Reverse Engineered Breakdown of the Client

Christopher Brooks, Adam Waldie

Professor:
Jonathan Weissman

April 17th, 2023

Subroutine Location Table

Client

main	sub_0040167d
parseCommand	sub_004019c7
cmdShutdown	sub_00401b40
cmdInform	sub_00401b59
cmdProc	sub_00401f7d
cmdUpload	sub_0040218a
cmdDownload	sub_00402296

Shared

recvMsg	sub_00401531
sendMsg(SOCKET sock, char* msg, int len)	sub_0040148a
sendMsg(SOCKET sock, char* msg)	sub_00401460
encryptBuffer	sub_004015ff

Encryption Function Breakdown

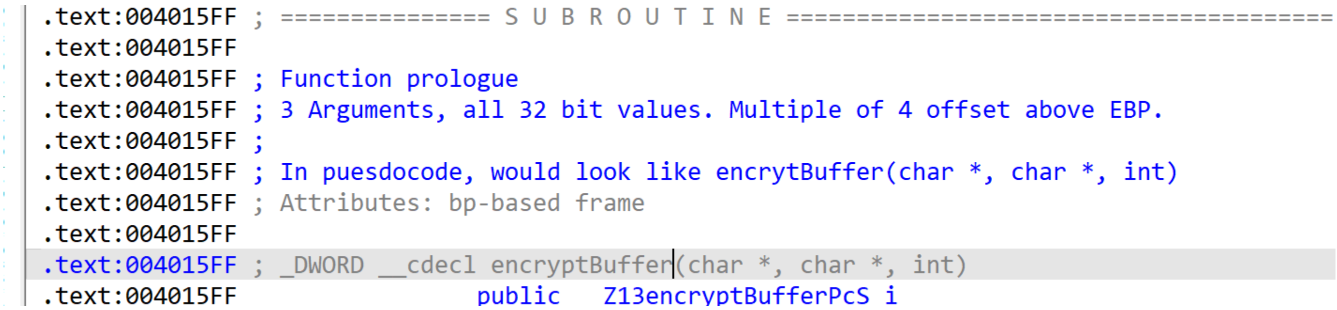
Task 2.1:
Completely annotate the subroutine(s) responsible for your encryption/encoding algorithms.

- Appropriately naming subroutine(s)
- Naming all variables, and what they do within the IDB

Our encryption function is located at sub_004015ff, titled encryptBuffer. This function is shared between both the client and server executables, as both use this function to encrypt the data sent between them. Let's look at the function in IDA before we look at the source code.

Arguments

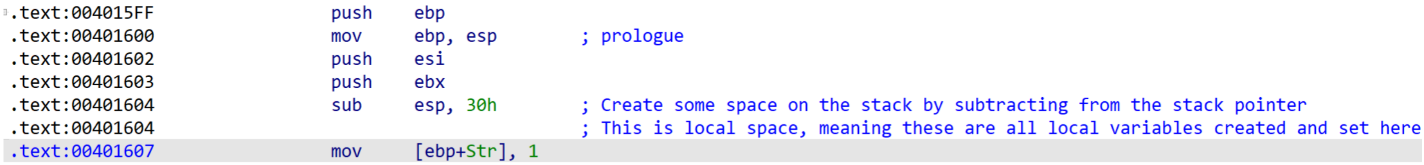
This function takes in 3 arguments. According to the IDA disassembly itself, these are a char *, another char * and an int. On the stack these values would be located above the base pointer (EBP + some offset, going towards a higher memory location). Since all of these arguments are 32 bit values the offsets would either be the base pointer plus 0x4, 0x8 or 0xc. This is what the function disassembly looks like in IDA:



```
.text:004015FF ; =============== S U B R O U T I N E =======================================
.text:004015FF
.text:004015FF ; Function prologue
.text:004015FF ; 3 Arguments, all 32 bit values. Multiple of 4 offset above EBP.
.text:004015FF ;
.text:004015FF ; In puesdocode, would look like encrytBuffer(char *, char *, int)
.text:004015FF ; Attributes: bp-based frame
.text:004015FF
.text:004015FF ; _DWORD __cdecl encryptBuffer(char *, char *, int)
.text:004015FF                 public  Z13encryptBufferPcS i
```

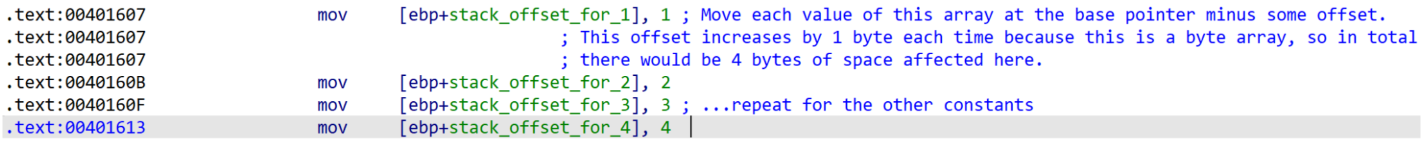
Local Variables

There are lots of local variables used for this function. The first is the constants for our encryption function, which we will need to make space on the stack for. The disassembly shows that the code creates space on the stack by subtracting 0x30 from the base pointer (at 0x00401604). This essentially "allocates" the space on the stack that we'll be using in the next few lines.



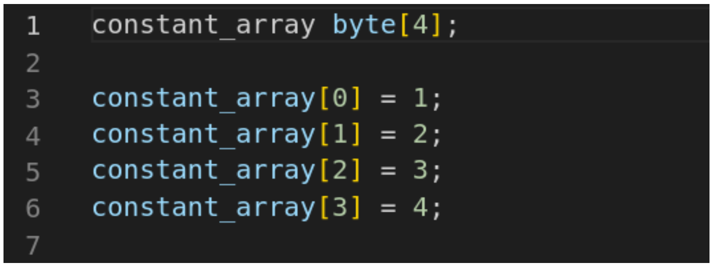
```
.text:004015FF                 push    ebp
.text:00401600                 mov     ebp, esp        ; prologue
.text:00401602                 push    esi
.text:00401603                 push    ebx
.text:00401604                 sub     esp, 30h        ; Create some space on the stack by subtracting from the stack pointer
.text:00401604                                         ; This is local space, meaning these are all local variables created and set here
.text:00401607                 mov     [ebp+Str], 1
```

The next part that it does is create an array with some constant values in it. These array values are moved one at a time on the stack via the `mov` instruction. As you can see starting at 0x00401607, we move the values 1 through 4 into the new array. This array is a local variable which we will use later for encryption.



```
.text:00401607                 mov     [ebp+stack_offset_for_1], 1 ; Move each value of this array at the base pointer minus some offset.
.text:00401607                                         ; This offset increases by 1 byte each time because this is a byte array, so in total
.text:00401607                                         ; there would be 4 bytes of space affected here.
.text:0040160B                 mov     [ebp+stack_offset_for_2], 2
.text:0040160F                 mov     [ebp+stack_offset_for_3], 3 ; ...repeat for the other constants
.text:00401613                 mov     [ebp+stack_offset_for_4], 4
```

The pseudocode for this part would simply look like this if it were written out one by one:



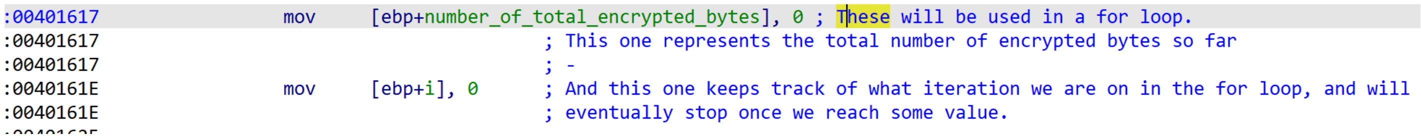
```
constant_array byte[4];

constant_array[0] = 1;
constant_array[1] = 2;
constant_array[2] = 3;
constant_array[3] = 4;

```

This array (or string) is referred to as “secret” in our source code and is basically just what we are using as the key for the encoding.

