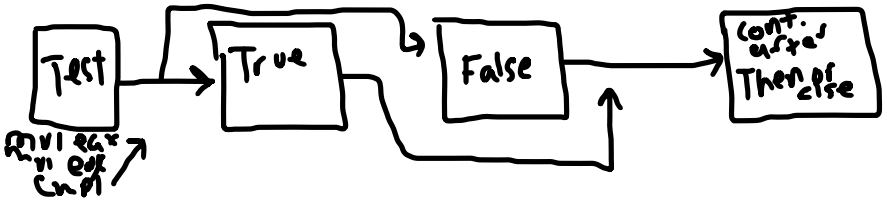
Shane Irons

Homework #2 Control Flow Analysis, Structure Identification and C++ Reversing

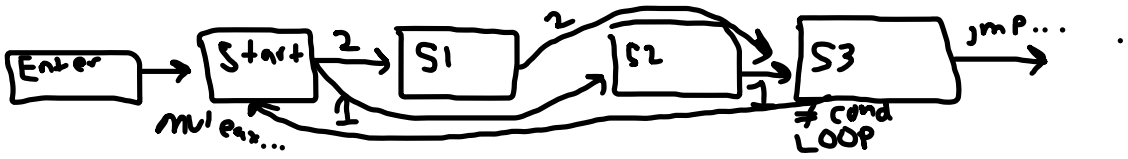
CIS4138/CAP5137 Software Reverse Engineering and Malware Analysis Fall 2022

Question 1 (10 points) Summarize in your own words how to recognize 1) if-then-else code segment, and 2) a loop code segment in a binary program. In each case, give a diagram to illustrate the involved basic blocks, and conditional and unconditional branches.

Recognizing an if-then-else statement in a binary program requires finding two registers with values stored there being compared and having a jump follow. In many cases, eax and edx (or some other register) will get values for the if statement, they will be compared, a type of jump command will follow, and under this jump command will be the subsequent commands if the jump condition is not met. The conditional branches are typically pointing to the "true" and "false" paths. Unconditional branches are instructions jumping to a new sequence that does not take any condition from the if statement, this may be right outside of the construct.



Loop code in a binary program can be seen when JMP conditions must be met before moving on. Various instructions can manipulate the value from the register, thus forming the loop where the instruction set will continue to compare the registers until the conditions match and the JMP can commence. The conditional branches maintain the loop and keep values inside incrementing or decrementing to attain the correct condition to follow a desired jump command. An unconditional branch within the loop code may take the form of a jump command after a condition is already met, for example, if the loop fails, multiple jump and various other instructions will be followed unconditionally since the original condition already failed. These would typically be nested within the loop and not the loop branch itself.



Question 2 (20 points) This question requires a binary file that is available from http://www.cs.fsu.edu/~liux/courses/reversing/assignments/reverse_vtables_exe. In the binary, there are four user-defined classes and their names contain "Class". You can use Ghidra or IDA to help you answer the following questions.

1) Where are the vtables for these classes? (Hint: If a virtual function is not defined, __purecall will be used in its place.) You need to give a brief explanation how you have identified them.

```
.data:0041B674 ; ThirdClass::`RTTI Base Class Descriptor at (0, -1, 0, 64)'
.data:0041B674 ??_R1A@?0A@EA@ThirdClass@@8 dd offset ??_R0?AVThirdClass@@@8
.data:0041B674                                         ; DATA XREF: .rdata:ThirdClass::`RTTI Base Class Array'↑o
.data:0041B674                                         ; .rdata:0041B6B8↓o
.data:0041B674                                         ; reference to type description
.data:0041B678                 dd 0                    ; # of sub elements within base class array
.data:0041B67C                 dd 0                    ; member displacement
.data:0041B680                 dd -1                   ; vftable displacement
.data:0041B684                 dd 0                    ; displacement within vftable
.data:0041B688                 dd 40h                  ; base class attributes
.data:0041B68C                 dd offset ??_R3ThirdClass@@8 ; reference to class hierarchy descriptor
.data:0041B690 ; const SecondClass::`RTTI Complete Object Locator'
.data:0041B690 ??_R4SecondClass@@6B@ dd 0              ; DATA XREF: .rdata:004191FC↑o
.data:0041B690                                         ; signature
```

Above, I can see the vftable displacement for the third class (which was listed first). Following this, I went above to the references and found this snippet:

```
.data:004191EC                 dd offset ??_R4ThirdClass@@6B@ ; const ThirdClass::`RTTI Complete Object Locator'
.data:004191F0 ; const ThirdClass::`vftable'
.data:004191F0 ??_7ThirdClass@@6B@ dd offset __purecall
.data:004191F0                                         ; DATA XREF: sub_401000+A↑o
.data:004191F4                 dd offset __purecall
.data:004191F8                 dd offset __purecall
.data:004191FC                 dd offset ??_R4SecondClass@@6B@ ; const SecondClass::`RTTI Complete Object Locator'
.data:00419200 ; const SecondClass::`vftable'
.data:00419200 ??_7SecondClass@@6B@ dd offset sub_401180
.data:00419200                                         ; DATA XREF: sub_401030+14↑o
.data:00419200                                         ; .text:004010AC↑o
.data:00419204                 dd offset sub_4011A0
.data:00419208                 dd offset sub_4011C0
.data:0041920C                 dd offset sub_401110
.data:00419210                 dd offset ??_R4FirstClass@@6B@ ; const FirstClass::`RTTI Complete Object Locator'
.data:00419214 ; const FirstClass::`vftable'
.data:00419214 ??_7FirstClass@@6B@ dd offset sub_401350
.data:00419214                                         ; DATA XREF: .text:004012EA↑o
.data:00419214                                         ; sub_401310+A↑o
.data:00419218                 dd offset __purecall
.data:0041921C                 dd offset sub_4013B0
```

Above is the __purecall functions being used for first, second, and third class from the program.

2) What are the user-defined classes? You need to give a brief explanation.

I believe the user defined classes are ThirdClass SecondClass FirstClass and FourthClass:

```
.data:004191EC                 dd offset ??_R4ThirdClass@@6B@ ; const ThirdClass::`RTTI Complete Object Locator'
.data:004191F0 ; const ThirdClass::`vftable'
.data:004191F0 ??_7ThirdClass@@6B@ dd offset __purecall
.data:004191F0                                         ; DATA XREF: sub_401000+A↑o
.data:004191F4                 dd offset __purecall
.data:004191F8                 dd offset __purecall
.data:004191FC                 dd offset ??_R4SecondClass@@6B@ ; const SecondClass::`RTTI Complete Object Locator'
.data:00419200 ; const SecondClass::`vftable'
.data:00419200 ??_7SecondClass@@6B@ dd offset sub_401180
.data:00419200                                         ; DATA XREF: sub_401030+14↑o
.data:00419200                                         ; .text:004010AC↑o
.data:00419204                 dd offset sub_4011A0
.data:00419208                 dd offset sub_4011C0
.data:0041920C                 dd offset sub_401110
.data:00419210                 dd offset ??_R4FirstClass@@6B@ ; const FirstClass::`RTTI Complete Object Locator'
.data:00419214 ; const FirstClass::`vftable'
.data:00419214 ??_7FirstClass@@6B@ dd offset sub_401350
.data:00419214                                         ; DATA XREF: .text:004012EA↑o
.data:00419214                                         ; sub_401310+A↑o
.data:00419218                 dd offset __purecall
.data:0041921C                 dd offset sub_4013B0
```

```
00419190 aBadAllocation_0 db 'bad allocation',0  ; DATA XREF: .data:0041E000↓o
0041919F                  align 10h
004191A0 ; const char aFooBar[]
004191A0 aFooBar          db 'foo bar',0Ah,0      ; DATA XREF: sub_401000+10↑o
004191A9                  align 4
004191AC ; const char aMessageA[]
004191AC aMessageA        db 'message A',0Ah,0    ; DATA XREF: sub_401180+7↑o
004191B7                  align 4
004191B8 ; const char aMessageB[]
004191B8 aMessageB        db 'Message B',0Ah,0    ; DATA XREF: sub_4011A0+7↑o
004191C3                  align 4
004191C4 ; const char aMessageC[]
004191C4 aMessageC        db 'Message C',0Ah,0    ; DATA XREF: sub_4011C0+7↑o
004191CF                  align 10h
004191D0 ; const char aAnObject[]
004191D0 aAnObject        db 'an object',0        ; DATA XREF: _main+35↑o
004191DA                  align 4
004191DC ; const char String[]
004191DC String           db '45:36:0.707',0      ; DATA XREF: _main+A6↑o
```

