

# Assignment 3 Final Write up

---

## E Enterprises Binary Report - Evan Haaland

---

### Executive Summary

I was given a program called challenge 2 to analyze. This program is a Linux program what has the user enter in a pass phrase. If the pass phrase is wrong it says access denied. If the user is able to guess the password it would say access granted. However, in my discovery we can actually get passed the password check without knowing the password.

When looking at the program I saw that there was no debug symbols which made it a bit harder to see the names of certain variables and functions. However, I was able to find the main function which was a massive help. From there I could see that there was another function called *f* that was called. I saw when I put no argument in the program would jump to the end without calling the *f* function. This is a good technique that could confuse someone when looking at it. At first I thought the program did nothing until I dug in a little deeper.

When I started to investigated the *f* function I saw that it was very long. I also noticed that there was no other suspicious functions. This gave me a sign that this could be very important. As I would step through I would see many commands modifying single registers using commands that I have seen in encryption. This made me think that the *f* function would take the passphrase typed in by the user and perform some sort of encryption on it to see if it matches a comparison check in main. When I noticed this my first instinct was to look at main for this comparison.

When I saw the comparison in main I saw that the value it was being compared to was visible and a set arbitrary value. This made me think that I could use this value and put it back through the encryption algorithm in the *f* function to get the passphrase. Luckily, I noticed a vulnerability that made the process much easier. I saw that I could actually set the value of the password check to the value that it needs to be compared to. Once I set that value to the set value I was able to bypass the program and get the access granted method. This was without even knowing the password.

Even though I have cracked the password I wasn't entirely sure if I understood the full functionality of the *f* function. This is because I saw a lot of interesting commands that I think were put in to throw the reverse engineer off. I noticed that the program would recourse a bunch randomly but it would always end by replacing the return value with a set number. This made me realize that maybe there is no pass phrase that could be used to get access granted.

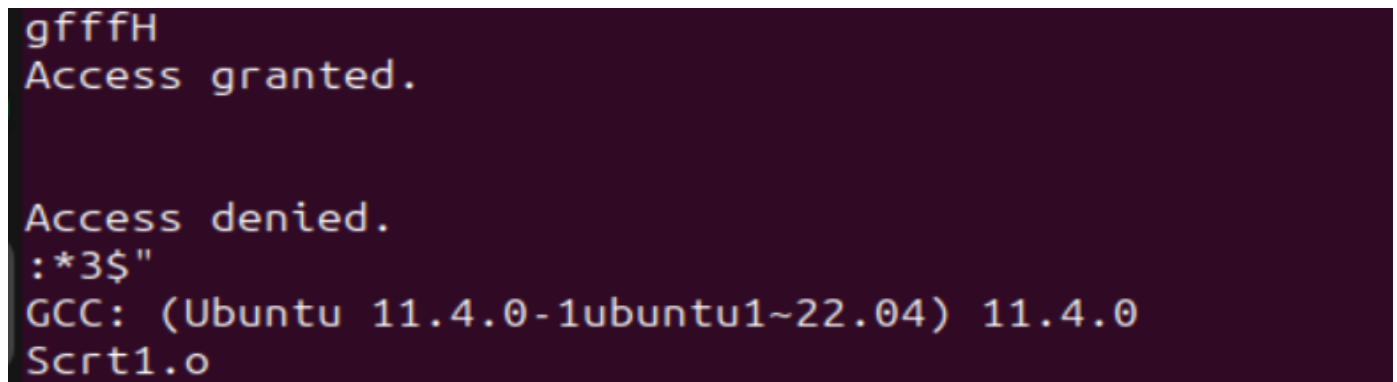
In hsort, the program is not secure. Anyone with access to decompiler could figure out what they need to bypass the password check. Anyone with a disassembler/debugger can see what they need to set to get the access granted for the answer. Without Obfuscation techniques it was easy to see where the vulnerability was.

## Technical Summary

The first thing I did when presented with the binary was running

`file challenge2`. This should be that I was looking at an unstripped 64-bit Linux elf file.

I would then run `strings challenge2` to see if there was any strings I could see that would provide insight to what I was looking at. When I did this I was able to see **Access Granted** and **Access Denied**. This made me realize that I could be trying to reverse something with a passcode.