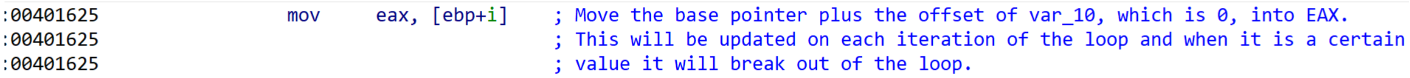
We also create some other variables which are initially just set to 0. These will be used in the upcoming encryption for loop. They are both defined starting at 0x00401617. Refer to the screenshot annotation for what each of these are used for:



```
:00401617                 mov     [ebp+number_of_total_encrypted_bytes], 0 ; These will be used in a for loop.
:00401617                                         ; This one represents the total number of encrypted bytes so far
:00401617                                         ; -
:0040161E                 mov     [ebp+i], 0      ; And this one keeps track of what iteration we are on in the for loop, and will
:0040161E                                         ; eventually stop once we reach some value.
```

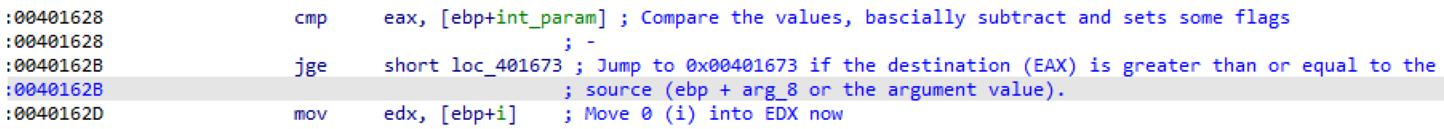
The For Loop

Now we can finally start looking at the for loop. First, it moves i (which IDA calls var_10, but I renamed it to “i”) into EAX (as seen at 0x00401625).



```
:00401625                 mov     eax, [ebp+i]    ; Move the base pointer plus the offset of var_10, which is 0, into EAX.
:00401625                                         ; This will be updated on each iteration of the loop and when it is a certain
:00401625                                         ; value it will break out of the loop.
```

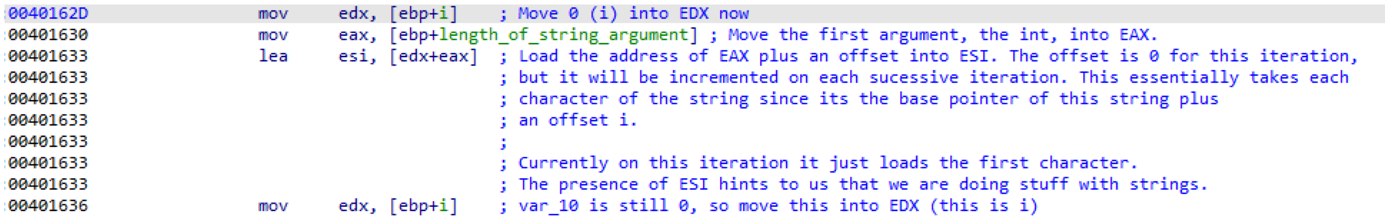
It determines when to break out of the loop using a `jge`, which jumps if i is greater than or equal to the value of EBP+arg_8, or the int argument. This now shows us that the third argument of this function controls the amount of times this loop runs, a hint into what it could be.



```
:00401628               cmp     eax, [ebp+int_param] ; Compare the values, bascially subtract and sets some flags
:00401628                                            ; -
:0040162B               jge     short loc_401673 ; Jump to 0x00401673 if the destination (EAX) is greater than or equal to the
:0040162B                                        ; source (ebp + arg_8 or the argument value).
:0040162D               mov     edx, [ebp+i]     ; Move 0 (i) into EDX now
```

Note: The following annotations only refer to the first loop of the iteration, meaning the value of var_10 (or renamed as “i”) is set to 0 here. The annotations only pertain to this first loop, obviously the numbers would be different on succeeding iterations.

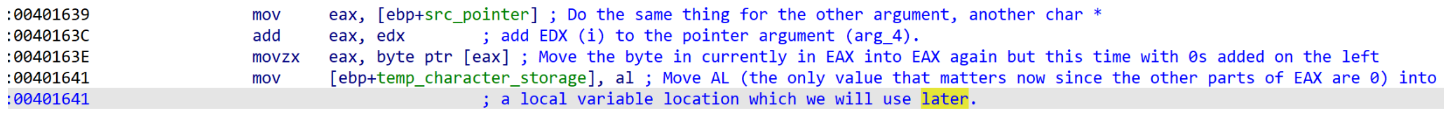
The next thing we do is get the byte (or the character) at the index of the string of what iteration we are on. For example, if we are on the 4th iteration (and i is 4) get the fourth character of the string. Currently we are just on 0 so get the first byte.



```
0040162D            mov     edx, [ebp+i]    ; Move 0 (i) into EDX now
00401630            mov     eax, [ebp+length_of_string_argument] ; Move the first argument, the int, into EAX.
00401633            lea     esi, [edx+eax]  ; Load the address of EAX plus an offset into ESI. The offset is 0 for this iteration,
00401633                                    ; but it will be incremented on each sucessive iteration. This essentially takes each
00401633                                    ; character of the string since its the base pointer of this string plus
00401633                                    ; an offset i.
00401633                                    ;
00401633                                    ; Currently on this iteration it just loads the first character.
00401633                                    ; The presence of ESI hints to us that we are doing stuff with strings.
00401636            mov     edx, [ebp+i]    ; var_10 is still 0, so move this into EDX (this is i)
```

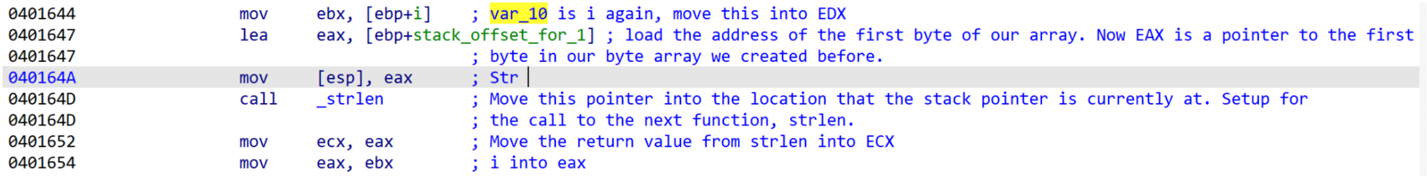
Now what the code does is ensure that there is only one byte of our string input (one character) in AL so we could manipulate it later. The code actually moves AL into a temporary location so it could be used later and not take up EAX (this is var_19, you can see it in the following screenshot at location 0x00401641). Var_19 (renamed to temp_character_storage) is used for the XOR later.

`movzx` is what sets the other parts of EAX outside of AL to zero. This ensures that only one byte is in the lower portion of EAX and nothing else but 0 are in other parts. For example, it ensures that EAX is 000000XX where XX is some byte value.



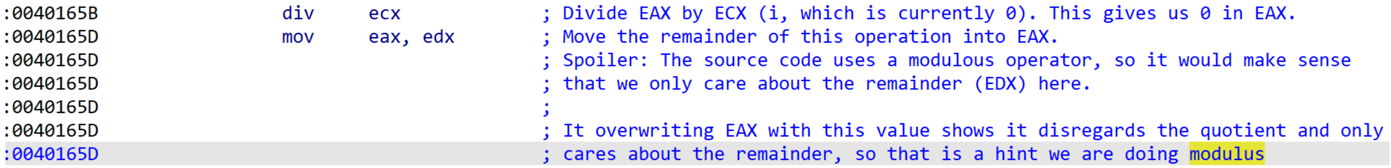
```
:00401639           mov     eax, [ebp+src_pointer] ; Do the same thing for the other argument, another char *
:0040163C           add     eax, edx        ; add EDX (i) to the pointer argument (arg_4).
:0040163E           movzx   eax, byte ptr [eax] ; Move the byte in currently in EAX into EAX again but this time with 0s added on the left
:00401641           mov     [ebp+temp_character_storage], al ; Move AL (the only value that matters now since the other parts of EAX are 0) into
:00401641                                   ; a local variable location which we will use later.
```

Now we call strlen on our “secret” array. This returns the length of our secret array, which is used in our encoding process. A pointer address to the beginning of this array is loaded into EAX, then passed to the strlen function. It also moves the return value of this in ECX temporarily since we need EAX later.



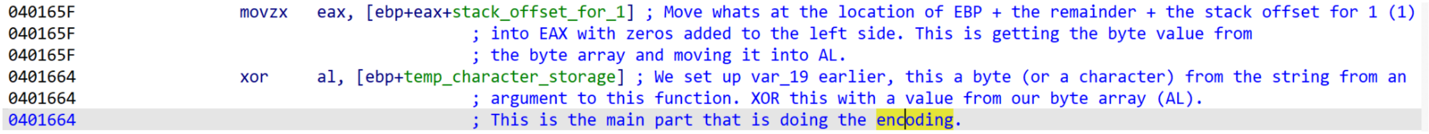
```
0401644          mov     ebx, [ebp+i]    ; var_10 is i again, move this into EDX
0401647          lea     eax, [ebp+stack_offset_for_1] ; load the address of the first byte of our array. Now EAX is a pointer to the first
0401647                                  ; byte in our byte array we created before.
040164A          mov     [esp], eax      ; Str
040164D          call    _strlen         ; Move this pointer into the location that the stack pointer is currently at. Setup for
040164D                                  ; the call to the next function, strlen.
0401652          mov     ecx, eax        ; Move the return value from strlen into ECX
0401654          mov     eax, ebx        ; i into eax
```

Finally we do the encoding "math" on each character. The code does a modulus on the current iteration and the length of secret. For example, if the iteration was 2 and the length of secret is 4 (which it is in our source code) the result of the mod would be 2 % 4 = 2. We know it's the modulus since the code makes special use of EDX which is set to the return value after a `div` instruction. The code actually completely disregards EAX (the normal return of `div`) so it really only cares about the remainder.



