**Final Exam**

Collin Hays, Bryce Reynolds, Austin Brown, Samantha McGregor

**Group ID:** 2467209

**Password:** MWXLNFAD

**SHA1 BAD FILE:** D90F6718C73184184EA610033AC55A2EBAC89755

**GOOD FILE** At the end of this document there are some notes about the good file

**Method:** We worked on this final exam together while on a Discord call. Below are our meeting times.

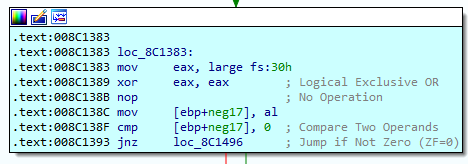
* Friday 12/04/2020 5pm - 9pm
* Monday 12/07/2020 6:30pm - 9:30pm

**Tools and resources we used for analysis:**

* IDA
* Intel x86 Manual
* BNinja
* Ghidra
* gdb

**Issues we encountered:**

* Collin, Bryce, and Samantha were unable to run the final\_exam.exe in their Windows VM. Installed IDA on our home computers to fix the issue.
* Ghidra’s decompiled code was a bit messy and hard to read, so we abandoned the tool for analysis. It seemed to be unable to identify the “main” function.
* Binary Ninja for desktop was also unable to identify the main function.
* Unfortunately Dr. Denton gave us the wrong binary to analyze.



* This section of code occurs four times, but seems to be modified to not do anything. It looks like this could be used to obfuscate imports and exports, etc...

**Does the “group id” affect the correct password?:**

* We formulated a hypothesis that assumed that each group received the same binary, but a different **“group id”** based on Dr.Denton’s wording in the final assignment prompt*.* We also assumed that the different **“group id”** would result in a second correct password that is different from ours.
* To test this hypothesis, Austin wrote the following fuzzer in python

|  |
| --- |
| import os  count = 1  while(count < 9999999):  exe\_string = "final\_exam.exe " + str(count) + " 1 >> out.txt"  os.system('echo ' + str(count) + ' >> out.txt')  os.system(exe\_string)  print(count)  count = count+1 |

* It was determined **“group id’s”** are seven digit integers.
* This code determined that there was at least one other **“group id”** that was counted as valid, but it ran too slow to find all of the other values.

**Final\_exam.exe Process:**

* First step was to identify the main function of final\_exam.exe. We ran final\_exam.exe in IDA, Ghidra, and BNinja, and IDA was the most reliable program for identifying the main function. We began our analysis using IDA.











* *Str1* looping through and changing the string value of **“password”** to the actual password of **“MWXLNFAD”**. Each step reads from left to right on each row.

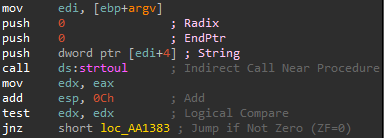


* Original password & final password in IDA.

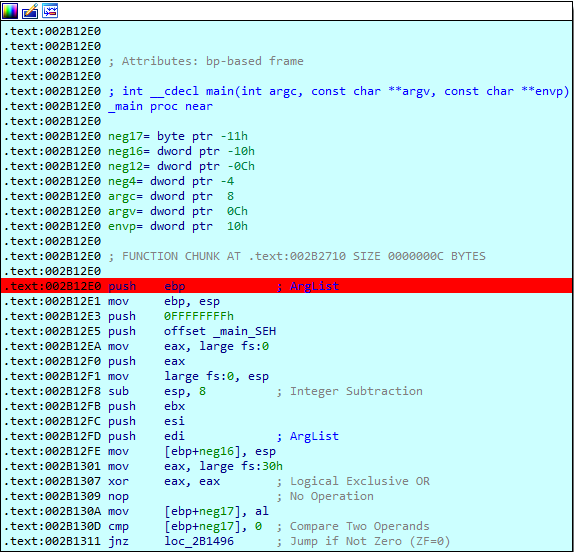




* In this screenshot from Bryce, the output is shown for our group number and password being correct. This was the most exciting part of the analysis! The second screenshot shows that only the first eight bits are checked, and anything beyond it doesn’t matter. We used zeros as an example, but tested numerous values beyond the eight bits.