A terminal window with a dark purple background. The text displayed is: gffffH, Access granted., Access denied., :\*3\$, GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, Scrt1.o. The text is in a monospaced font with a light green color.

```
gffffH
Access granted.

Access denied.
:*3$
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Scrt1.o
```

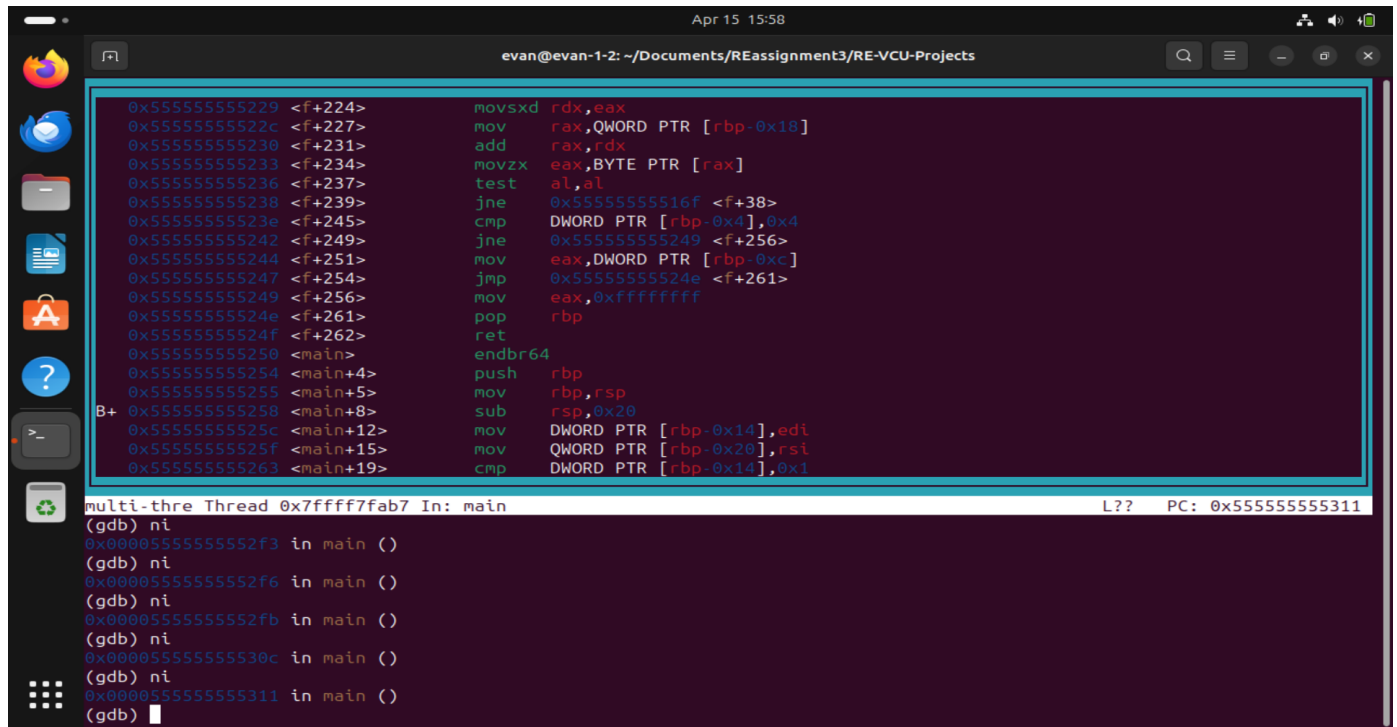
I then ran `objdump -M intel -d challenge2 > challenge2intel.asm` I got an an assembly view of the program and can see the main function as well as a function called `*f`. I forget to change this to intel syntax but I was still able to perform some basic static analysis of the binary. By seeing that I could see main and another function that it calls helped me triage past a lot of work. I already had some areas of interest to look at.

I then decided it was time to perform dynamic analysis with GDB. I started by using `chmod +x challenge2` to get it working as an executable. When I ran it `./challenge2` it would not print any information. I then ran `./challenge2 hello` and I received the **Access denied** message. This proved my theory of it being some sort of password check program.

When I ran the debugger `(gdb) challenge2` I was able to see that there where no debugging symbols. This made me realize that I might need to take a deeper look at some of the functions being called since they won't have obvious names. I set break points at main and f to see what happens at these functions. `(gdb) break main` `(gdb) break f`.

