     [ebp+4714h], ecx ; 0x51E
.text:005D7C44                 jz      short loc_5D7C7A
.text:005D7C46                 mov     ecx, [ebp+1DA0h] ; Retrieve second counter
```

```
.text:005D7C4C                 mov     [edi+8], ecx     ; Write value into
packet
.text:005D7C4F                 mov     eax, [ebp+10E0h] ; player position
.text:005D7C55                 mov     ecx, [ebp+1DA0h] ; second counter
value
.text:005D7C5B                 mov     [ebp+eax*4+1DE4h], ecx
.text:005D7C62                 mov     eax, [ebp+1DA0h] ; Get second counter
value
.text:005D7C68                 inc     eax              ; Increment it for
next packet
.text:005D7C69                 mov     [ebp+1DA0h], eax ; Write new value
back in
.text:005D7C6F                 mov     eax, 0Ch
.text:005D7C74                 mov     [esp+30h+var_20], eax
.text:005D7C78                 jmp     short loc_5D7C89
.text:005D7C89
.text:005D7C89 loc_5D7C89:                              ; CODE XREF:
sub_5D7BC0+B8j
.text:005D7C89                 lea     ecx, [eax+edx]   ; size of header +
data
.text:005D7C8C                 cmp     ecx, 0FA0h       ; maximum packet size
.text:005D7C92                 mov     [esp+30h+var_1C], ecx ; write in full
size
.text:005D7C96                 jbe     short loc_5D7CB3
```

Again there are some magic values that seemingly appear out of nowhere. There are some more familiar ones like [EBP+0x10E0], which was shown in previous functions as storing the player index. The value of the counter is kept at [EBP+0x1DA0], which is written into the packet through EDI. The timestamp bytes are also written in, which are apparently kept at [EBP+0x4714]. The code in the second block just grabs the counter, writes it to the packet buffer, and increments it for the next time it is used. The next block writes in the size of the packet so far. This will be a check to make sure that the header and data do not exceed the maximum length allowed for a chat packet. The next field is an unknown field that is written by:

```
.text:005D7CB3
.text:005D7CB3 loc_5D7CB3:                              ; CODE XREF:
sub_5D7BC0+D6j
.text:005D7CB3                 xor     eax, eax
.text:005D7CB5                 mov     byte ptr [edi+1], 0
.text:005D7CB9                 mov     byte ptr [edi], 0
.text:005D7CBC                 mov     al, [ebp+1E74h] ; Number of players
.text:005D7CC2                 mov     ecx, [ebp+10C8h] ; 2?
.text:005D7CC8                 lea     ebx, [ebp+1DA8h] ; 4?
.text:005D7CCE                 add     eax, ecx
.text:005D7CD0                 mov     [esp+30h+var_14], eax
.text:005D7CD4                 mov     [edi+4], eax     ; 06 00 00 00 added
here
```

This writes in 06 00 00 00 to the packet, which comes from adding [EBP+0x10C8] and [EBP+0x1DA8] together. Where these two things come from, I'm not entirely quite sure. However, they remain constant across all packets regardless of size, player index, and so on, so it's not too important for being able to forge packets. The other counter value can be found at

```
.text:005D7D22                 mov     ecx, [esp+30h+var_14] ; Full packet
size
.text:005D7D26                 push    ecx
.text:005D7D27                 mov     ecx, ebp
.text:005D7D29                 call    sub_5D7B50
.text:005D7D2E                 mov     [edi+1], al      ; Counter value
.text:005D7D31                 mov     ecx, [ebp+1E3Ch] ; 3?
.text:005D7D37                 cmp     ecx, 5
.text:005D7D3A                 jz      short loc_5D7D6C
.text:005D7D3C                 test    al, al
.text:005D7D3E                 jnz     short loc_5D7D6C
```

in the call to 0x005D7B50, which returns the counter byte. This can be seen here

```
.text:005D7B6F ; -------------------------------------------------------------
---------------
.text:005D7B6F
.text:005D7B6F loc_5D7B6F:                             ; CODE XREF:
sub_5D7B50+Cj
.text:005D7B6F                 mov     al, byte ptr dword_790FAC ; Get
counter byte
.text:005D7B74                 inc     al               ; Increment counter
byte
.text:005D7B76                 cmp     al, 0FFh         ; Compare with max
allowed
.text:005D7B78                 mov     byte ptr dword_790FAC, al ; Write
value back
.text:005D7B7D                 jb      short loc_5D7BA3
```

which just gets the byte, increments it, and resets it if it's greater than 0xFF. The last important block of code is shown below:

```
.text:005D7D6C
.text:005D7D6C loc_5D7D6C:                             ; CODE XREF:
sub_5D7BC0+17Aj
.text:005D7D6C                                         ; sub_5D7BC0+17Ej
.text:005D7D6C                 mov     eax, [esp+30h+var_1C] ; Full packet
size
.text:005D7D70                 mov     [edi], bl        ; Packet type 60
.text:005D7D72                 inc     eax
.text:005D7D73                 push    eax              ; unsigned int
.text:005D7D74                 call    ??2@YAPAXI@Z     ; operator new(uint)
.text:005D7D79                 mov     ebx, eax         ; Buffer size of the
packet header + data
.text:005D7D7B                 mov     eax, [esp+34h+var_20] ; 0xC?
.text:005D7D7F                 mov     ecx, eax
.text:005D7D81                 mov     esi, edi
.text:005D7D83                 mov     edx, ecx
.text:005D7D85                 mov     edi, ebx
.text:005D7D87                 shr     ecx, 2
.text:005D7D8A                 rep movsd                ; Packet minus session
key is written here
.text:005D7D8C                 mov     ecx, edx
.text:005D7D8E                 add     esp, 4
.text:005D7D91                 and     ecx, 3
```

```
.text:005D7D94                 mov     [esp+30h+var_10], ebx
.text:005D7D98                 rep movsb
.text:005D7D9A                 mov     ecx, [esp+30h+arg_C]
.text:005D7D9E                 test    ecx, ecx
.text:005D7DA0                 jz      short loc_5D7DB7
```

Prior to entering this block, the packet is fully built. This is responsible for creating the buffer for the full packet to send and copying the contents in there. There's not much more to it at this point. Looking down the function, it will call into another function which in turn calls into DirectPlay (seen earlier in this post).

The entire explanation has basically been following across the call stack and seeing how the chat input is transformed from it's initial stage into a fully built packet. The specific knowledge gained may not have been too great — there are still those unknown fields — however, it was useful to see that there are special checksums or integrity checks performed on the data which will go in to the packet. Also, it was possible to learn how those counters in the packet function, how their values change, and if their values have any effect on how data will be transmitted. I don't expect this explanation to be incredibly clear since it was written over a period of a few days, but these are some notes that I wanted to publish online both for my records and as a demonstration of how difficult and confusing it can be to even reverse one specific type of packet in a protocol. Unfortunately, the code is extremely optimized so this post may not serve as a great starting point into reversing protocols altogether, but the general idea should be the same of finding out where and how certain input is transformed into a transmittable packet.