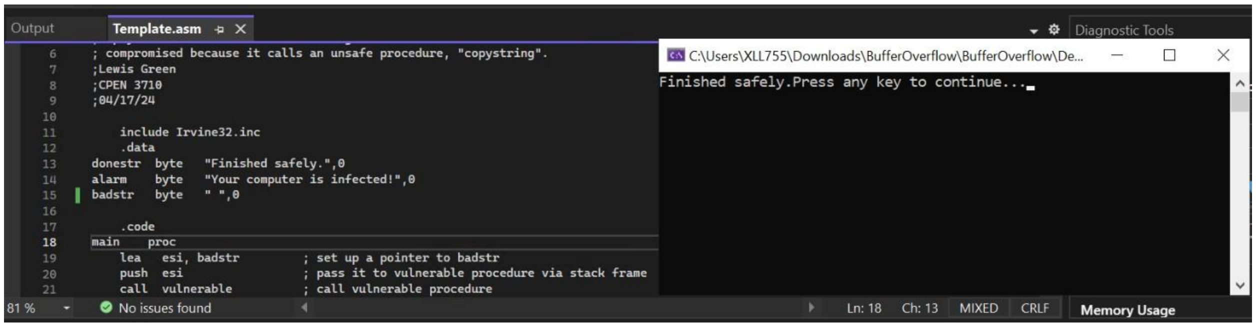
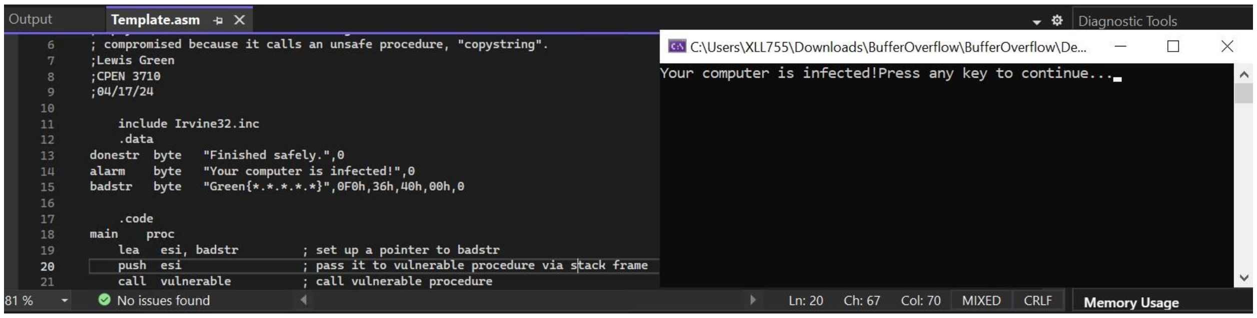
Lab 12



```
; compromised because it calls an unsafe procedure, "copystring".
;Lewis Green
;CPEN 3710
;04/17/24

    include Irvine32.inc
    .data
donestr  byte   "Finished safely.",0
alarm    byte   "Your computer is infected!",0
badstr   byte   " ",0

    .code
main    proc
    lea   esi, badstr         ; set up a pointer to badstr
    push  esi                 ; pass it to vulnerable procedure via stack frame
    call  vulnerable          ; call vulnerable procedure
```

C:\Users\XLL755\Downloads\BufferOverflow\BufferOverflow\De...