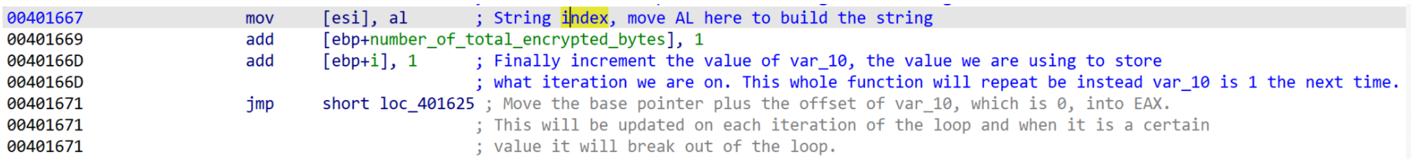
```
:0040165B        div     ecx             ; Divide EAX by ECX (i, which is currently 0). This gives us 0 in EAX.
:0040165D        mov     eax, edx        ; Move the remainder of this operation into EAX.
:0040165D                                ; Spoiler: The source code uses a modulous operator, so it would make sense
:0040165D                                ; that we only care about the remainder (EDX) here.
:0040165D                                ;
:0040165D                                ; It overwriting EAX with this value shows it disregards the quotient and only
:0040165D                                ; cares about the remainder, so that is a hint we are doing modulus
```

Finally, we XOR the character from the input string with a value from our secret array. First, the code sets up the values for the XOR by moving them into EAX using a `movzx` instruction which makes all the values except AL 0. Then it does the XOR using temp_character_storage (originally var_19) which is the temporary value the code set up earlier.



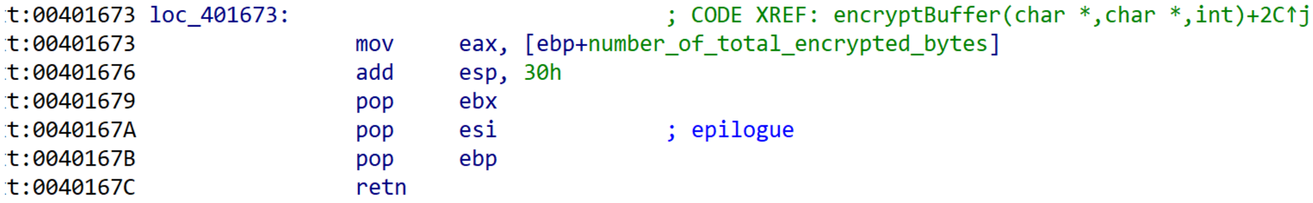
```
040165F          movzx   eax, [ebp+eax+stack_offset_for_1] ; Move whats at the location of EBP + the remainder + the stack offset for 1 (1)
040165F                                  ; into EAX with zeros added to the left side. This is getting the byte value from
040165F                                  ; the byte array and moving it into AL.
0401664          xor     al, [ebp+temp_character_storage] ; We set up var_19 earlier, this a byte (or a character) from the string from an
0401664                                  ; argument to this function. XOR this with a value from our byte array (AL).
0401664                                  ; This is the main part that is doing the encoding.
```

Finally we increment i by one and do the loop over again.



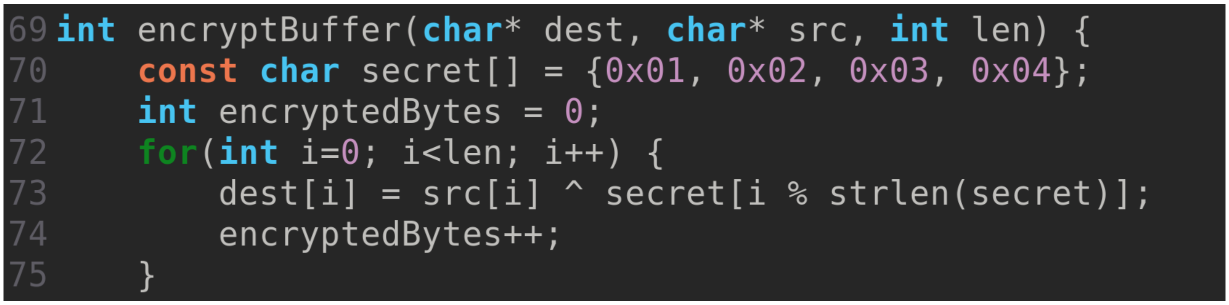
```
00401667         mov     [esi], al       ; String index, move AL here to build the string
00401669         add     [ebp+number_of_total_encrypted_bytes], 1
0040166D         add     [ebp+i], 1      ; Finally increment the value of var_10, the value we are using to store
0040166D                                 ; what iteration we are on. This whole function will repeat be instead var_10 is 1 the next time.
00401671         jmp     short loc_401625 ; Move the base pointer plus the offset of var_10, which is 0, into EAX.
00401671                                 ; This will be updated on each iteration of the loop and when it is a certain
00401671                                 ; value it will break out of the loop.
```

When the loop is over, it moves the total number of bytes we encrypted into EAX (the return value), moves the stack pointer up by 30 again, and returns the function. The return value according to the source code is this value, the number of bytes we encrypted.



```
t:00401673 loc_401673:                             ; CODE XREF: encryptBuffer(char *,char *,int)+2C↑j
t:00401673                 mov     eax, [ebp+number_of_total_encrypted_bytes]
t:00401676                 add     esp, 30h
t:00401679                 pop     ebx
t:0040167A                 pop     esi             ; epilogue
t:0040167B                 pop     ebp
t:0040167C                 retn
```

For reference, this is the original source code of the encryption function:



```
int encryptBuffer(char* dest, char* src, int len) {
    const char secret[] = {0x01, 0x02, 0x03, 0x04};
    int encryptedBytes = 0;
    for(int i=0; i<len; i++) {
        dest[i] = src[i] ^ secret[i % strlen(secret)];
        encryptedBytes++;
    }
```

How the Program Interprets Commands in IDA

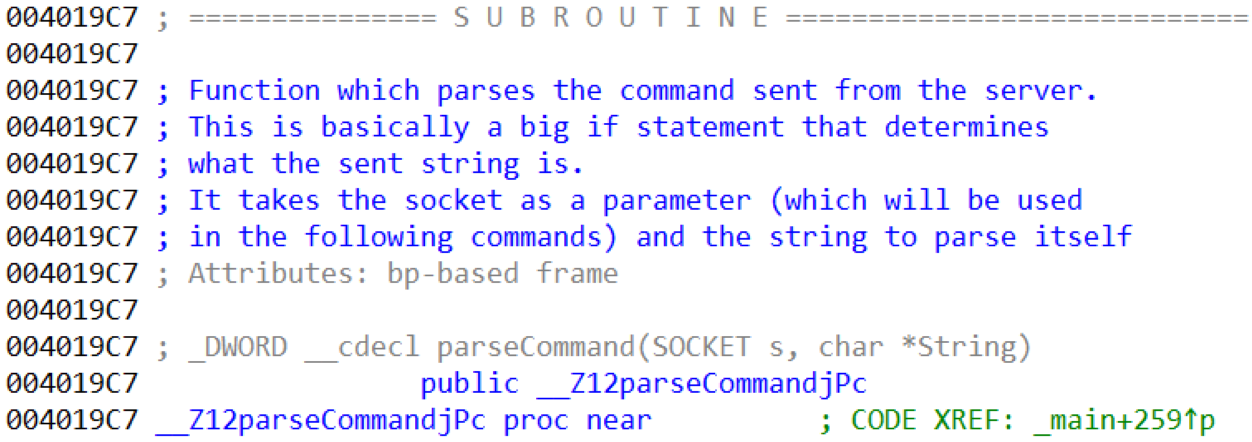
Task 2.2:
Explain, using IDA, how the program interprets commands from the server and how it processes the output

- Use screenshots, here, to help illustrate exactly what is going on.
- Annotations within your IDB may also be helpful.

The server sends commands to the client to get it to do various tasks. The function that has to do specifically with interpreting these commands is the `parseCommand()` function in the client. This is our code which parses the messages sent from the server. It's located at 0x004019c7 in our binary.

Arguments

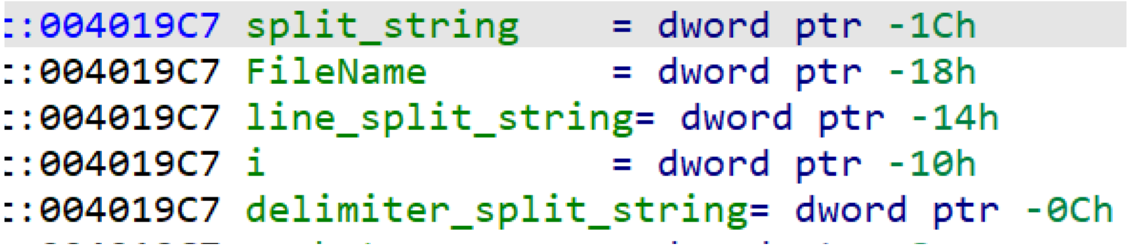
When you first look at this command in IDA, it identifies two arguments. These are a SOCKET object and some char * string. A standard call to this function in pseudocode would look like `parseCommand(SOCKET sock, char* input)`.



```
004019C7 ; =============== S U B R O U T I N E =======================================
004019C7
004019C7 ; Function which parses the command sent from the server.
004019C7 ; This is basically a big if statement that determines
004019C7 ; what the sent string is.
004019C7 ; It takes the socket as a parameter (which will be used
004019C7 ; in the following commands) and the string to parse itself
004019C7 ; Attributes: bp-based frame
004019C7
004019C7 ; _DWORD __cdecl parseCommand(SOCKET s, char *String)
004019C7                 public __Z12parseCommandjPc
004019C7 __Z12parseCommandjPc proc near          ; CODE XREF: _main+259↑p
```

Local Variables

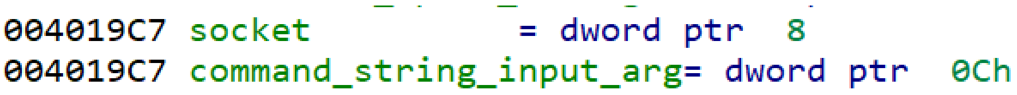
This function has several local variables, which are shown below. IDA named some of these for us, such as `FileName`. The rest will be explained as we get to them.



```
::004019C7 split_string     = dword ptr -1Ch
::004019C7 FileName         = dword ptr -18h
::004019C7 line_split_string= dword ptr -14h
::004019C7 i                = dword ptr -10h
::004019C7 delimiter_split_string= dword ptr -0Ch
```

Arguments

This function takes in two arguments, which are values above the base pointer. The arguments for this function are the socket and the input string to parse. These are shown in IDA renamed as `socket` and `command_string_input_arg`.