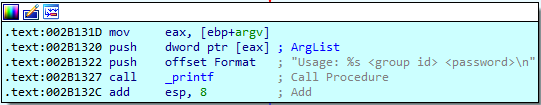
* *Strtoul* converts a string to an unsigned long integer. It then moves the value from *eax* into *edx.* 
  + It adds twelve to *esp,* and then does a test against *edx* using a logical compare.
  + If the value is not zero, jump to *loc\_AA1383.*



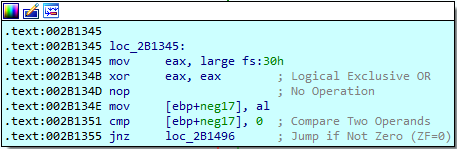
* The main block of code defines variables and sets their initial values. Since this main function is a function itself, it pushes the base pointer to the stack to eventually be popped back to *ebp.*
  + Next it sets the current stack pointer to the new base pointer for a local stack within this function.
  + Allocates space on the stack.
  + It then backs up registers from the parent function onto the stack to be restored later.
  + Updates the position of the backed up stack pointer to give space for arguments returned by the main function.
  + This program seems to make use of *large fs:##* to check for environmental conditions that could be checking for a vm or debugging. These can also be used to get the next available memory address and exception handling.
  + This function also generates zero by *XOR* *eax* with itself, and stores it as a byte at seventeen spots below the stack base pointer. It then compares that value to zero. If for some reason the value stored in memory is not zero or has changed jump to an exit.



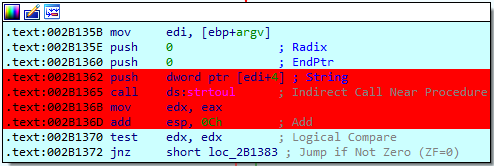
* The *cmp* statement checks the number of command line arguments entered. If anything other than three is entered, it jumps to the code where instructions for use are printed.



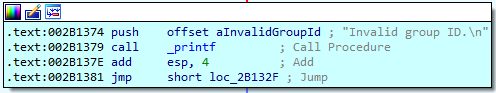
* Here the program moves the name of the executable to *eax,* andpushes it on to the stack to be printed with \_*printf*.
  + Push instructions for use to the stack to be printed with *printf.*
  + Call \_*printf* to print to the console and clean the stack.



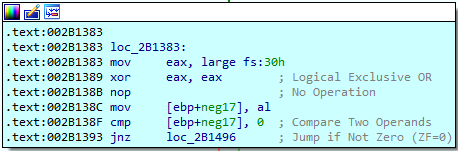
* *loc\_2B1345* moves the thirtieth byte of the *fs* register into *eax*. It usually contains the address of the PEB which can be used to detect whether or not the program is being debugged.
  + Generate zero by *XOR eax* with itself, and store it as a byte at seventeen below the stack base pointer. Then compare that value to zero. If for some reason the value stored in memory is not zero, or has changed, jump to an exit.



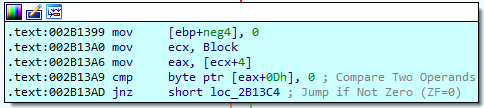
* The code here moves the address of the argument list into *edi*. Then *edi+4* is where the **“group id’** should be.
  + Push the arguments for *strtoul* onto the stack and call *strtoul* to convert the string to an unsigned long integer. This function modifies the values of all registers a, b, c, d, but the integer value of the **“group id”** is returned in register *eax*.
  + The result in *eax* is copied into *edx.*
  + The stack pointer is restored to its previous state.
  + The result is then tested using the test instruction, performs a logical *AND,* and then sets flags but does not store the result.
  + If the result is zero then continue to an exit that prints an error for invalid **“group id”**. If the result is not zero then there is a possibility that a valid group code was entered so jump to the next step.



* This pushes the error to the stack for invalid **“group id”.** It then calls *printf* to print the error to the console, and restores the stack pointer and jumps to an exit.



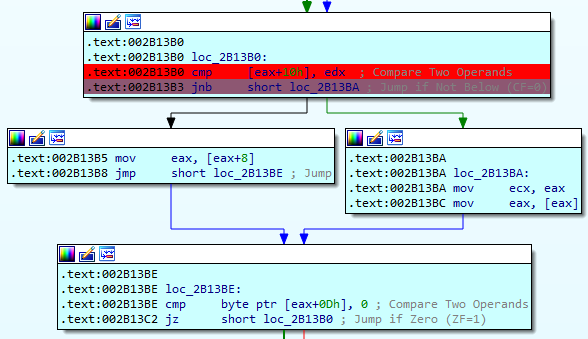
* *loc\_2B1383* moves the thirtieth byte of the *fs* register into *eax*. This usually contains the address of the PEB which can be used to detect whether or not the program is being debugged.
  + It next generates zero by *XOR eax* with itself, and stores it as a byte at the seventeenth position below the stack base pointer. Then compare that value to zero. If for some reason the value stored in memory is not zero, or has changed, jump to an exit.