This is due to the fact that they are the classes within the program being used in the vtables and containing message characters that are used in their functions.

```
var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+var_4], ecx
mov     eax, [ebp+var_4]
mov     dword ptr [eax], offset ??_7ThirdClass@@6B@ ; const ThirdClass::`vftable'
push    offset aFooBar  ; "foo bar\n"
push    offset unk_41F7A8 ; int
call    sub_401540
add     esp, 8
mov     eax, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn
sub_401000 endp
```

For the third user defined class, foo bar is the offset like shown above in the snippet.

```
; const SecondClass::`vftable'
??_7SecondClass@@6B@ dd offset sub_401180
                                        ; DATA XREF: sub_401030+14↑o
                                        ; .text:004010AC↑o
                dd offset sub_4011A0
                dd offset sub_4011C0
                dd offset sub_401110
                dd offset ??_R4FirstClass@@6B@ ; const FirstClass::`RTTI Complete Object Locator'
; const FirstClass::`vftable'
```

In the following snippet, the second class has multiple possible offsets that the program runs through regarding the second class that could be Message B, Message C, or the deletion.

3) What is the object hierarchy among the user-defined classes? You need to give a brief explanation.

```
data:0041B738 ; FourthClass::`RTTI Class Hierarchy Descriptor'
data:0041B738 ??_R3FourthClass@@8 dd 0                ; DATA XREF: .rdata:0041B734↑o
data:0041B738                                         ; .rdata:0041B76C↓o
data:0041B738                                         ; signature
data:0041B73C                 dd 0                    ; attributes
data:0041B740                 dd 2                    ; # of items in the array of base classes
data:0041B744                 dd offset ??_R2FourthClass@@8 ; reference to the array of base classes
data:0041B748 ; FourthClass::`RTTI Base Class Array'
```

Above is a snippet of the hierarchy descriptor for FourthClass. The order from the program is: Third, Second, First, Fourth regarding the user defined classes. Some of the classes share the same number of items in the array of base classes and all classes share the same number of attributes from the snippet.

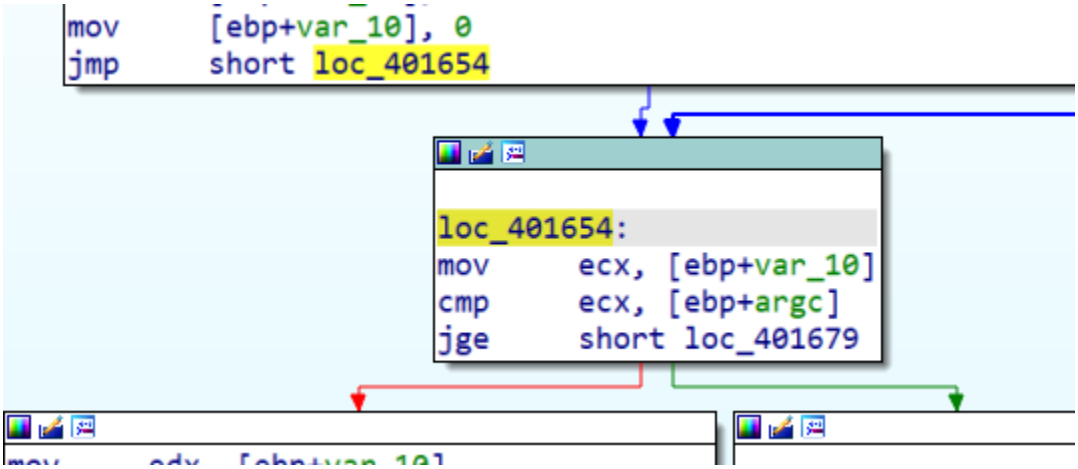
4) How many virtual functions does each user-defined class have? Briefly explain.

```
data:004191F0 ; const ThirdClass::`vftable'
data:004191F0 ??_7ThirdClass@@6B@ dd offset __purecall
data:004191F0                                         ; DATA XREF: sub_401000+A↑o
data:004191F4                 dd offset __purecall
data:004191F8                 dd offset __purecall
data:004191FC                 dd offset ??_R4SecondClass@@6B@ ; const SecondClass::`RTTI Complete Object Locator'
data:00419200 ; const SecondClass::`vftable'
data:00419200 ??_7SecondClass@@6B@ dd offset sub_401180
data:00419200                                         ; DATA XREF: sub_401030+14↑o
data:00419200                                         ; .text:004010AC↑o
data:00419204                 dd offset sub_4011A0
data:00419208                 dd offset sub_4011C0
data:0041920C                 dd offset sub_401110
data:00419210                 dd offset ??_R4FirstClass@@6B@ ; const FirstClass::`RTTI Complete Object Locator'

data:00419214 ; const FirstClass::`vftable'
data:00419214 ??_7FirstClass@@6B@ dd offset sub_401350
data:00419214                                         ; DATA XREF: .text:004012EA↑o
data:00419214                                         ; sub_401310+A↑o
data:00419218                 dd offset __purecall
data:0041921C                 dd offset sub_4013B0
data:00419220                 dd offset sub_4013E0
data:00419224                 align 8
data:00419228 dbl_419228      dq 3.14159              ; DATA XREF: sub_401400+1F↑r
data:00419230                 dd offset ??_R4FourthClass@@6B@ ; const FourthClass::`RTTI Complete Object Locator'

data:00419234 ; const FourthClass::`vftable'
data:00419234 ??_7FourthClass@@6B@ dd offset sub_401440
data:00419234                                         ; DATA XREF: sub_401400+16↑o
data:00419238                 dd offset sub_401490
data:0041923C                 dd offset sub_4014D0
data:00419240                 dd offset sub_4013E0
```

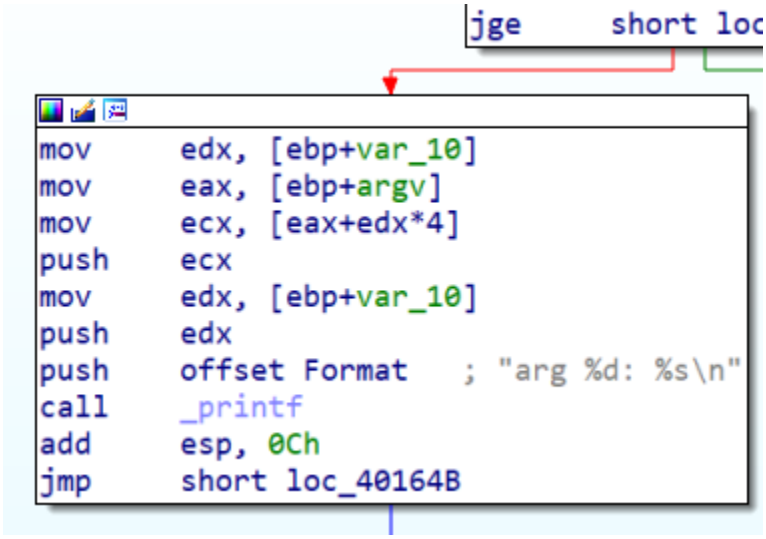
It appears each class has a connection to a ‘vftable’, however, each one has different ‘dd offset’ values. For example, ThirdClass only calls __purecall in the vftable, whereas SecondClass calls upon different ‘sub_’ calls that all contain varying ‘Message (A-C)’ offsets. So from this information, I gather that each class either has one vftable that calls upon different offsets, or each offset that is called is the vtable, thus each class has between 3-4 vftables (This was my train of thought). Either way, at the beginning of these snippets, a const _Class::’vftable’ is declared which leads me to think the true answer is the latter option, where **each class has one vftable** and within this table **multiple offsets** can occur.