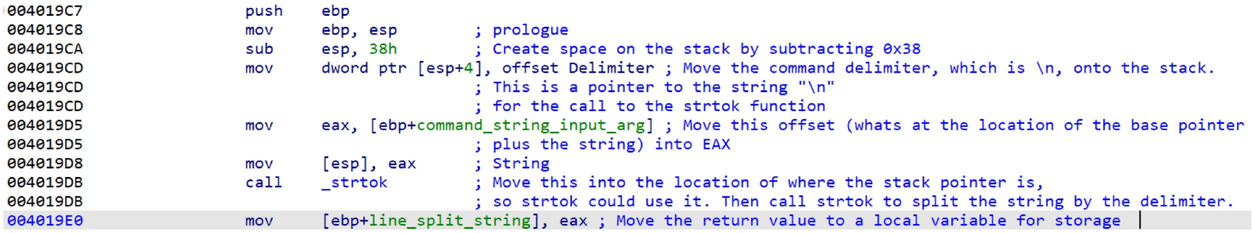


```
004019C7 socket               = dword ptr  8
004019C7 command_string_input_arg= dword ptr  0Ch
```

Split Loop

The first thing we need to do is split up the input string to parse it. There are several calls to strtok here, which is what we are using to split up the string. The first split we do is on “\n”, because that would be the end of the command after pressing enter.

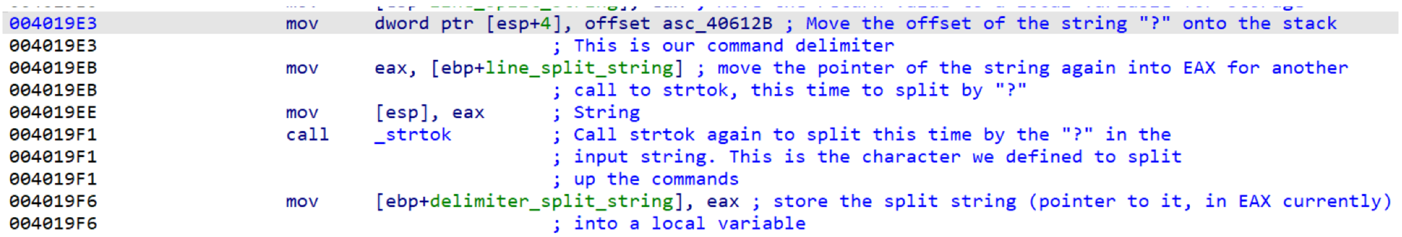
This string is split by strtok after pushing the values to this function on the stack then calling it, and the pointer to this now split string is stored in a local variable renamed to `line_split_string`.



```
004019C7                push    ebp
004019C8                mov     ebp, esp        ; prologue
004019CA                sub     esp, 38h        ; Create space on the stack by subtracting 0x38
004019CD                mov     dword ptr [esp+4], offset Delimiter ; Move the command delimiter, which is \n, onto the stack.
004019CD                                        ; This is a pointer to the string "\n"
004019CD                                        ; for the call to the strtok function
004019D5                mov     eax, [ebp+command_string_input_arg] ; Move this offset (whats at the location of the base pointer
004019D5                                        ; plus the string) into EAX
004019D8                mov     [esp], eax      ; String
004019DB                call    _strtok         ; Move this into the location of where the stack pointer is,
004019DB                                        ; so strtok could use it. Then call strtok to split the string by the delimiter.
004019E0                mov     [ebp+line_split_string], eax ; Move the return value to a local variable for storage
```

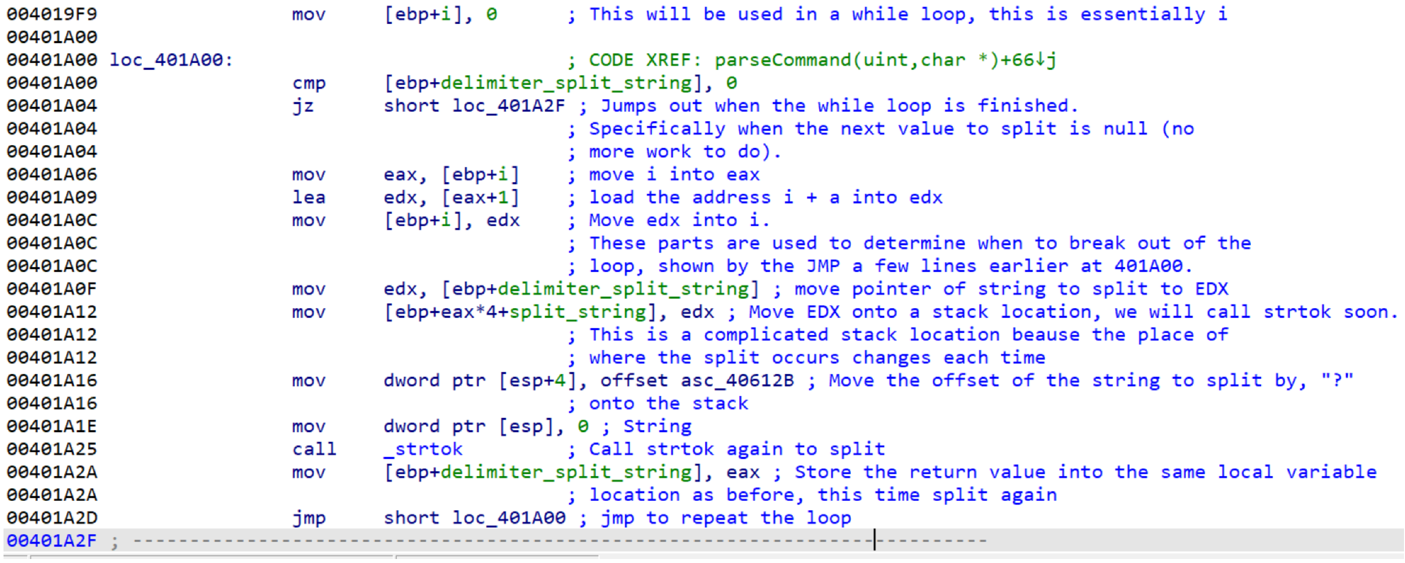
The next thing we do is split by “?”, which is what is the delimiter between string arguments (for example, in our program the syntax for the upload command is:

`upload?<filepath>`). This is another call to strtok, this time with the string from before but this time splitting on “?”.



```
004019E3                mov     dword ptr [esp+4], offset asc_40612B ; Move the offset of the string "?" onto the stack
004019E3                                        ; This is our command delimiter
004019EB                mov     eax, [ebp+line_split_string] ; move the pointer of the string again into EAX for another
004019EB                                        ; call to strtok, this time to split by "?"
004019EE                mov     [esp], eax      ; String
004019F1                call    _strtok         ; Call strtok again to split this time by the "?" in the
004019F1                                        ; input string. This is the character we defined to split
004019F1                                        ; up the commands
004019F6                mov     [ebp+delimiter_split_string], eax ; store the split string (pointer to it, in EAX currently)
004019F6                                        ; into a local variable
```

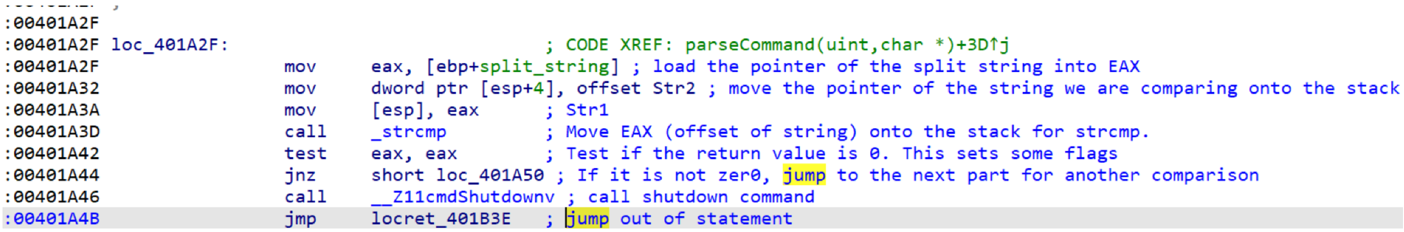
The next part of this function calls strtok in a loop. We need to call this continually on the string because there could be multiple command delimiters in one command. For example, the download command uses multiple delimiters `download?<filename>?<url>` and strtok will only split the string one delimiter at a time, so continually call this in a while loop until all delimiters are processed. The following screenshot shows this while loop, and the variable i is used to keep track of when to break out of the loop.



```
004019F9                mov     [ebp+i], 0      ; This will be used in a while loop, this is essentially i
00401A00
00401A00 loc_401A00:                            ; CODE XREF: parseCommand(uint,char *)+66↓j
00401A00                cmp     [ebp+delimiter_split_string], 0
00401A04                jz      short loc_401A2F ; Jumps out when the while loop is finished.
00401A04                                        ; Specifically when the next value to split is null (no
00401A04                                        ; more work to do).
00401A06                mov     eax, [ebp+i]    ; move i into eax
00401A09                lea     edx, [eax+1]    ; load the address i + a into edx
00401A0C                mov     [ebp+i], edx    ; Move edx into i.
00401A0C                                        ; These parts are used to determine when to break out of the
00401A0C                                        ; loop, shown by the JMP a few lines earlier at 401A00.
00401A0F                mov     edx, [ebp+delimiter_split_string] ; move pointer of string to split to EDX
00401A12                mov     [ebp+eax*4+split_string], edx ; Move EDX onto a stack location, we will call strtok soon.
00401A12                                        ; This is a complicated stack location beause the place of
00401A12                                        ; where the split occurs changes each time
00401A16                mov     dword ptr [esp+4], offset asc_40612B ; Move the offset of the string to split by, "?"
00401A16                                        ; onto the stack
00401A1E                mov     dword ptr [esp], 0 ; String
00401A25                call    _strtok         ; Call strtok again to split
00401A2A                mov     [ebp+delimiter_split_string], eax ; Store the return value into the same local variable
00401A2A                                        ; location as before, this time split again
00401A2D                jmp     short loc_401A00 ; jmp to repeat the loop
00401A2F ; ---------------------------------------------------------------------------
```