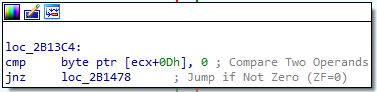
* This moves zero into the first value above the base of the stack. It then clears the four bytes of all ones and replaces them with all zeros.
  + Next move Block into *ecx*. It appears that the block seems to be a memory address. In one case the block contains *0x00FBF050*. It then moves the contents of the memory at the address of Block+4 into *eax*. In the same case as above, it loads *0x00FC37E8* into *eax*.
  + It then compares zero to the value in the memory byte at *eax+13* which in the original case mentioned above contained ‘00’. This causes the zero flag to be set by the *cmp* instruction. Therefore, the jump instruction does not jump to an exit and continues on.



* A no-op is performed.This literally does nothing.



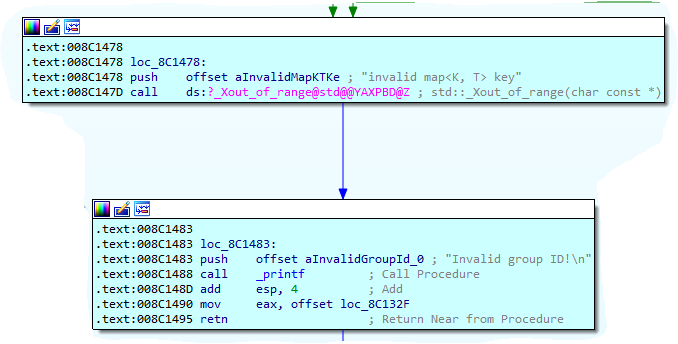
* These blocks of code compare the value stored in *edx* with the value stored in memory at *eax+10h* which is *0xFC37F8* and holds the value 00257314. This does not match the value in *edx* currently and is below, therefore the CF was set. The value in *edx* was 0025A589. Since the CF was set, the jump instruction does not execute and the code continues on the left path.



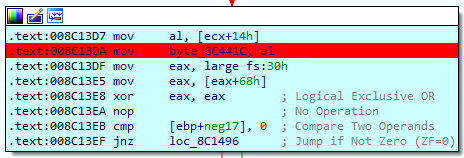
* *loc\_2B13C4* compares zero with the value stored in program memory at *ecx + 13*. If a valid group code is entered, the value in *ecx+13* will be zero and the program will not jump to an exit.
  + If the value in *ecx+13* is not zero, the program will jump to an exit.



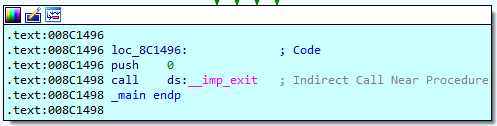
* This compares the value stored at *ecx + 16* with the value in *edx*. If the value in *edx* is below the value stored in memory at *ecx+16,* it then jumps to an exit. This indicates that the **‘group-id’** is not valid.



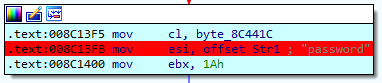
* *loc\_8C1483* is the exit patch that occurs when a invalid **“group id”** is entered.
  + Push arguments for the ?\_Xout\_of\_range@std@@TAXPBD@Z function and then call the function and restore the stack pointer.
  + Push arguments onto the stack for *printf* that indicate if an invalid **“group id”** was entered. Call *printf* then restore the stack pointer and move a value from RAM into *eax.*



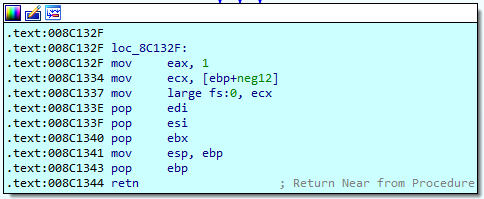
* This code moves the least significant eight bits stored in memory at *ecx+20* into the last eight bits of *eax*.
  + It then does the *cmp* that does nothing where *eax* from the *fs* register is immediately overwritten with zero, and then written to memory and checked if its value has changed. If the value has changed and is no longer zero after being read from memory then jump to and exit.