Hint: More information about the RTTI structure definitions can be found from pages 3 to 5.

Question 3 (20 points) This question requires a binary file that is available from http://www.cs.fsu.edu/~liux/courses/reversing/assignments/reverse_structs_exe. For each function that is defined in the binary itself and called by function main, what basic C control construct(s) (such as a switch statement using a jump table, a loop, if-then-else, and so on) are being used within the function? You need to give a brief explanation how you have figured out the control construct(s).

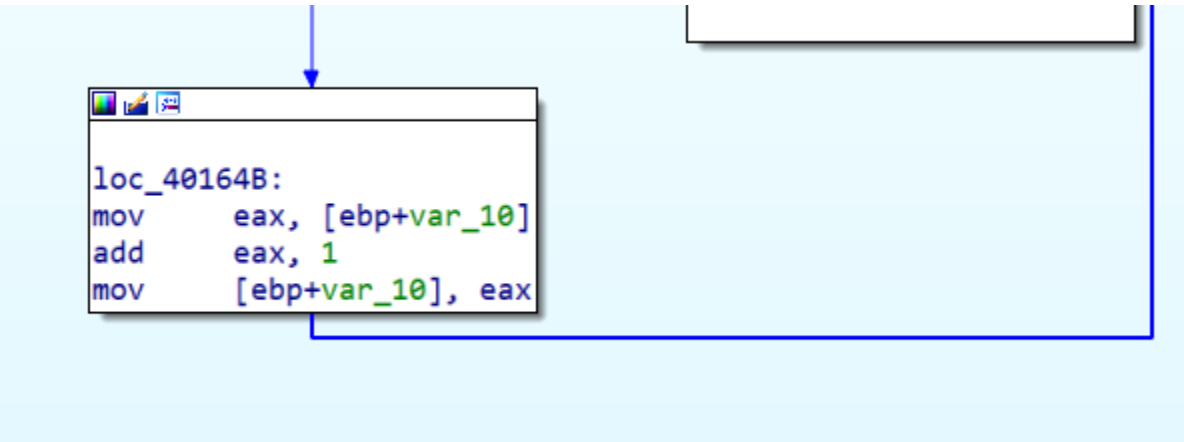


Above is the first function in main from the binary file, 401654. In this function, it looks like **an if statement** comparing ecx (which is now ebp+var_10) to ebp+argc. Following this is a jge command **(jump if >=)** pointing to 401679 (which is the arrow right). If this condition is not met (False), then the function moves on following the arrow left:

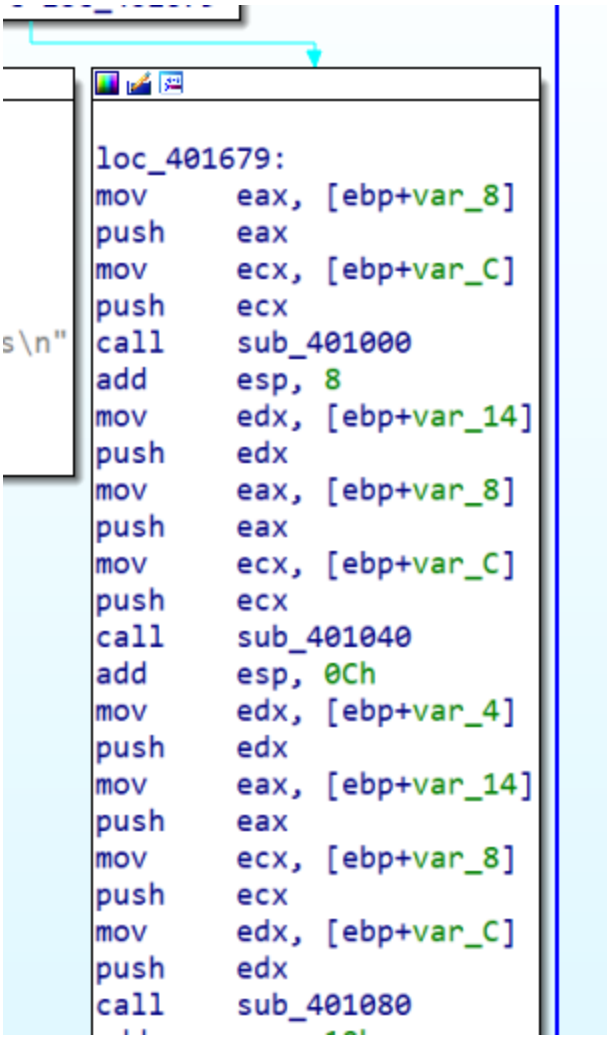


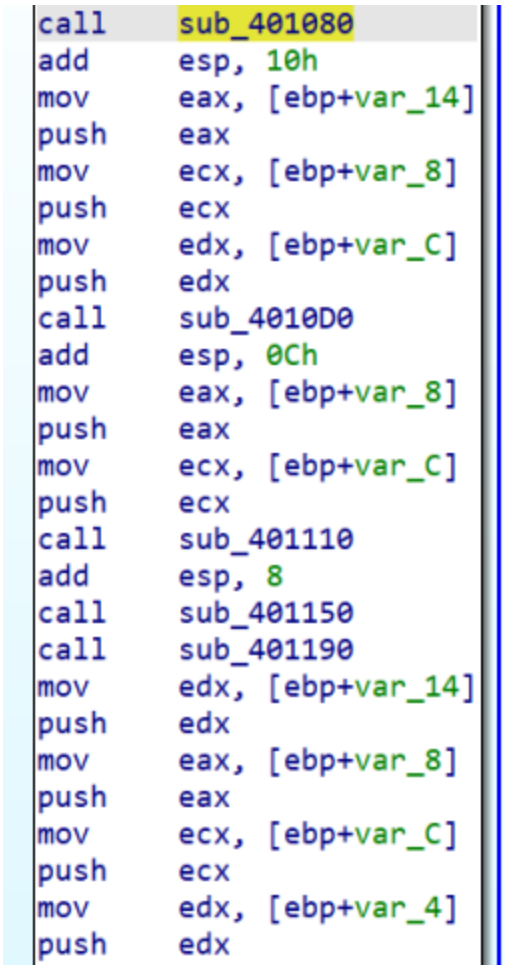
In this left side condition to function 401654, registers are moved, and the offset Format is pushed onto the stack. _printf is called, 0ch is added to esp and another jump command is executed (unconditional)

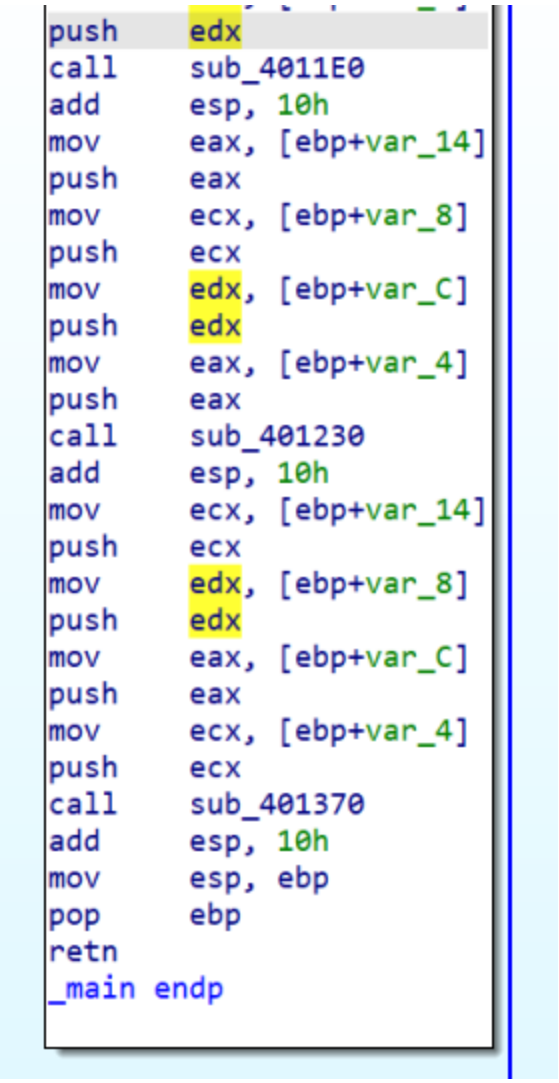
to location 40164B:



Above is the function 40164B. Here, ebp+var_10 is moved to eax, then 1 is added to it. Following this, eax is moved to ebp+var_10 (the opposite from before) and the function unconditionally moves on. This loops back to 401654 from before (the first function) meaning that **function 401654 is a Loop!** If the condition is not met on the first pass, then the pointer follows the left side, gets incremented, and follows the function again until it can pass the condition (cmp ecx, [ebp+argc]).


s\n"
loc_401679:
mov eax, [ebp+var_8]
push eax
mov ecx, [ebp+var_C]
push ecx
call sub_401000
add esp, 8
mov edx, [ebp+var_14]
push edx
mov eax, [ebp+var_8]
push eax
mov ecx, [ebp+var_C]
push ecx
call sub_401040
add esp, 0Ch
mov edx, [ebp+var_4]
push edx
mov eax, [ebp+var_14]
push eax
mov ecx, [ebp+var_8]
push ecx
mov edx, [ebp+var_C]
push edx
call sub_401080


call sub_401080
add esp, 10h
mov eax, [ebp+var_14]
push eax
mov ecx, [ebp+var_8]
push ecx
mov edx, [ebp+var_C]
push edx
call sub_4010D0
add esp, 0Ch
mov eax, [ebp+var_8]
push eax
mov ecx, [ebp+var_C]
push ecx
call sub_401110
add esp, 8
call sub_401150
call sub_401190
mov edx, [ebp+var_14]
push edx
mov eax, [ebp+var_8]
push eax
mov ecx, [ebp+var_C]
push ecx
mov edx, [ebp+var_4]
push edx



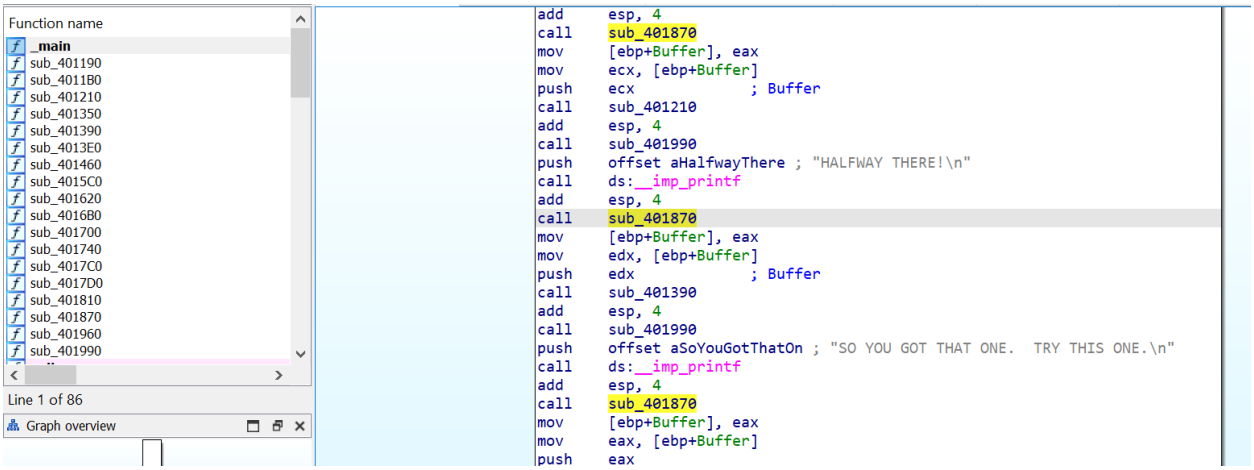
```
push    edx
call    sub_4011E0
add     esp, 10h
mov     eax, [ebp+var_14]
push    eax
mov     ecx, [ebp+var_8]
push    ecx
mov     edx, [ebp+var_C]
push    edx
mov     eax, [ebp+var_4]
push    eax
call    sub_401230
add     esp, 10h
mov     ecx, [ebp+var_14]
push    ecx
mov     edx, [ebp+var_8]
push    edx
mov     eax, [ebp+var_C]
push    eax
mov     ecx, [ebp+var_4]
push    ecx
call    sub_401370
add     esp, 10h
mov     esp, ebp
pop     ebp
retn
_main endp
```

Above is the very long function 401679 following the successful pass of the JGE instruction from function 401654. In this larger function, registers are moved and pushed, then another function 'sub_401000' is called. Following this call, 8 is added to esp and more mov/call instructions are given for the registers and various variables. The functions being called in the various call instructions throughout all contain primarily JLE instructions, this being compared to the previous function in main using JGE instruction. Following every call to another sub_* function, some value is added to the esp register, them being in order: 8, 0ch, 10h, 0ch, 8, 10h, 10h, and 10h. Following the final add instruction, ebp is moved to esp and popped, then the function is returned and _main is at the endpoint.