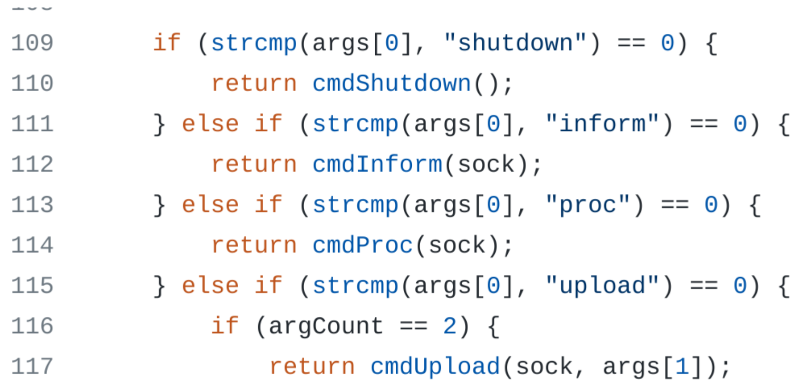
After completing the loop which splits the string up, there is a large if statement that compares the now split string and determines which code path to take based on this string. For example, if the first element of the split string array is “shutdown”, the code will jump to the code for “shutdown”. Going to the location right after this while loop brings us to here, which is where these comparisons are made.

The first comparison uses the strcmp function to compare if the first element of the split array is equal to “shutdown”. If it is, jump to the code for that. The variable `split_string` is the split string that we are comparing to.



```
:00401A2F
:00401A2F loc_401A2F:                           ; CODE XREF: parseCommand(uint,char *)+3D↑j
:00401A2F                 mov     eax, [ebp+split_string] ; load the pointer of the split string into EAX
:00401A32                 mov     dword ptr [esp+4], offset Str2 ; move the pointer of the string we are comparing onto the stack
:00401A3A                 mov     [esp], eax      ; Str1
:00401A3D                 call    _strcmp         ; Move EAX (offset of string) onto the stack for strcmp.
:00401A42                 test    eax, eax        ; Test if the return value is 0. This sets some flags
:00401A44                 jnz     short loc_401A50 ; If it is not zer0, jump to the next part for another comparison
:00401A46                 call    __Z11cmdShutdownv ; call shutdown command
:00401A4B                 jmp     locret_401B3E   ; jump out of statement
```

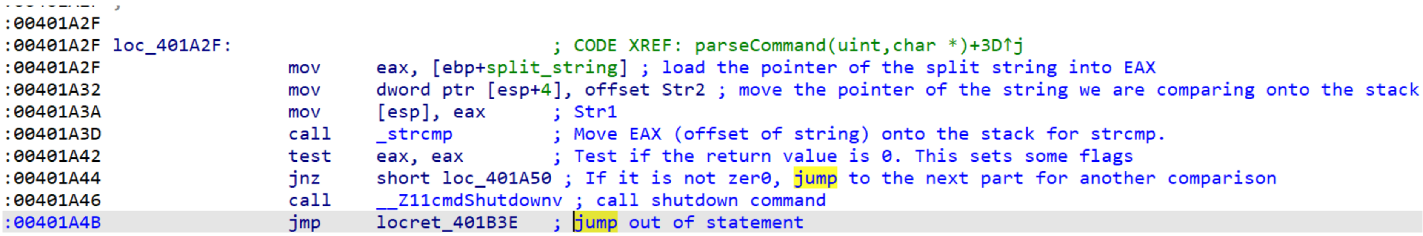
For reference, this is what the if statement looks like in the source code:



```
    if (strcmp(args[0], "shutdown") == 0) {
        return cmdShutdown();
    } else if (strcmp(args[0], "inform") == 0) {
        return cmdInform(sock);
    } else if (strcmp(args[0], "proc") == 0) {
        return cmdProc(sock);
    } else if (strcmp(args[0], "upload") == 0) {
        if (argCount == 2) {
            return cmdUpload(sock, args[1]);
```

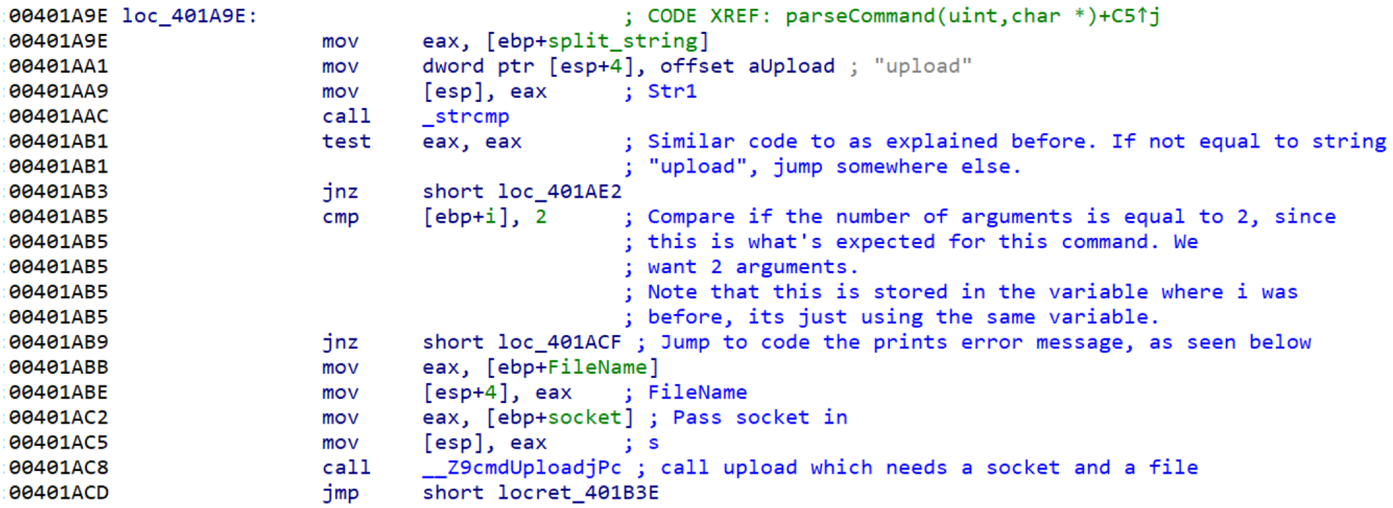
Similar code to the “shutdown” command is carried out for the rest of the following functions. For example, “inform” has pretty much the same assembly but this time also pushes the socket to the stack as well because the inform command needs that.

Similar code for inform:



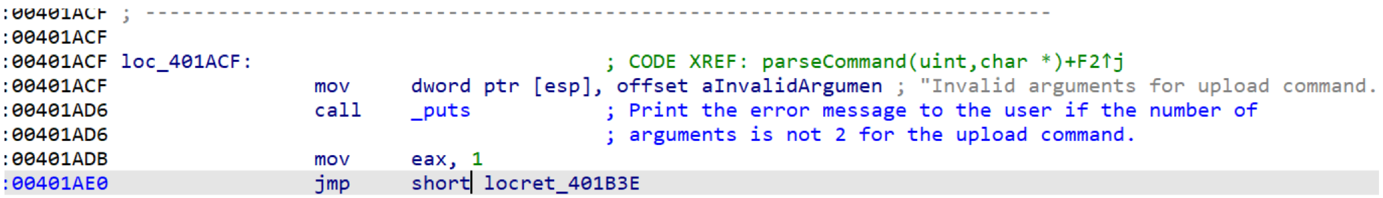
```
:00401A2F
:00401A2F loc_401A2F:                           ; CODE XREF: parseCommand(uint,char *)+3D↑j
:00401A2F                 mov     eax, [ebp+split_string] ; load the pointer of the split string into EAX
:00401A32                 mov     dword ptr [esp+4], offset Str2 ; move the pointer of the string we are comparing onto the stack
:00401A3A                 mov     [esp], eax      ; Str1
:00401A3D                 call    _strcmp         ; Move EAX (offset of string) onto the stack for strcmp.
:00401A42                 test    eax, eax        ; Test if the return value is 0. This sets some flags
:00401A44                 jnz     short loc_401A50 ; If it is not zer0, jump to the next part for another comparison
:00401A46                 call    __Z11cmdShutdownv ; call shutdown command
:00401A4B                 jmp     locret_401B3E   ; jump out of statement
```

The next function, “proc”, has similar code, so I won't bother to repeat it here. “Upload” and “download” are similar but both have an extra if statement determining if the number of arguments are correct. If it is not, it jumps to code which prints an “invalid command” error message.