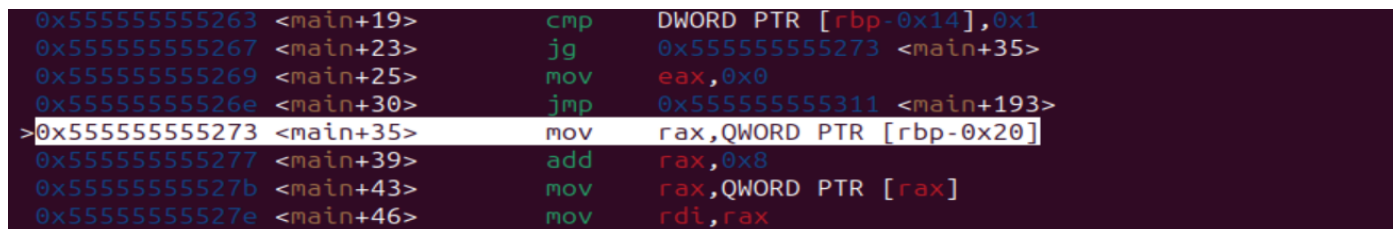
I needed to look at the disassembly at this function. `(gdb) disassemble`. However, I wanted to find a way to get a more visible way of looking at the assembly step by step. I ran `(gdb) layout asm` and I

was given a way to look at the assembly as I step through.



```
evan@evan-1-2: ~/Documents/REAssignment3/RE-VCU-Projects
0x55555555229 <f+224>    movsxd  rdx,eax
0x5555555522c <f+227>    mov     rax,QWORD PTR [rbp-0x18]
0x55555555230 <f+231>    add     rax,rdx
0x55555555233 <f+234>    movzx   eax,BYTE PTR [rax]
0x55555555236 <f+237>    test    al,al
0x55555555238 <f+239>    jne     0x5555555516f <f+38>
0x5555555523e <f+245>    cmp     DWORD PTR [rbp-0x4],0x4
0x55555555242 <f+249>    jne     0x55555555249 <f+256>
0x55555555244 <f+251>    mov     eax,DWORD PTR [rbp-0xc]
0x55555555247 <f+254>    jmp     0x5555555524e <f+261>
0x55555555249 <f+256>    mov     eax,0xffffffff
0x5555555524e <f+261>    pop     rbp
0x5555555524f <f+262>    ret
0x55555555250 <main>    endbr64
0x55555555254 <main+4>    push    rbp
0x55555555255 <main+5>    mov     rbp,rsi
0x55555555258 <main+8>    sub     rsp,0x20
0x5555555525c <main+12>   mov     DWORD PTR [rbp-0x14],edi
0x5555555525f <main+15>   mov     QWORD PTR [rbp-0x20],rsi
0x55555555263 <main+19>   cmp     DWORD PTR [rbp-0x14],0x1
(gdb) ni
0x0000555555552f3 in main ()
(gdb) ni
0x0000555555552f6 in main ()
(gdb) ni
0x0000555555552fb in main ()
(gdb) ni
0x00005555555530c in main ()
(gdb) ni
0x000055555555311 in main ()
(gdb)
```

I would use `(gdb) ni` to step over instructions. I mainly did this as some of the Linux or glibc functions I could skip over as they are not created by the user and a product of the compiler or operating system. I realized that I had to use another command to set arguments for the function. I realized this because there was actually a comparison that would check if the user had any input. This explained why I got no input with no argument.



```
0x55555555263 <main+19>    cmp     DWORD PTR [rbp-0x14],0x1
0x55555555267 <main+23>    jg      0x55555555273 <main+35>
0x55555555269 <main+25>    mov     eax,0x0
0x5555555526e <main+30>    jmp     0x55555555311 <main+193>
>0x55555555273 <main+35>    mov     rax,QWORD PTR [rbp-0x20]
0x55555555277 <main+39>    add     rax,0x8
0x5555555527b <main+43>    mov     rax,QWORD PTR [rax]
0x5555555527e <main+46>    mov     rdi,rax
```

I used `(gdb) set args hello` to set an argument to something I know would be wrong. This made be able to bypass the cmp check for the necessary amount of arguments. I realized that the next point of investigation for me was the f function call. I saw that the function is called no matter what. I also saw that main would do a comparison to a set value of 0x532 and if it is not equal than it would jump past a bunch of puts() calls. Knowing this I had assumptions that if I bypass that comparison I will be able to view the locked information.