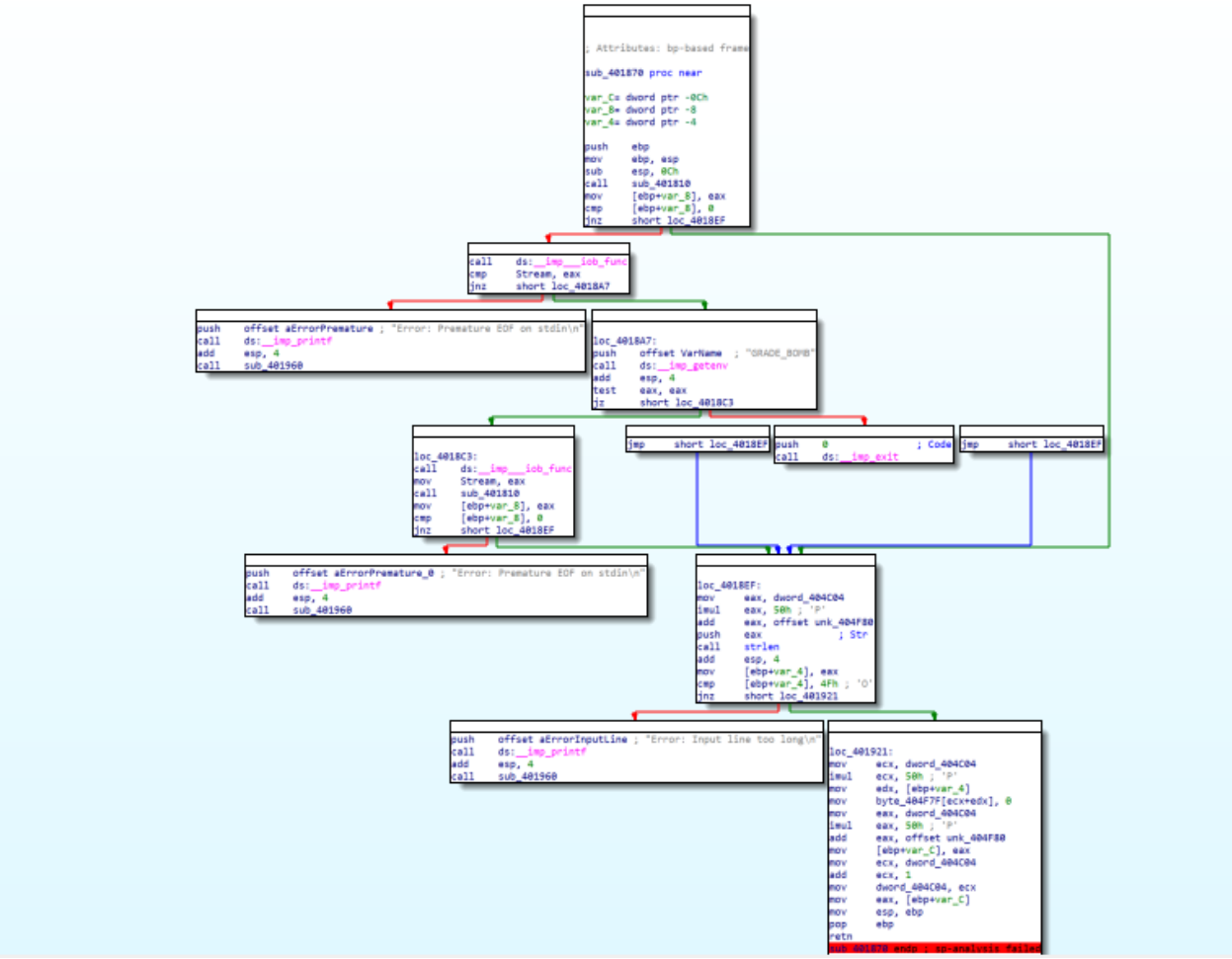
It seems for this long function that values stored in registers are manipulated before a sub_* function is called, where each one carries out even more functions that target the register until eventually returning and a value is added to esp. It is difficult to determine the C Constructs here. My initial determination was a jump table, however, the references are being called and this function never jumps, so I do not think that is right. Also, this is not storing a list of addresses. At this point, I realize that in function 401654, the JGE instruction acts as "**If ebp+argc >= ecx, goto short loc_401679**" which would be **a goto statement** (C control construct).

The following questions (4 - 5) require the following binary file that is available at http://www.cs.fsu.edu/~liux/courses/reversing/assignments/reverse_homework_exe , which is the one used for questions 5 and 6 of Homework #1.

Question 4 (30 points) Find the inputs for the binary file so that it will pass Phase 3 using static and dynamic analysis. Identify the type of statement (or statements) used to determine which of the 8 paths to follow. Give an explanation with screenshots of the key steps of your analysis process in addition to the solved password.

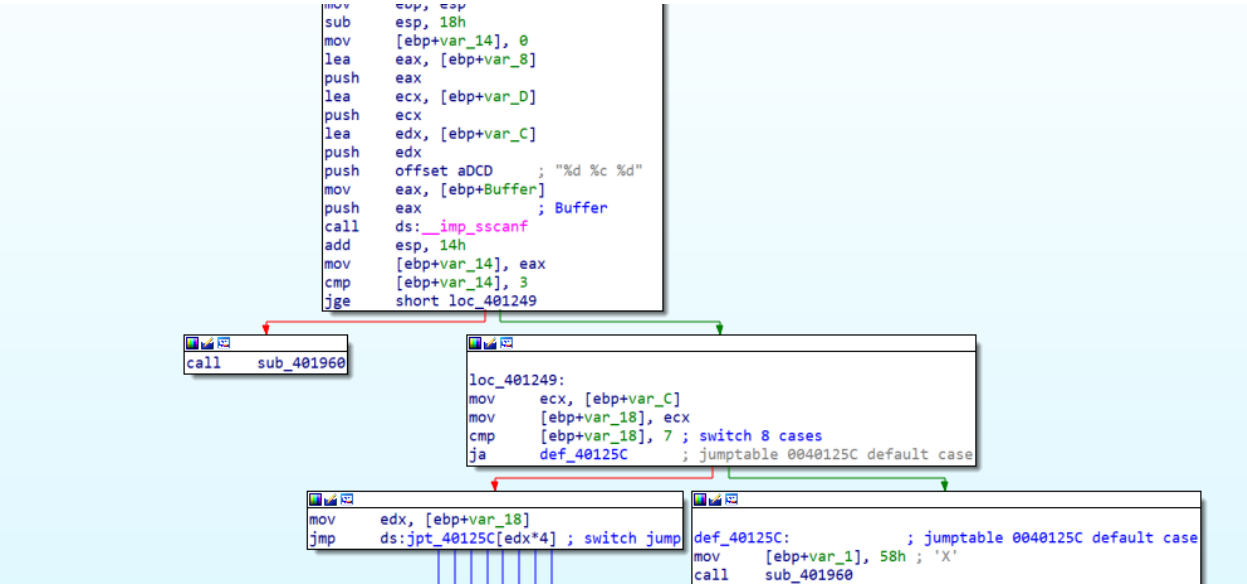


Above in function main, past the “Half way” point (phase 2) I see a call to another function sub_401870.

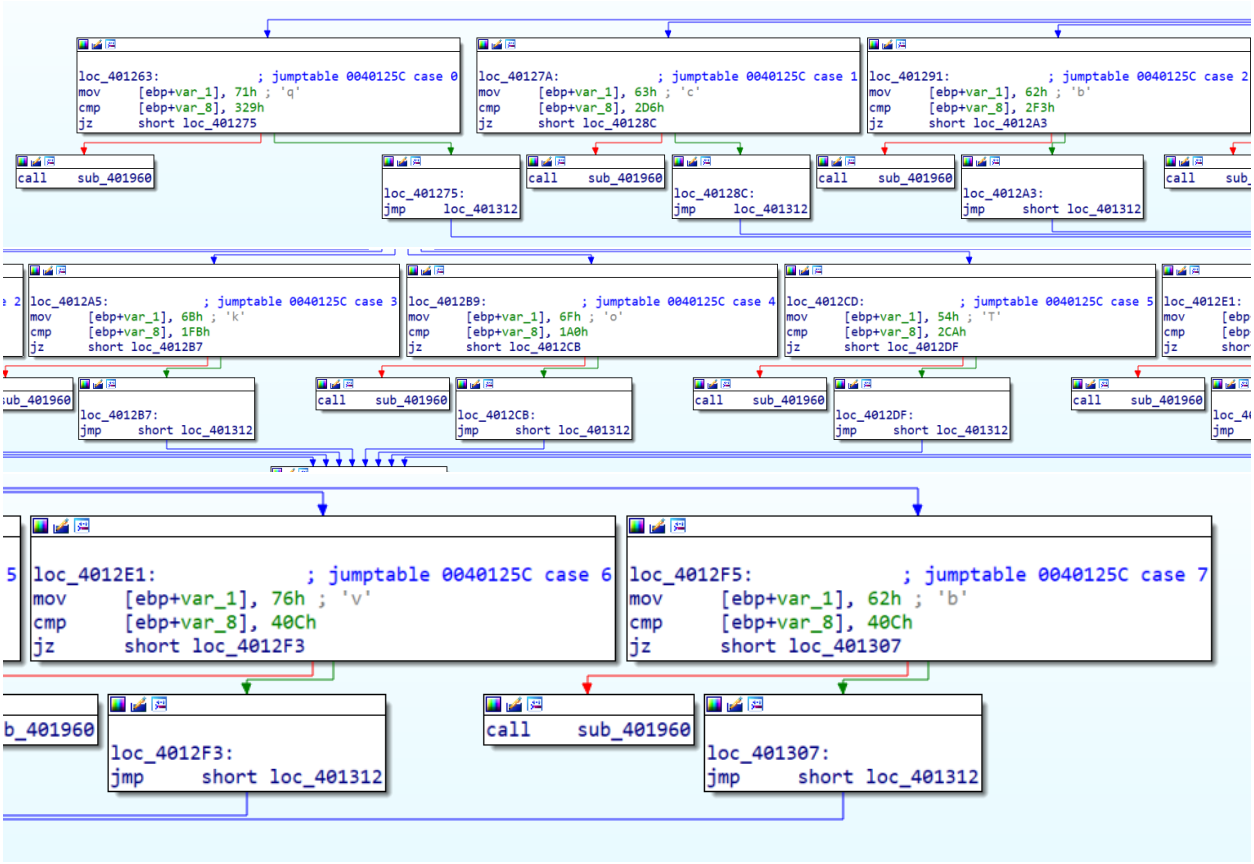


This function is rather large and has multiple conditional jumps that could possibly end with the bomb exploding. I believe this is connected to phase 3. Originally, I thought that ‘GRADE_BOMB’ was the input key, however, after snooping around, I discovered the jump table containing the 8 possible paths to follow (sub_401210):