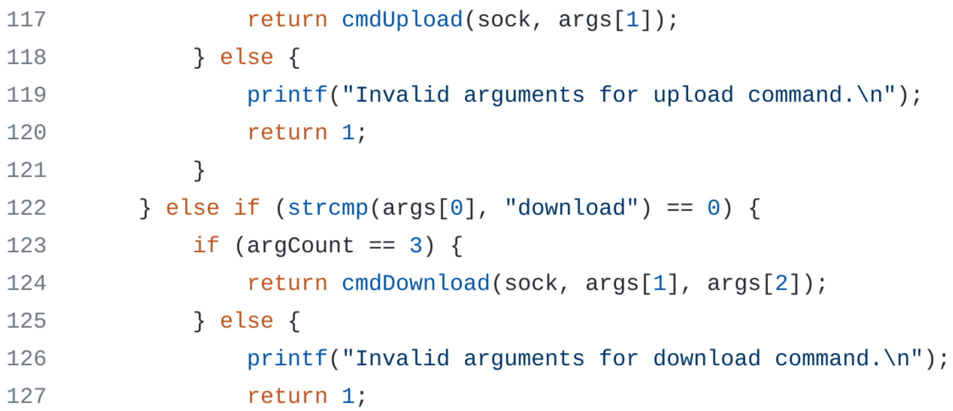
```
00401A9E loc_401A9E:                            ; CODE XREF: parseCommand(uint,char *)+C5↑j
00401A9E                 mov     eax, [ebp+split_string]
00401AA1                 mov     dword ptr [esp+4], offset aUpload ; "upload"
00401AA9                 mov     [esp], eax      ; Str1
00401AAC                 call    _strcmp
00401AB1                 test    eax, eax        ; Similar code to as explained before. If not equal to string
00401AB1                                         ; "upload", jump somewhere else.
00401AB3                 jnz     short loc_401AE2
00401AB5                 cmp     [ebp+i], 2      ; Compare if the number of arguments is equal to 2, since
00401AB5                                         ; this is what's expected for this command. We
00401AB5                                         ; want 2 arguments.
00401AB5                                         ; Note that this is stored in the variable where i was
00401AB5                                         ; before, its just using the same variable.
00401AB9                 jnz     short loc_401ACF ; Jump to code the prints error message, as seen below
00401ABB                 mov     eax, [ebp+FileName]
00401ABE                 mov     [esp+4], eax    ; FileName
00401AC2                 mov     eax, [ebp+socket] ; Pass socket in
00401AC5                 mov     [esp], eax      ; s
00401AC8                 call    __Z9cmdUploadjPc ; call upload which needs a socket and a file
00401ACD                 jmp     short locret_401B3E
```

If the number of arguments are not 2, it jumps to this code which prints the error message.



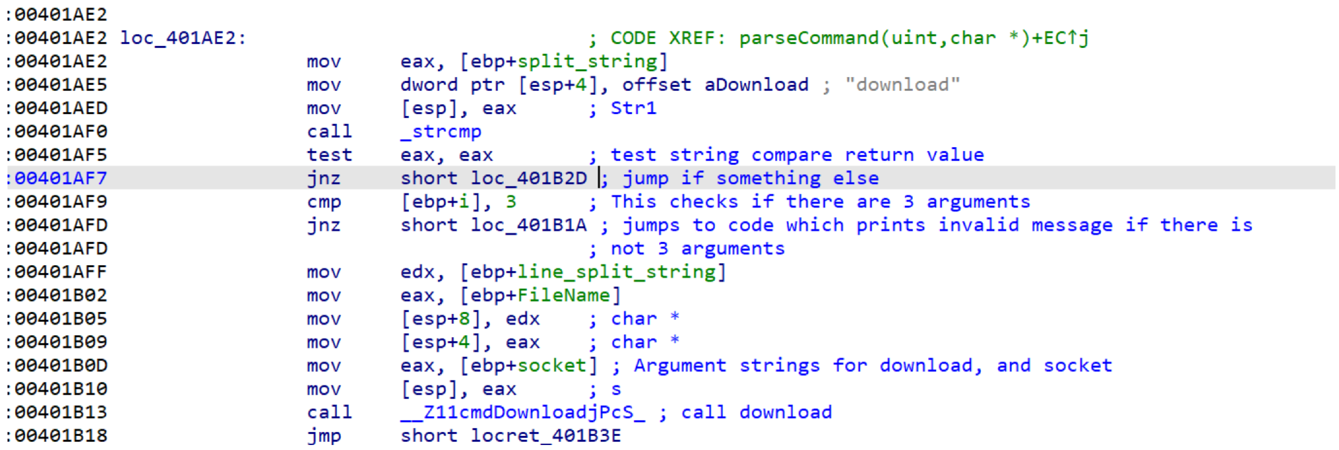
```
:00401ACF ; ---------------------------------------------------------------------------
:00401ACF
:00401ACF loc_401ACF:                           ; CODE XREF: parseCommand(uint,char *)+F2↑j
:00401ACF                 mov     dword ptr [esp], offset aInvalidArgumen ; "Invalid arguments for upload command.
:00401AD6                 call    _puts           ; Print the error message to the user if the number of
:00401AD6                                         ; arguments is not 2 for the upload command.
:00401ADB                 mov     eax, 1
:00401AE0                 jmp     short locret_401B3E
```

For reference, the source code of this if statement for both upload and download looks like this:



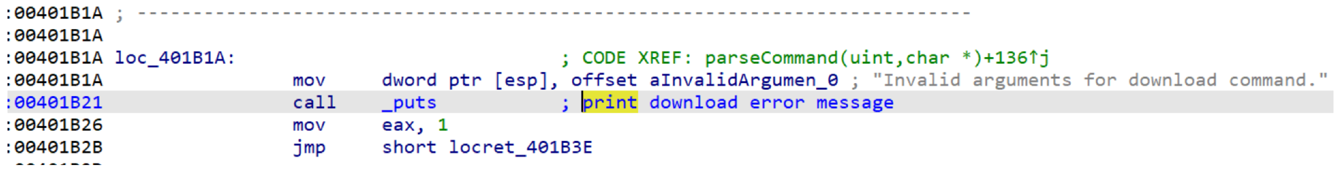
```
            return cmdUpload(sock, args[1]);
        } else {
            printf("Invalid arguments for upload command.\n");
            return 1;
        }
    } else if (strcmp(args[0], "download") == 0) {
        if (argCount == 3) {
            return cmdDownload(sock, args[1], args[2]);
        } else {
            printf("Invalid arguments for download command.\n");
            return 1;
```

Download is similar but this time it checks if there are 3 arguments. The assembly code is also similar to upload.



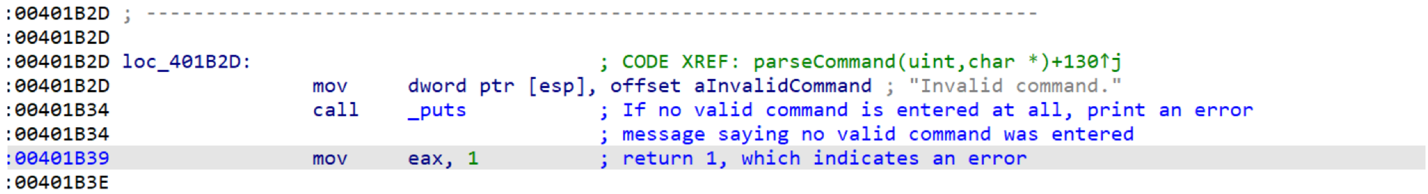
```
:00401AE2
:00401AE2 loc_401AE2:                             ; CODE XREF: parseCommand(uint,char *)+EC↑j
:00401AE2                 mov     eax, [ebp+split_string]
:00401AE5                 mov     dword ptr [esp+4], offset aDownload ; "download"
:00401AED                 mov     [esp], eax      ; Str1
:00401AF0                 call    _strcmp
:00401AF5                 test    eax, eax        ; test string compare return value
:00401AF7                 jnz     short loc_401B2D ; jump if something else
:00401AF9                 cmp     [ebp+i], 3      ; This checks if there are 3 arguments
:00401AFD                 jnz     short loc_401B1A ; jumps to code which prints invalid message if there is
:00401AFD                                         ; not 3 arguments
:00401AFF                 mov     edx, [ebp+line_split_string]
:00401B02                 mov     eax, [ebp+FileName]
:00401B05                 mov     [esp+8], edx    ; char *
:00401B09                 mov     [esp+4], eax    ; char *
:00401B0D                 mov     eax, [ebp+socket] ; Argument strings for download, and socket
:00401B10                 mov     [esp], eax      ; s
:00401B13                 call    __Z11cmdDownloadjPcS_ ; call download
:00401B18                 jmp     short locret_401B3E
```

This is the code which prints the error message for download:



```
:00401B1A ; ---------------------------------------------------------------------------
:00401B1A
:00401B1A loc_401B1A:                             ; CODE XREF: parseCommand(uint,char *)+136↑j
:00401B1A                 mov     dword ptr [esp], offset aInvalidArgumen_0 ; "Invalid arguments for download command."
:00401B21                 call    _puts           ; print download error message
:00401B26                 mov     eax, 1
:00401B2B                 jmp     short locret_401B3E
```

In addition to these error messages, there is also an additional "catch all" error message if no invalid command is entered. This code is located at 0x00401b2d and is jumped to specifically if no command is detected after the "download" command check.



```
:00401B2D ; ---------------------------------------------------------------------------
:00401B2D
:00401B2D loc_401B2D:                             ; CODE XREF: parseCommand(uint,char *)+130↑j
:00401B2D                 mov     dword ptr [esp], offset aInvalidCommand ; "Invalid command."
:00401B34                 call    _puts           ; If no valid command is entered at all, print an error
:00401B34                                         ; message saying no valid command was entered
:00401B39                 mov     eax, 1          ; return 1, which indicates an error
:00401B3E
```

At this point, all of the code for parsing the string input is completed, and the code has moved on to executing the specific function for the entered command. This function will be called again when it needs to parse another input string.