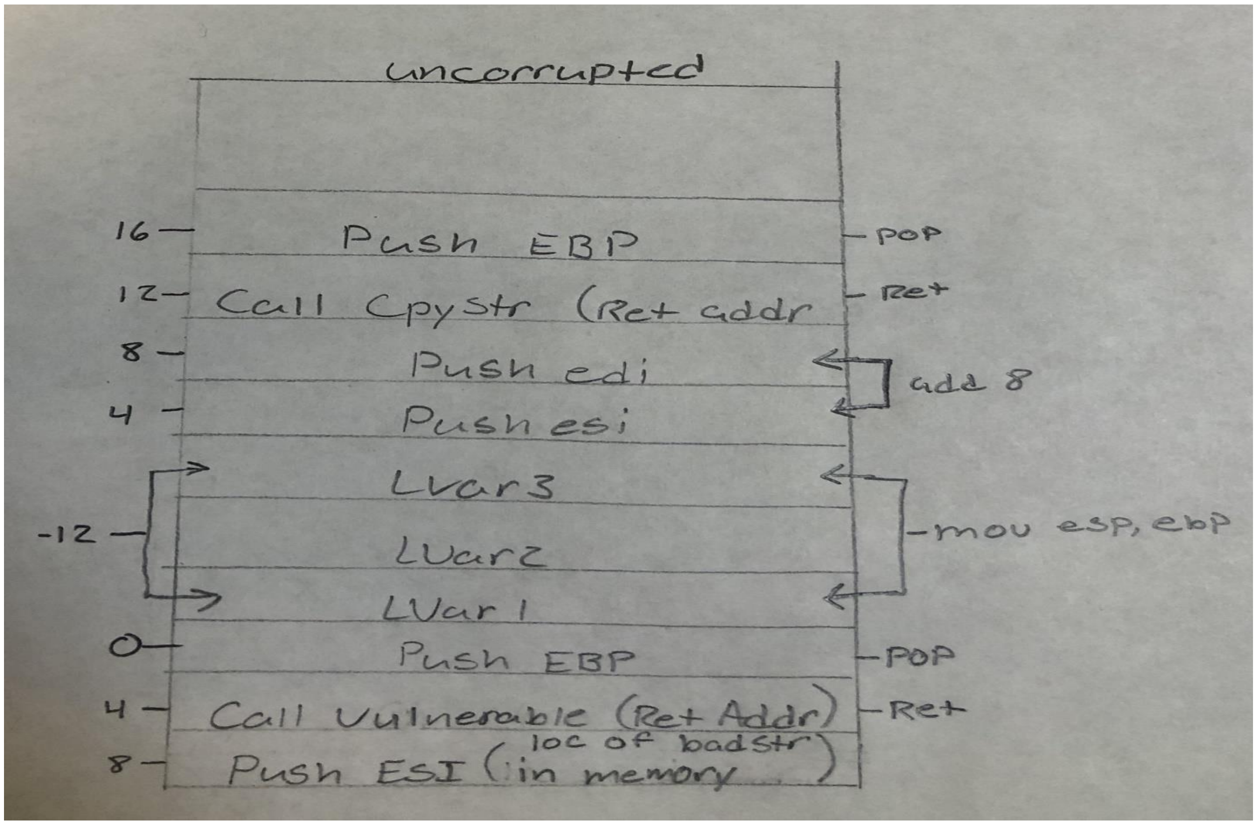
```
Finished safely.Press any key to continue...
```