▼ main function code (click to expand)

```
0000000000001250 <main>:
1250:    f3 0f 1e fa    endbr64
1254:    55             push    rbp
1255:    48 89 e5       mov     rbp,rsi
1258:    48 83 ec 20     sub     rsp,0x20
125c:    89 7d ec       mov     DWORD PTR [rbp-0x14],edi
```

```

125f:      48 89 75 e0      mov     QWORD PTR [rbp-0x20],rsi
1263:      83 7d ec 01      cmp     DWORD PTR [rbp-0x14],0x1
1267:      7f 0a              jg      1273 <main+0x23>
1269:      b8 00 00 00 00    mov     eax,0x0
126e:      e9 9e 00 00 00    jmp     1311 <main+0xc1>
1273:      48 8b 45 e0      mov     rax,QWORD PTR [rbp-0x20]
1277:      48 83 c0 08      add     rax,0x8
127b:      48 8b 00          mov     rax,QWORD PTR [rax]
127e:      48 89 c7          mov     rdi,rax
1281:      e8 c3 fe ff ff    call    1149 <f>
1286:      89 45 fc          mov     DWORD PTR [rbp-0x4],eax
1289:      81 7d fc 32 05 00 00 cmp     DWORD PTR [rbp-0x4],0x532
1290:      75 6b              jne     12fd <main+0xad>
1292:      48 8d 05 6f 0d 00 00 lea     rax,[rip+0xd6f]          # 2008
<_IO_stdin_used+0x8>
1299:      48 89 c7          mov     rdi,rax
129c:      e8 af fd ff ff    call    1050 <puts@plt>
12a1:      48 8d 05 78 0d 00 00 lea     rax,[rip+0xd78]          # 2020
<_IO_stdin_used+0x20>
12a8:      48 89 c7          mov     rdi,rax
12ab:      e8 a0 fd ff ff    call    1050 <puts@plt>
12b0:      48 8d 05 b1 0d 00 00 lea     rax,[rip+0xdb1]          # 2068
<_IO_stdin_used+0x68>
12b7:      48 89 c7          mov     rdi,rax
12ba:      e8 91 fd ff ff    call    1050 <puts@plt>
12bf:      48 8d 05 ea 0d 00 00 lea     rax,[rip+0xdea]          # 20b0
<_IO_stdin_used+0xb0>
12c6:      48 89 c7          mov     rdi,rax
12c9:      e8 82 fd ff ff    call    1050 <puts@plt>
12ce:      48 8d 05 1b 0e 00 00 lea     rax,[rip+0xe1b]          # 20f0
<_IO_stdin_used+0xf0>
12d5:      48 89 c7          mov     rdi,rax
12d8:      e8 73 fd ff ff    call    1050 <puts@plt>
12dd:      48 8d 05 4c 0e 00 00 lea     rax,[rip+0xe4c]          # 2130
<_IO_stdin_used+0x130>
12e4:      48 89 c7          mov     rdi,rax
12e7:      e8 64 fd ff ff    call    1050 <puts@plt>
12ec:      48 8d 05 8d 0e 00 00 lea     rax,[rip+0xe8d]          # 2180
<_IO_stdin_used+0x180>
12f3:      48 89 c7          mov     rdi,rax
12f6:      e8 55 fd ff ff    call    1050 <puts@plt>
12fb:      eb 0f              jmp     130c <main+0xbc>
12fd:      48 8d 05 bd 0e 00 00 lea     rax,[rip+0xebd]          # 21c1

```

```

<_IO_stdin_used+0x1c1>
1304:      48 89 c7                mov     rdi, rax
1307:      e8 44 fd ff ff         call    1050 <puts@plt>
130c:      b8 00 00 00 00         mov     eax, 0x0
1311:      c9                     leave
1312:      c3                     ret