How Compiler Settings Affect Output

Task 2.3

Explain how tinkering with compiler settings can change the disassembly of your program.

- Provide examples of settings you used and screenshots.

We compiled the program using g++.exe version 6.3.0. We used the following commands for both the server and the client.

Client:

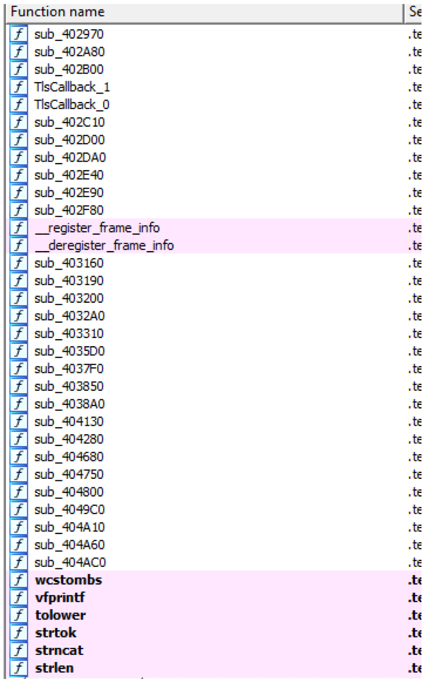
```
g++ client\client.cpp -o client -lws2_32 -lwininet -lpsapi
-liphlpapi
```

Server:

```
g++ server\server.cpp -o server -lws2_32
```

There are several extra flags you can add to the compiler to alter the resulting assembly. The most substantial one related to reverse engineering is disabling the function names (so you won't see them in the strings output). You can do this by adding the `-s` flag.

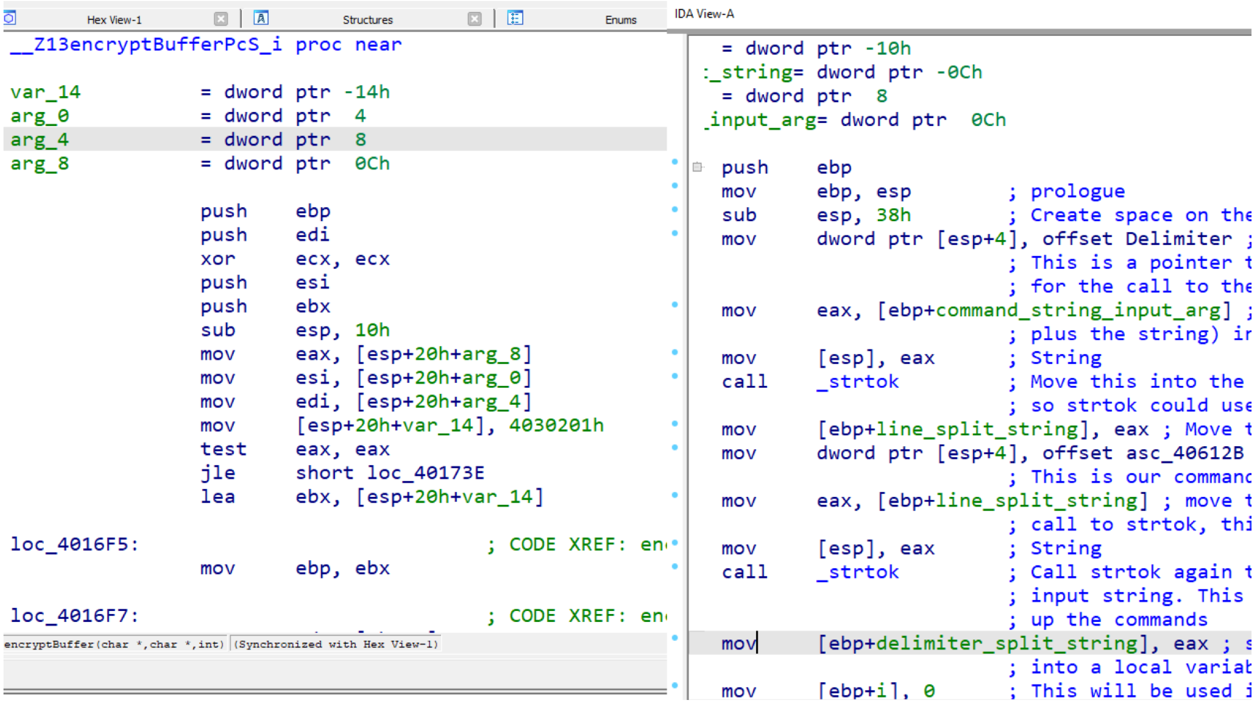
As you can see just by looking at the IDA function listing, there are no names for any of the functions we wrote (besides basic c++ functions and the dll functions).



Function name	Se
sub_402970	.te
sub_402A80	.te
sub_402B00	.te
TlsCallback_1	.te
TlsCallback_0	.te
sub_402C10	.te
sub_402D00	.te
sub_402DA0	.te
sub_402E40	.te
sub_402E90	.te
sub_402F80	.te
__register_frame_info	.te
__deregister_frame_info	.te
sub_403160	.te
sub_403190	.te
sub_403200	.te
sub_4032A0	.te
sub_403310	.te
sub_4035D0	.te
sub_4037F0	.te
sub_403850	.te
sub_4038A0	.te
sub_404130	.te
sub_404280	.te
sub_404680	.te
sub_404750	.te
sub_404800	.te
sub_4049C0	.te
sub_404A10	.te
sub_404A60	.te
sub_404AC0	.te
wcstombs	**.te**
vfprintf	**.te**
tolower	**.te**
strtok	**.te**
strncat	**.te**
strlen	**.te**

This would make it harder for us to reverse engineer the program since the functions aren't already named for us, so it would take a lot longer to find out what each one does. If you want to prevent people from reversing your program you should use this flag.

Another flag we tried that changed the output is the `-o3` flag. This flag has to do with the level of optimization that the compiler puts on the code. It has 4 options, -o0, -o1, -o2 and -o3 with -o0 being the default (no optimization) and what our code was originally compiled in. Using the -o3 option resulted in significant differences between the code. For example, here you can see the encryptBuffer() function with both high optimization (left) and no optimization (original, right):



```
__Z13encryptBufferPcS_i proc near

var_14          = dword ptr -14h
arg_0           = dword ptr  4
arg_4           = dword ptr  8
arg_8           = dword ptr  0Ch

                push    ebp
                push    edi
                xor     ecx, ecx
                push    esi
                push    ebx
                sub     esp, 10h
                mov     eax, [esp+20h+arg_8]
                mov     esi, [esp+20h+arg_0]
                mov     edi, [esp+20h+arg_4]
                mov     [esp+20h+var_14], 4030201h
                test    eax, eax
                jle     short loc_40173E
                lea     ebx, [esp+20h+var_14]

loc_4016F5:                             ; CODE XREF: en
                mov     ebp, ebx

loc_4016F7:                             ; CODE XREF: en
```

encryptBuffer(char *,char *,int) (Synchronized with Hex View-1)

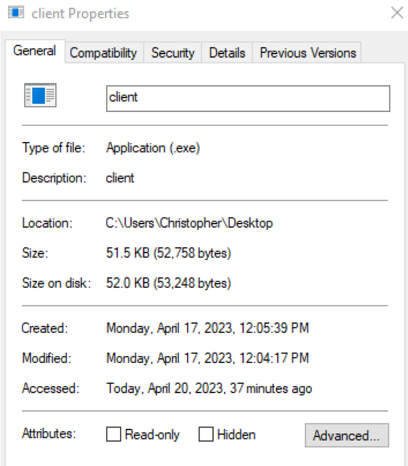
```
  = dword ptr -10h
:_string= dword ptr -0Ch
  = dword ptr  8
_input_arg= dword ptr  0Ch

  push    ebp
  mov     ebp, esp          ; prologue
  sub     esp, 38h          ; Create space on the
  mov     dword ptr [esp+4], offset Delimiter ;
                            ; This is a pointer t
                            ; for the call to the
  mov     eax, [ebp+command_string_input_arg] ;
                            ; plus the string) ir
  mov     [esp], eax        ; String
  call    _strtok           ; Move this into the
                            ; so strtok could use
  mov     [ebp+line_split_string], eax ; Move t
  mov     dword ptr [esp+4], offset asc_40612B
                            ; This is our commanc
  mov     eax, [ebp+line_split_string] ; move t
                            ; call to strtok, thi
  mov     [esp], eax        ; String
  call    _strtok           ; Call strtok again t
                            ; input string. This
                            ; up the commands
  mov     [ebp+delimiter_split_string], eax ; s
                            ; into a local variat
  mov     [ebp+i], 0        ; This will be used i
```

The code on the left is a lot different and more complicated. It seems to use more optimized assembly in order to make a faster program. For example, it uses XOR ecx, ecx in order to make ECX 0 (faster than mov ECX, 0). It also seems to use the `push` instruction more than just manually moving things on the stack via `mov`.