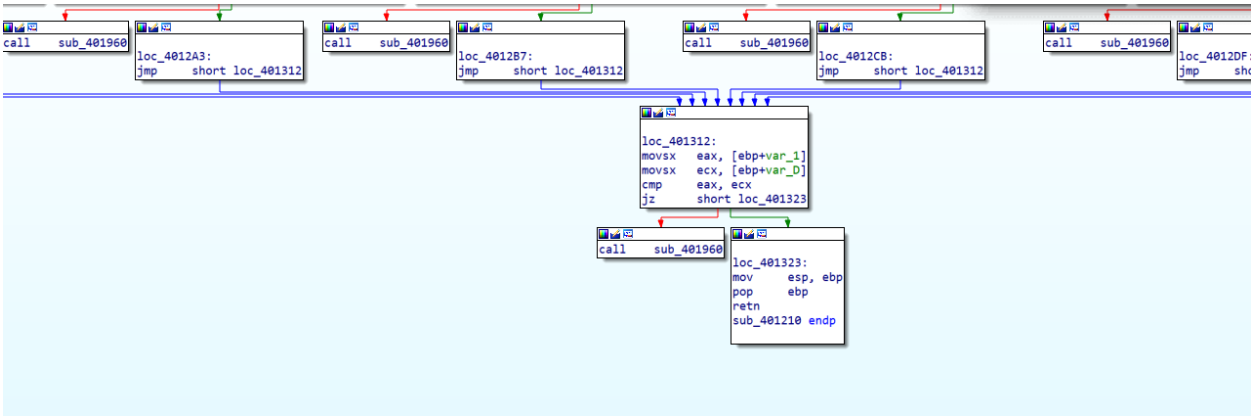
```
.text:00401328 jpt_40125C       dd offset loc_401263    ; DATA XREF: sub_401210+4C↑r
.text:00401328                  dd offset loc_40127A    ; jump table for switch statement
.text:00401328                  dd offset loc_401291
.text:00401328                  dd offset loc_4012A5
.text:00401328                  dd offset loc_4012B9
.text:00401328                  dd offset loc_4012CD
.text:00401328                  dd offset loc_4012E1
.text:00401328                  dd offset loc_4012F5
.text:00401348                  align 10h
.text:00401350
.text:00401350 ; =============== S U B R O U T I N E =======================================
.text:00401350
.text:00401350 ; Attributes: bp-based frame
.text:00401350
```



This is the beginning of the function. It looks like the string DCD, or %d %c %d, is pushed onto the stack and input is compared to it. This is the beginning of the input. A conditional jump is at the bottom where if condition is not met the bomb is detonated. Following the correct path, a default jump path is given where if JA condition IS MET, the X is moved onto the ebp+var_1 variable and the bomb is detonated. So here JA condition should follow the red arrow to move to the switch jump portion of the function. Here there are 8 cases where the jump table could follow.



Here, it looks like the conditional jump is comparing the input to one of 8 variables. In order, they are ‘q’, ‘c’, ‘b’, ‘k’, ‘o’, ‘T’, ‘v’, and ‘b’. Since this jump table is like an array of small functions, the input after DCD will have to be one of the previously listed variables to continue. For example, the string DCDq should move the program onward, whereas the string DCDX will cause the bomb to detonate.

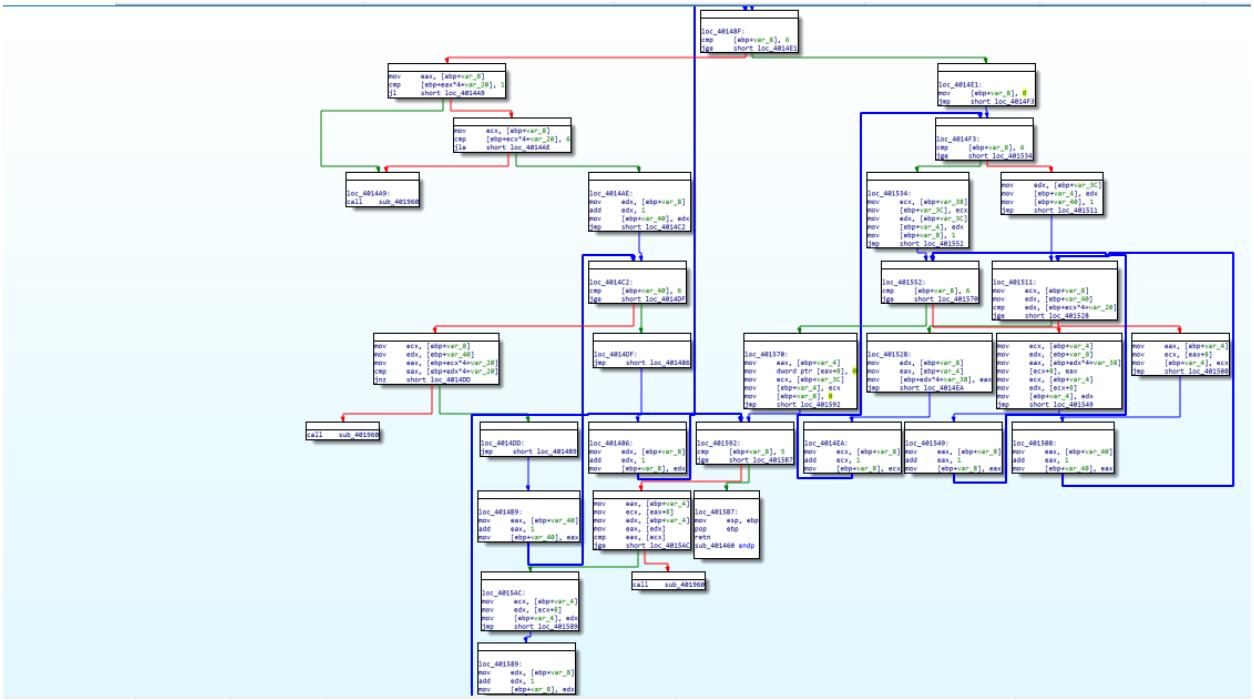


I believe this is the case because the cases in the jump table do not seem to call each other or appear meant to be performed one after another. If one of the variables comes after the original DCD from the first conditional jump, then the program moves on to loc_401312. If not, then the bomb detonates (if the 4th character is not in the jump table). At loc_401312, ebp+var_1 is movsx (read contents of register as a word) to eax and movsx ebp+var_D to ecx and compares them. Another conditional jump is instructed where if it is not met then the bomb detonates and if it does meet then the function returns.