```

When I was stepping through the `f` function I could see a lot of interesting information. I saw calls such as `shl`, `shr`, `imul` which were either math or bit manipulation calls. This made me think that the function could be taking the user's argument and performing some encryption on it and then checking if it matches any hardcoded value that I could find. I did have some suspicions. I noticed that there were many lines like `MOV eax, 0xffffffff` that were called right before the function left. I did not think much of this until I saw that the next instruction in `main` when we leave the program sets the value of `eax` into the `DWORD PTR`. This made me think that the `f` function could actually be an attempt to generate nonsense code to distract the reverse engineer. I am not sure if there even is a specific passphrase that would give you the answer. By running `(gdb) break *f+167` I was able to set the breakpoint at a specific address. I used different variations of this technique.

```

0x55555551d0 <f+135> sub     eax, ecx
0x55555551d2 <f+137> cmp     eax, 0x4
0x55555551d5 <f+140> je      0x55555551f2 <f+169>
0x55555551d7 <f+142> mov     eax, DWORD PTR [rbp-0x8]
0x55555551da <f+145> movsxd  rdx, eax
0x55555551dd <f+148> mov     rax, QWORD PTR [rbp-0x18]
0x55555551e1 <f+152> add     rax, rdx
0x55555551e4 <f+155> movzx   eax, BYTE PTR [rax]
0x55555551e7 <f+158> cmp     al, 0x2d
0x55555551e9 <f+160> jne     0x55555551f2 <f+169>
B+>0x55555551eb <f+162> mov     eax, 0xffffffff
0x55555551f0 <f+167> jmp     0x555555524e <f+261>
0x55555551f2 <f+169> mov     eax, DWORD PTR [rbp-0x8]
0x55555551f5 <f+172> movsxd  rdx, eax
0x55555551f8 <f+175> mov     rax, QWORD PTR [rbp-0x18]
0x55555551fc <f+179> add     rax, rdx
0x55555551ff <f+182> movzx   eax, BYTE PTR [rax]
0x5555555202 <f+185> cmp     al, 0x2d
0x5555555204 <f+187> je      0x555555521e <f+213>
0x5555555206 <f+189> mov     eax, DWORD PTR [rbp-0x8]

```

Multi-thre Thread 0x7ffff7fab7 In: f L?? PC: 0x55555551eb  
(gdb) break \*main+54  
Breakpoint 4 at 0x5555555286  
(gdb) c  
Continuing.  
Breakpoint 2, 0x00005555555151 in f ()  
(gdb) c  
Continuing.  
Breakpoint 3, 0x000055555551eb in f ()  
(gdb)

▼ `f` function code (click to expand)

```

0000000000001149 <f>:
1149:      f3 0f 1e fa            endbr64
114d:      55                     push    rbp
114e:      48 89 e5                mov     rbp, rsp
1151:      48 89 7d e8             mov     QWORD PTR [rbp-0x18], rdi
1155:      c7 45 f4 00 00 00 00    mov     DWORD PTR [rbp-0xc], 0x0
115c:      c7 45 fc 00 00 00 00    mov     DWORD PTR [rbp-0x4], 0x0
1163:      c7 45 f8 00 00 00 00    mov     DWORD PTR [rbp-0x8], 0x0