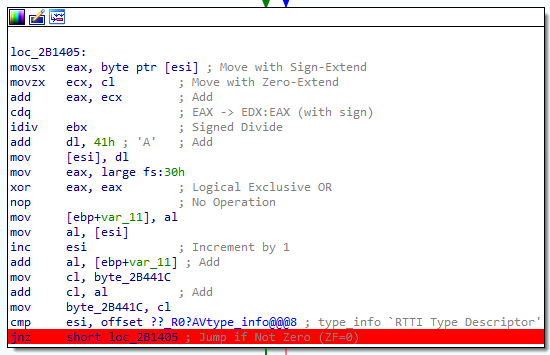
* *loc\_8C1496* is the exit function that is jumped to after the FS register is read into *eax* then overwritten with zero and checked to see if zero magically changes while it is stored in RAM.
  + It then pushes zero to the stack which is likely to return zero from the main function.
  + Call the *imp\_exit* function which ends the execution.



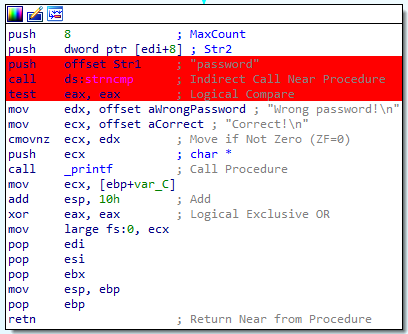
* This block moves a byte from RAM into the least significant eight bits of *eax*.The *esi* register holds the current character of the password as it is being calculated, and sets the offset in esi to the beginning of the first character. It then moves the immediate value of twenty-six into *ebx*.



* *loc\_8C132F* begins by setting *eax* to one.
  + Set *ecx* to the value stored in a variable on the stack.
  + Set the address in the zero value of the *fs* register to the value that was recalled in the last instruction from the stack.
  + Restore the values in the registers to their corresponding original values before the main function was called.
  + Restore the value of the stack pointer.
  + Return to the calling function.



* *loc\_2B1405* 
  + In a previous block, the string “password is loaded into memory.
  + This block is going to loop through that string and perform various operations one character at a time.
  + The first thing that it will do is move whatever character that esi is pointing to to eax.
  + It will then replace the value in ecx with to lower 8 bits of ecx. The value that ecx holds was determined in loc\_1B13BA.
  + The value in eax is now divided by 26. The value 26 is stored in the ebx register. The result of the division is stored in eax, and the remainder is stored in edx.
  + Then a value of 0x41or ‘A’ is added to the remainder stored in edx.
  + This result is stored at the value pointed by esi. This replaces the original value in memory.
  + The stack pointer is then incremented.
  + The value of esi is then compared to an offset to determine if all of the characters have been processed. If they have, then the program jumps to the next block. Otherwise, the current block is repeated on the next character.



* This block of code pushes the arguments for the *strncmp* onto the stack *Str1* which is the calculated correct password. *Str2* is the password we entered as a command line argument and 8 is the maximum number of characters to be compared.
  + There is an indirect call to *strncmp* to compare the calculated correct password to the password we entered.
  + It then uses the test instructions to set flags using the result of *strncmp.*
  + Loads “correct” into *ECX,* and *ECX* will be printed using *printf.*
  + Loads “incorrect” into *EDX.*
  + Checks the zero flag condition of move *EDX* into *ECX* (move incorrect to *ECX* to be used by *printf*).
  + Calls *printf.*
  + Cleanup the stack and exit.

**SHA1 GOOD FILE:** 7BF6DABEFE723C1176DDADE58FF3E912CBC65B7D

* The file is packed with UPX.
  + Initially, I tried unpacking it by setting a breakpoint at the end of the executable and running it in the debugger. I then took a memory snapshot and reanalyzed the file. At this point a massive amount of functions appeared. I could identify the main function, but I couldn't identify any function calls.
  + The next method I tried was to just use UPX to unpack the executable. I downloaded upx.exe from their website and ran

upx.exe -d bintext\_packed.exe -o bintext\_unpacked.exe. This successfully unpacked the executable.