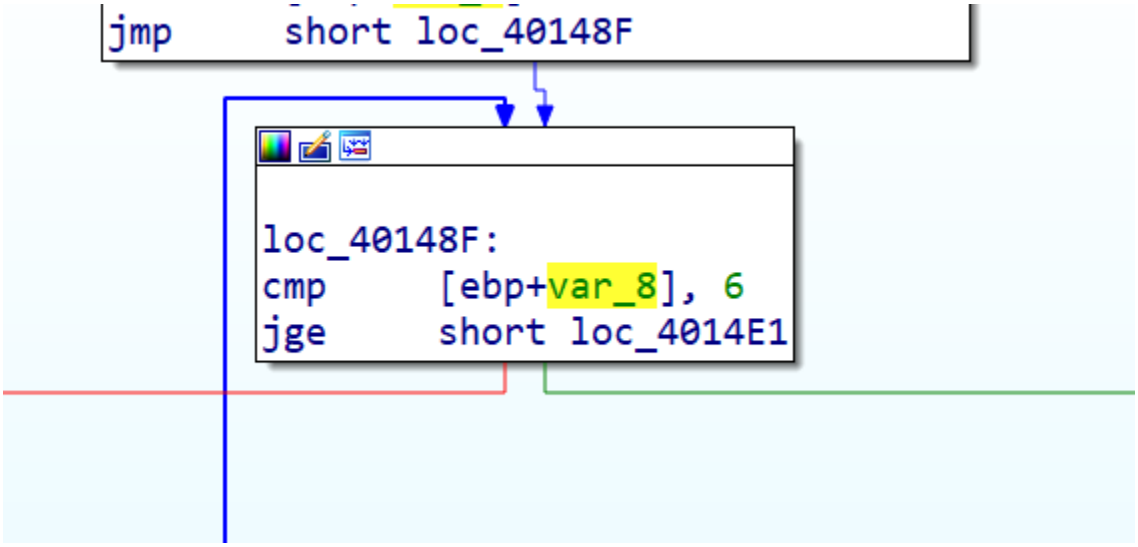
Question 5 (30 points) Find the input for the binary file so that it will pass Phase 4 using static and dynamic analysis. Use the call graph to find out what is unique about this section’s function (what type of function is it?). Give an explanation with the screenshots of the key steps of your analysis process in addition to the solved password.

```
add     esp, 4
call    sub_401990
push    offset aGoodWorkOnToTh ; "GOOD WORK!  ON TO THE NEXT...\n"
call    ds:__imp_printf
add     esp, 4
call    sub_401870
mov     [ebp+Buffer], eax
mov     ecx, [ebp+Buffer]
push    ecx             ; Buffer
call    sub_401460
add     esp, 4
call    sub_401990
xor     eax, eax
mov     esp, ebp
```

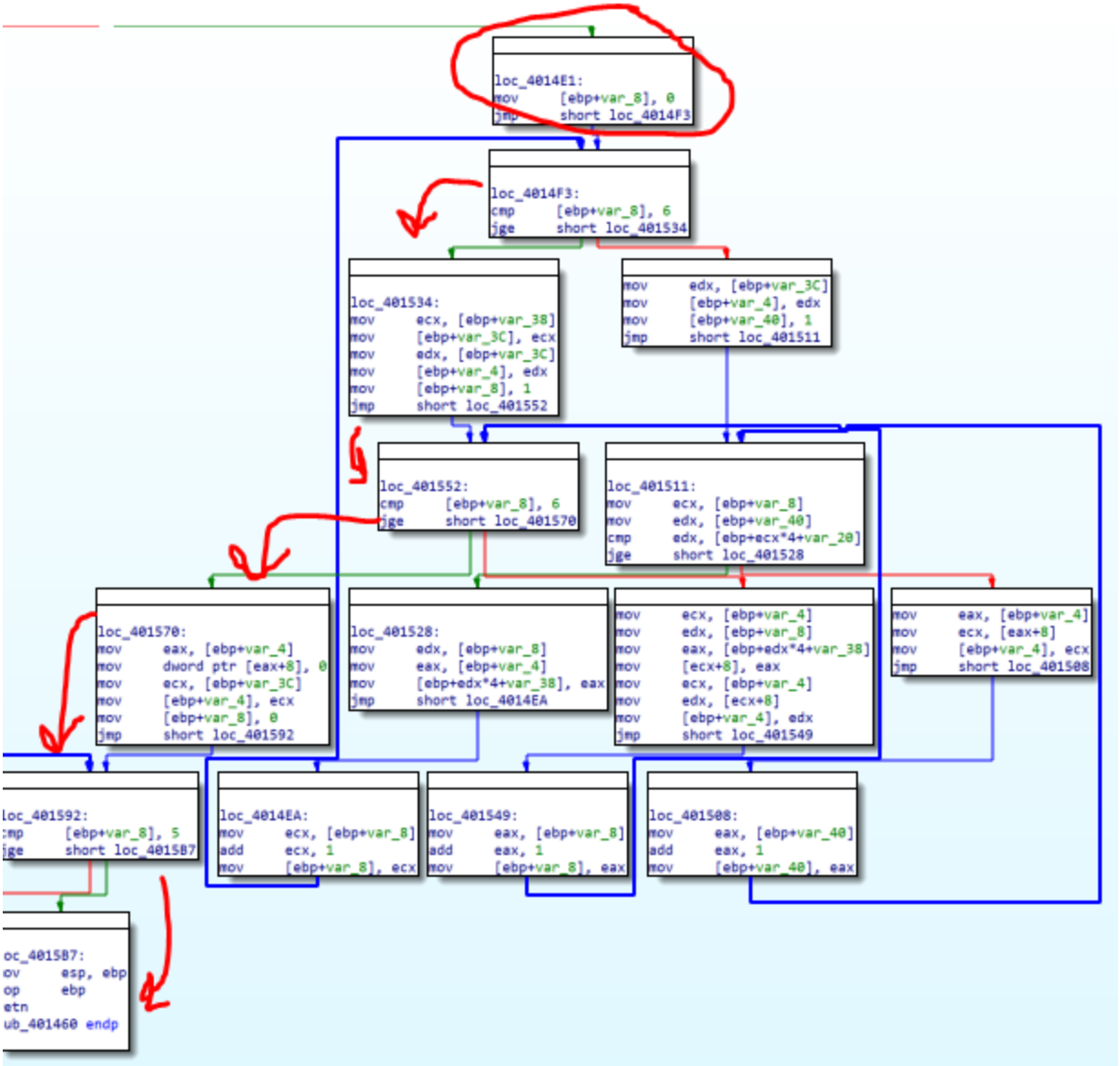
Here should be the function being called for phase 4. This was found in main.