```

116a:	e9 b7 00 00 00	jmp	1226 <f+0xdd>
116f:	8b 55 f8	mov	edx,DWORD PTR [rbp-0x8]
1172:	48 63 c2	movsxd	rax,edx
1175:	48 69 c0 67 66 66 66	imul	rax,rax,0x66666667
117c:	48 c1 e8 20	shr	rax,0x20
1180:	d1 f8	sar	eax,1
1182:	89 d1	mov	ecx,edx
1184:	c1 f9 1f	sar	ecx,0x1f
1187:	29 c8	sub	eax,ecx
1189:	89 c1	mov	ecx,eax
118b:	c1 e1 02	shl	ecx,0x2
118e:	01 c1	add	ecx,eax
1190:	89 d0	mov	eax,edx
1192:	29 c8	sub	eax,ecx
1194:	83 f8 04	cmp	eax,0x4
1197:	75 14	jne	11ad <f+0x64>
1199:	8b 45 f8	mov	eax,DWORD PTR [rbp-0x8]
119c:	48 63 d0	movsxd	rdx,eax
119f:	48 8b 45 e8	mov	rax,QWORD PTR [rbp-0x18]
11a3:	48 01 d0	add	rax,rdx
11a6:	0f b6 00	movzx	eax,BYTE PTR [rax]
11a9:	3c 2d	cmp	al,0x2d
11ab:	75 3e	jne	11eb <f+0xa2>
11ad:	8b 55 f8	mov	edx,DWORD PTR [rbp-0x8]
11b0:	48 63 c2	movsxd	rax,edx
11b3:	48 69 c0 67 66 66 66	imul	rax,rax,0x66666667
11ba:	48 c1 e8 20	shr	rax,0x20
11be:	d1 f8	sar	eax,1
11c0:	89 d1	mov	ecx,edx
11c2:	c1 f9 1f	sar	ecx,0x1f
11c5:	29 c8	sub	eax,ecx
11c7:	89 c1	mov	ecx,eax
11c9:	c1 e1 02	shl	ecx,0x2
11cc:	01 c1	add	ecx,eax
11ce:	89 d0	mov	eax,edx
11d0:	29 c8	sub	eax,ecx
11d2:	83 f8 04	cmp	eax,0x4
11d5:	74 1b	je	11f2 <f+0xa9>
11d7:	8b 45 f8	mov	eax,DWORD PTR [rbp-0x8]
11da:	48 63 d0	movsxd	rdx,eax
11dd:	48 8b 45 e8	mov	rax,QWORD PTR [rbp-0x18]
11e1:	48 01 d0	add	rax,rdx
11e4:	0f b6 00	movzx	eax,BYTE PTR [rax]

```

11e7:      3c 2d      cmp     al,0x2d
11e9:      75 07      jne     11f2 <f+0xa9>
11eb:      b8 ff ff ff ff  mov     eax,0xffffffff
11f0:      eb 5c      jmp     124e <f+0x105>
11f2:      8b 45 f8      mov     eax,DWORD PTR [rbp-0x8]
11f5:      48 63 d0      movsxd  rdx,eax
11f8:      48 8b 45 e8      mov     rax,QWORD PTR [rbp-0x18]
11fc:      48 01 d0      add     rax,rdx
11ff:      0f b6 00      movzx   eax,BYTE PTR [rax]
1202:      3c 2d      cmp     al,0x2d
1204:      74 18      je      121e <f+0xd5>
1206:      8b 45 f8      mov     eax,DWORD PTR [rbp-0x8]
1209:      48 63 d0      movsxd  rdx,eax
120c:      48 8b 45 e8      mov     rax,QWORD PTR [rbp-0x18]
1210:      48 01 d0      add     rax,rdx
1213:      0f b6 00      movzx   eax,BYTE PTR [rax]
1216:      0f be c0      movsx   eax,al
1219:      01 45 f4      add     DWORD PTR [rbp-0xc],eax
121c:      eb 04      jmp     1222 <f+0xd9>
121e:      83 45 fc 01      add     DWORD PTR [rbp-0x4],0x1
1222:      83 45 f8 01      add     DWORD PTR [rbp-0x8],0x1
1226:      8b 45 f8      mov     eax,DWORD PTR [rbp-0x8]
1229:      48 63 d0      movsxd  rdx,eax
122c:      48 8b 45 e8      mov     rax,QWORD PTR [rbp-0x18]
1230:      48 01 d0      add     rax,rdx
1233:      0f b6 00      movzx   eax,BYTE PTR [rax]
1236:      84 c0      test    al,al
1238:      0f 85 31 ff ff ff  jne     116f <f+0x26>
123e:      83 7d fc 04      cmp     DWORD PTR [rbp-0x4],0x4
1242:      75 05      jne     1249 <f+0x100>
1244:      8b 45 f4      mov     eax,DWORD PTR [rbp-0xc]
1247:      eb 05      jmp     124e <f+0x105>
1249:      b8 ff ff ff ff  mov     eax,0xffffffff
124e:      5d      pop     rbp
124f:      c3      ret