This optimization setting results in a huge difference since it attempts to make the program faster by using more optimized assembly. This setting could make it harder for us to reverse engineer the program because the resulting code is more complicated.

The last flag we tried was the -g flag. This is used to enable debugging flags for use with the GDB GNU Debugger. It will add the symbol table to the output. One of the first things I noticed was that this option increases the file size by about 10kb.



client Properties

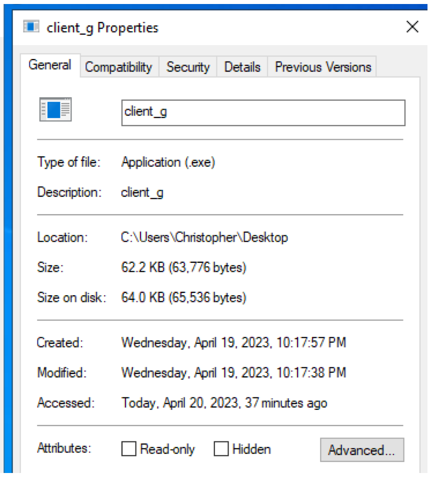
client

Type of file: Application (.exe)
Description: client

Location: C:\Users\Christopher\Desktop
Size: 51.5 KB (52,758 bytes)
Size on disk: 52.0 KB (53,248 bytes)

Created: Monday, April 17, 2023, 12:05:39 PM
Modified: Monday, April 17, 2023, 12:04:17 PM
Accessed: Today, April 20, 2023, 37 minutes ago

Attributes: □ Read-only □ Hidden



client_g Properties

client_g

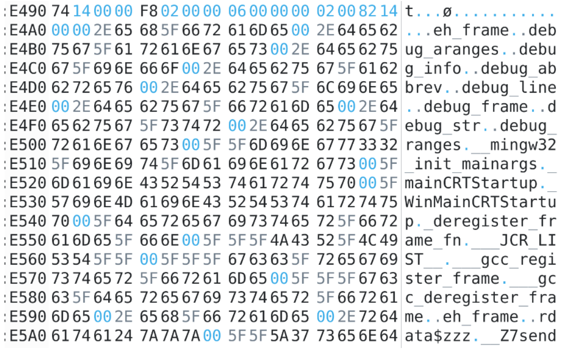
Type of file: Application (.exe)
Description: client_g

Location: C:\Users\Christopher\Desktop
Size: 62.2 KB (63,776 bytes)
Size on disk: 64.0 KB (65,536 bytes)

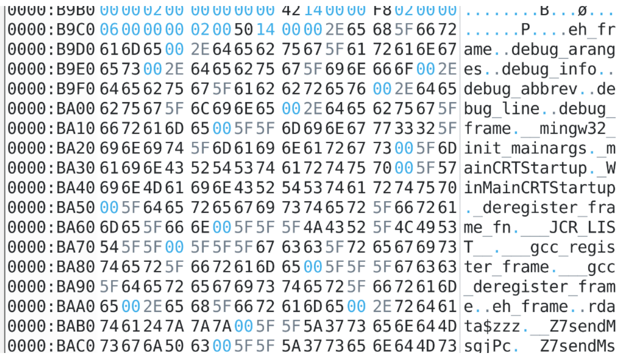
Created: Wednesday, April 19, 2023, 10:17:57 PM
Modified: Wednesday, April 19, 2023, 10:17:38 PM
Accessed: Today, April 20, 2023, 37 minutes ago

Attributes: □ Read-only □ Hidden

The main difference in the files itself is that there are some additional entries added in the symbol table at the end of the file. If you are looking at both files in a hex editor, you can simply notice that there are some more options in the file with the debug symbols added. As you can see in the following screenshot, the one without debug symbols (left) is different from the one with debug symbols (right).



```
:E490 74 14 00 00  F8 02 00 00  06 00 00 00  02 00 82 14 t...ø...........
:E4A0 00 00 2E 65  68 5F 66 72  61 6D 65 00  2E 64 65 62 ...eh_frame..deb
:E4B0 75 67 5F 61  72 61 6E 67  65 73 00 2E  64 65 62 75 ug_aranges..debu
:E4C0 67 5F 69 6E  66 6F 00 2E  64 65 62 75  67 5F 61 62 g_info..debug_ab
:E4D0 62 72 65 76  00 2E 64 65  62 75 67 5F  6C 69 6E 65 brev..debug_line
:E4E0 00 2E 64 65  62 75 67 5F  66 72 61 6D  65 00 2E 64 ..debug_frame..d
:E4F0 65 62 75 67  5F 73 74 72  00 2E 64 65  62 75 67 5F ebug_str..debug_
:E500 72 61 6E 67  65 73 00 5F  5F 6D 69 6E  67 77 33 32 ranges.__mingw32
:E510 5F 69 6E 69  74 5F 6D 61  69 6E 61 72  67 73 00 5F _init_mainargs._
:E520 6D 61 69 6E  43 52 54 53  74 61 72 74  75 70 00 5F mainCRTStartup._
:E530 57 69 6E 4D  61 69 6E 43  52 54 53 74  61 72 74 75 WinMainCRTStartu
:E540 70 00 5F 64  65 72 65 67  69 73 74 65  72 5F 66 72 p._deregister_fr
:E550 61 6D 65 5F  66 6E 00 5F  5F 5F 4A 43  52 5F 4C 49 ame_fn.___JCR_LI
:E560 53 54 5F 5F  00 5F 5F 5F  67 63 63 5F  72 65 67 69 ST__.___gcc_regi
:E570 73 74 65 72  5F 66 72 61  6D 65 00 5F  5F 5F 67 63 ster_frame.___gc
:E580 63 5F 64 65  72 65 67 69  73 74 65 72  5F 66 72 61 c_deregister_fra
:E590 6D 65 00 2E  65 68 5F 66  72 61 6D 65  00 2E 72 64 me..eh_frame..rd
:E5A0 61 74 61 24  7A 7A 7A 00  5F 5F 5A 37  73 65 6E 64 ata$zzz.__Z7send
```



```
0000:B9B0 00 00 02 00  00 00 00 00  42 14 00 00  F8 02 00 00 ........B...ø...
0000:B9C0 06 00 00 00  02 00 50 14  00 00 2E 65  68 5F 66 72 ......P....eh_fr
0000:B9D0 61 6D 65 00  2E 64 65 62  75 67 5F 61  72 61 6E 67 ame..debug_arang
0000:B9E0 65 73 00 2E  64 65 62 75  67 5F 69 6E  66 6F 00 2E es..debug_info..
0000:B9F0 64 65 62 75  67 5F 61 62  62 72 65 76  00 2E 64 65 debug_abbrev..de
0000:BA00 62 75 67 5F  6C 69 6E 65  00 2E 64 65  62 75 67 5F bug_line..debug_
0000:BA10 66 72 61 6D  65 00 5F 5F  6D 69 6E 67  77 33 32 5F frame.__mingw32_
0000:BA20 69 6E 69 74  5F 6D 61 69  6E 61 72 67  73 00 5F 6D init_mainargs._m
0000:BA30 61 69 6E 43  52 54 53 74  61 72 74 75  70 00 5F 57 ainCRTStartup._W
0000:BA40 69 6E 4D 61  69 6E 43 52  54 53 74 61  72 74 75 70 inMainCRTStartup
0000:BA50 00 5F 64 65  72 65 67 69  73 74 65 72  5F 66 72 61 ._deregister_fra
0000:BA60 6D 65 5F 66  6E 00 5F 5F  5F 4A 43 52  5F 4C 49 53 me_fn.___JCR_LIS
0000:BA70 54 5F 5F 00  5F 5F 5F 67  63 63 5F 72  65 67 69 73 T__.___gcc_regis
0000:BA80 74 65 72 5F  66 72 61 6D  65 00 5F 5F  5F 67 63 63 ter_frame.___gcc
0000:BA90 5F 64 65 72  65 67 69 73  74 65 72 5F  66 72 61 6D _deregister_fram
0000:BAA0 65 00 2E 65  68 5F 66 72  61 6D 65 00  2E 72 64 61 e..eh_frame..rda
0000:BAB0 74 61 24 7A  7A 7A 00 5F  5F 5A 37 73  65 6E 64 4D ta$zzz.__Z7sendM
0000:BAC0 73 67 6A 50  63 00 5F 5F  5A 37 73 65  6E 64 4D 73 sgjPc.__Z7sendMs
```