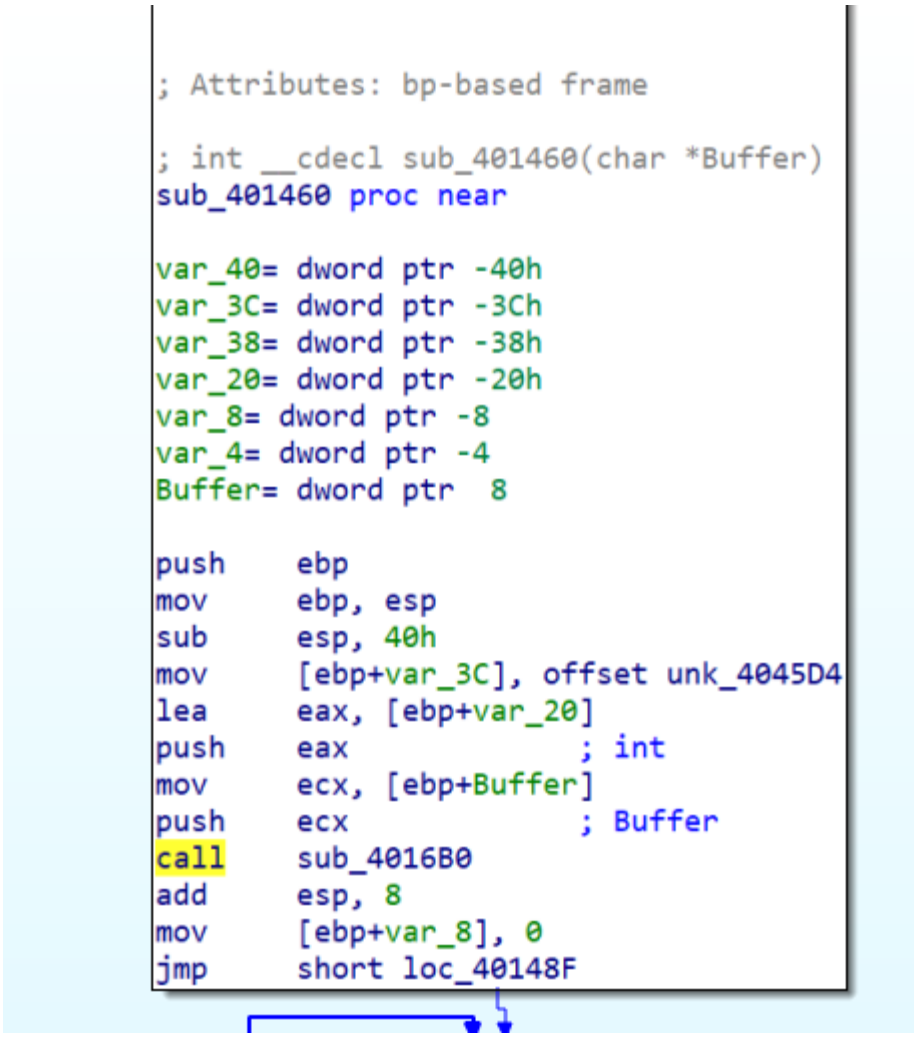
Above is a snippet of the function for phase 4. It is the largest one so far, however, seems to boil down to a complicated loop that is checking 6 against ebp+var_8:



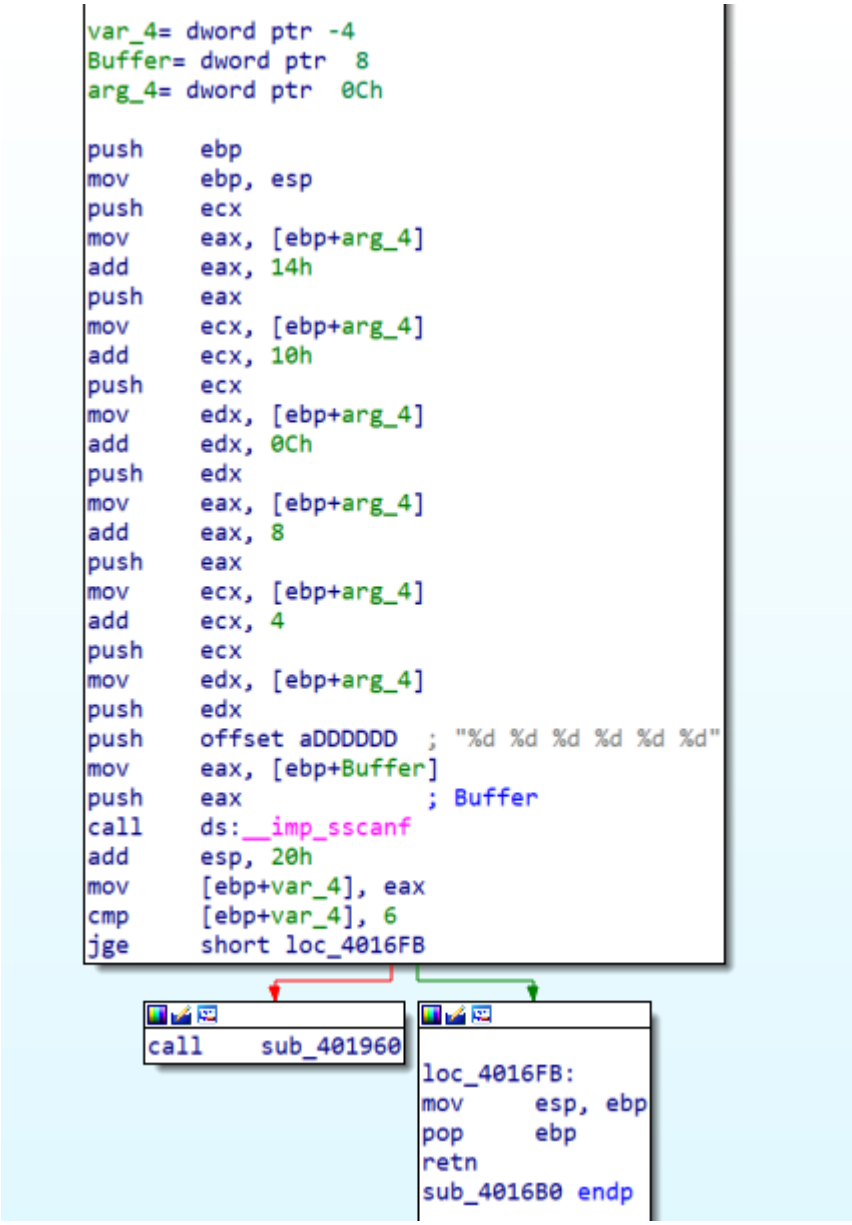
For this conditional jump, if it does not meet the condition, then a series of nested checks and loops move and compare var_* values until it can go back up to loc_40148F and succeed the condition. Once the function successfully meets the condition, it moves on to the right:



The edited red arrows show the correct path that leads to the endpoint of the function. For some reason, my call graph functionality in IDA was not working properly, so I am trying to manually look at the calls. Looking at the calls, the only other functions called here are the ones for the bomb detonation, and 4016B0.


; Attributes: bp-based frame
; int __cdecl sub_401460(char *Buffer)
sub_401460 proc near
var_40= dword ptr -40h
var_3C= dword ptr -3Ch
var_38= dword ptr -38h
var_20= dword ptr -20h
var_8= dword ptr -8
var_4= dword ptr -4
Buffer= dword ptr 8
push ebp
mov ebp, esp
sub esp, 40h
mov [ebp+var_3C], offset unk_4045D4
lea eax, [ebp+var_20]
push eax ; int
mov ecx, [ebp+Buffer]
push ecx ; Buffer
call sub_4016B0
add esp, 8
mov [ebp+var_8], 0
jmp short loc_40148F

This leads us to this function:



Here, the above function takes the user input and compares it. The beginning of this function takes arg_4 (0Ch) and begins moving and adding it around registers. In order, it goes to eax and 14h is added, then it goes to ecx and 10h is added, then it goes to edx and 0Ch is added, then to eax and 8 is added, then to exc once again and 4 is added, then finally to edx where it is pushed and the offset DDDDDD is pushed after it. It looks like below the sscanf that the buffer is moved to ebp+var_4 and 6 is compared here. So, I am led to believe that **DDDDDD** is the input for phase 4 since it is pushed onto the stack as the buffer, then moved to ebp+var_4 and compared. It could also be **'\x0Ch%d%d%d%d%d%d'** (since agr_4 is 0Ch), but ultimately, I am not totally sure of this as the answer. Following the comparison is a conditional jump that either leads to detonation if not met or to the endpoint and return of this function. Since this is phase 4, if the final input is correct the bomb is defused.

Extra Credit Problem

Question 6 (10 points) Suppose that we run the following program (on the next page) with a single command line argument "%08X%08x%08X%08x%08X%08x%08X%08x %llX %llx", what would the printf at 0x08048503 output? Be as specific as you can. When there is not enough information to determine the content for a particular word, provide as much information as possible (such as same as a particular register); when no information is available, just stay so.

This would print a hexdump of long long ints within the program, among which 'I_am_the_King' should be present in hexadecimal. Since '%llX %llx' is present at the end of the command line argument, I believe the program would get confused by this input, like the example from class, and run the 'I_am_the_King' function without knowing the proper input. I am unsure regarding the amount of information available from this snippet, however, given that the long long int is being executed as a command line argument, registers and possibly their values should be obtained by this argument.