```

Right before I was planning on trying to reverse the 0x532 through the *f* function I found a huge vulnerability. I noticed that the `mov DWORD PTR [rbp-0x4], eax` instruction allows me to set the pointer to the contents of `eax` which I could assign using `gdb`. This meant I could assign the value of `eax` to 0x532 and it would be placed into the pointer. I then ran `(gdb) set $eax = 0x532`. Once I ran that I stepped through the function and got the access granted screen and a print out of the VCU logo.



```
Apr 15 16:00
evan@evan-1-2: ~/Documents/REAssignment3/RE-VCU-Projects

0x5555555525f <main+15>    mov     QWORD PTR [rbp-0x20],rsi
0x55555555263 <main+19>    cmp     DWORD PTR [rbp-0x14],0x1
0x55555555267 <main+23>    jg      0x55555555273 <main+35>
0x55555555269 <main+25>    mov     eax,0x0
0x5555555526e <main+30>    jmp     0x55555555311 <main+193>
0x55555555273 <main+35>    mov     rax,QWORD PTR [rbp-0x20]
0x55555555277 <main+39>    add     rax,0x8
0x5555555527b <main+43>    mov     rax,QWORD PTR [rax]
0x5555555527e <main+46>    mov     rdi,rax
0x55555555281 <main+49>    call    0x55555555149 <f>
B+>0x55555555286 <main+54>    mov     DWORD PTR [rbp-0x4],eax
0x55555555289 <main+57>    cmp     DWORD PTR [rbp-0x4],0x532
0x55555555290 <main+64>    jne     0x555555552fd <main+173>
0x55555555292 <main+66>    lea     rax,[rip+0xd6f]      # 0x555555556008
0x55555555299 <main+73>    mov     rdi,rax
0x5555555529c <main+76>    call    0x55555555050 <puts@plt>
0x555555552a1 <main+81>    lea     rax,[rip+0xd78]      # 0x555555556020
0x555555552a8 <main+88>    mov     rdi,rax
0x555555552ab <main+91>    call    0x55555555050 <puts@plt>
0x555555552b0 <main+96>    lea     rax,[rip+0xdb1]      # 0x555555556068

multi-thre Thread 0x7ffff7fab7 In: main L?? PC: 0x55555555286

Breakpoint 2, 0x000055555555151 in f ()
(gdb) c
Continuing.

Breakpoint 3, 0x0000555555551eb in f ()
(gdb) c
Continuing.

Breakpoint 4, 0x000055555555286 in main ()
(gdb)
```

```
Apr 15 14:58
evan@evan-1-2: ~/Documents/REAssignment3/RE-VCU-Projects

0x55555555263 <main+19>    cmp     DWORD PTR [rbp-0x14],0x1
0x55555555267 <main+23>    jg      0x55555555273 <main+35>
0x55555555269 <main+25>    mov     eax,0x0
0x5555555526e <main+30>    jmp     0x55555555311 <main+193>
0x55555555273 <main+35>    mov     rax,QWORD PTR [rbp-0x20]
0x55555555277 <main+39>    add     rax,0x8
0x5555555527b <main+43>    mov     rax,QWORD PTR [rax]
0x5555555527e <main+46>    mov     rdi,rax
0x55555555281 <main+49>    call    0x55555555149 <f>
0x55555555286 <main+54>    mov     DWORD PTR [rbp-0x4],eax
0x55555555289 <main+57>    cmp     DWORD PTR [rbp-0x4],0x532
0x55555555290 <main+64>    jne     0x555555552fd <main+173>
0x55555555292 <main+66>    lea     rax,[rip+0xd6f]      # 0x555555556008
0x55555555299 <main+73>    mov     rdi,rax
>0x5555555529c <main+76>    call    0x55555555050 <puts@plt>
0x555555552a1 <main+81>    lea     rax,[rip+0xd78]      # 0x555555556020
0x5555555529c <main+76>    call    0x55555555050 <puts@plt>
0x555555552a1 <main+81>    lea     rax,[rip+0xd78]      # 0x555555556020
>0x555555552a8 <main+88>    mov     rdi,rax
multi-thre Thread 0x7ffff7fab7 In: __GI__IO_puts L35 PC: 0x7ffff7c83634
multi-thre Thread 0x7ffff7fab7 In: main L?? PC: 0x5555555529c
0x000055555555289 in main ()
(gdb) ni
(gdb) ni
0x000055555555299 in main ()
(gdb) ni
0x00005555555529c in main ()
(gdb) ni
Access granted.
0x0000555555552a1 in main ()
(gdb) ni
0x0000555555552a8 in main ()
(gdb)
```



## Impact

Having such a vulnerability is very detrimental. Customers are not as secure as they think. Having a data leak would make customers lose trust in us as a company and not continue to buy our products.



With a huge data leak our reputation could be ruined and prevent further contracts coming to our company. Having a user be able to access any forbidden information in such a quick way is very dangerous. I think using methods of obfuscation and encryption would make it much more secure. I think if the  $f$  function was used to confuse the reverse engineer that is a step in the write direction of anti reversing techniques. I think if we also did something similar to the  $f$  function by using a encryption algorithm to create a passcode that it would also make it harder to crack. Even if this software doesn't have a pass phrase, with out methods of anti-reversing I was easily able to get through the programs password check.