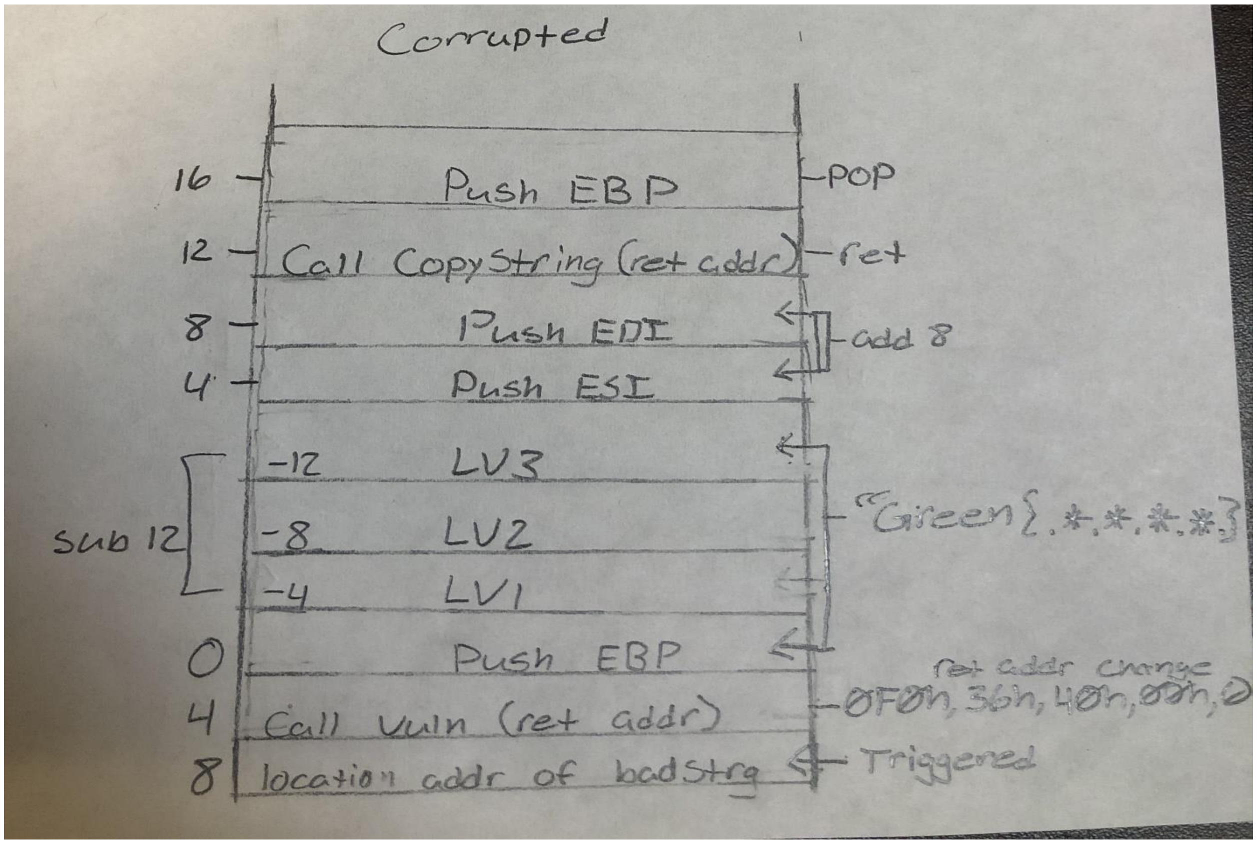
```
; compromised because it calls an unsafe procedure, "copystring".
;Lewis Green
;CPEN 3710
;04/17/24

    include Irvine32.inc
    .data
donestr  byte   "Finished safely.",0
alarm    byte   "Your computer is infected!",0
badstr   byte   "Green{*.*.*.*.*}",0F0h,36h,40h,00h,0

    .code
main    proc
    lea   esi, badstr         ; set up a pointer to badstr
    push  esi                 ; pass it to vulnerable procedure via stack frame
    call  vulnerable          ; call vulnerable procedure
```

C:\Users\XLL755\Downloads\BufferOverflow\BufferOverflow\De...

```
Your computer is infected!Press any key to continue...
```



From the screenshot above, we can see that I drew a stack diagram as I stepped through the program. The first thing that occurs at the start of the program is badstr's location in memory is being loaded into ESI then pushed onto the stack. As we progress into the program, the procedure Vulnerable is called, which pushes the main procedure's return address onto the stack. We push EBP, subtract by 12 to create 3 local variables, push ESI and EDI, and then calls another procedure called CopyString. It pushes EBP, performs a copy, pops EBP, adds 8 to ESP to remove EDI and ESI from the stack, restores the stack pointer, pops EBP, and then returns to the main procedure. This returns the expected result, "Finished Safely.Press any key to continue…" Knowing this and having the source code, we can manipulate the program.



In this stack diagram, nothing is being changed except the characters input into badstr, a location in memory added afterwards, and the null string being placed at the end, so it doesn't go further in memory than we want it. What we know is badstr is located at 00000060 when we look at the .lst file. We also know that the 00403690 is where the instruction pointer is pointing toward at the very beginning of the program when we start debugging it. We need to add them together so we can get the location in memory to return our corrupted string which is 004036F0. Now that we know that we also have to take into consideration how the stack is “cleaned up.” We are allotted 3 local variables which are dwords. Since badstr is a byte, we will need 12 characters to fill that space, but we cannot stop there. If we leave it at 12 characters, it will continue to process the program correctly, so we have to account for the pop for EBP at the end. In order to bypass it and corrupt the pointer, we need to add 4 additional characters which gets us pointing to the return address. Now, we have 004036F0 as the location in memory, but we need to put it in little-endian order. So badstr becomes “Green{*.*.*.*.*}”, 0F0h,36h,40h,00h,0. This causes a buffer overflow because the string currently contains more than expected, causing it to “bleed” over into memory. Now, with the new return address at the end of it, we have successfully corrupted the stack because it will now return to our desired location instead of 004