Shachi Shah
Course: CSE543 Spring B
15th April 2025

Finding Crashes

(1) Challenge 0.0

Code:

```
echo A > input.txt
/challenge/run input.txt
```

Explanation:

The source calls getchar() then immediately does puts((char*)(1)); which dereferences address 0x1. Any input causes a segmentation fault, so by simply providing "A" you crash the program and earn the flag.

(2) Challenge 1.0

Code:

```
echo -n S > input.txt
/challenge/run input.txt
```

Explanation:

The program reads one character. If that character is ASCII 0x53 ('S'), it executes puts((char*)0x53); which causes a segfault by dereferencing address 0x53. Thus, inputting "S" crashes the binary and returns the flag.

(3) Challenge 2.0

Code:

```
echo -e "-1059140866\n0" > input.txt
challenge/run input.txt
```

Explanation:

The program reads an integer and an unsigned long long. It compares the integer with 0xc0decafe. Because scanf("%d") reads a signed 32‑bit value, 0xc0decafe becomes –1059140866. Providing –1059140866 meets the condition, and passing "0" for the target causes the function pointer call to dereference a null pointer (crashing the program) and awarding the flag.

(4) Challenge 2.1

Code:

```
echo -e "-1059140866\n4198742" > input.txt
/challenge/run input.txt
```

Explanation:

Using the same condition as in 2.0 (–1059140866 equals 0xc0decafe when read as a signed int), you now supply a valid target. Here 0x401156 (the address of win()) converts to 4198742 in decimal. The program then calls win(), which prints "Great Job" and gives you the flag.

(5) Challenge 3.0

Code:

```
echo -e "-1597107401\n0" > input.txt
/challenge/run input.txt
```

Explanation:

The decompiled code shows a check against 0xa0ce1337. Converting 0xa0ce1337 to a signed 32-bit integer gives –1597107401. Supplying that value meets the condition; then passing "0" as the target makes the function pointer call dereference a null pointer, crashing the program and awarding the flag.

(6) Challenge 3.1

Code:

```
echo -e "-1597107401\n4198742" > input.txt
/challenge/run input.txt
```

Explanation:

Like 3.0, the program checks if the supplied integer equals 0xa0ce1337 (–1597107401 when read as signed). Instead of crashing by passing "0", you supply the address of win() (0x401156, which is 4198742 in decimal). This hijacks control flow so that the program jumps to win(), prints "Great Job" and returns the flag.

(7) Challenge 4.0

Code:

```
python3 -c 'print("A"*1100)' > input.txt
/challenge/run input.txt
```

Explanation:

The program reads a string into a 40-byte buffer with scanf("%s", buffer) and then prints it with printf("You sent: %s\n", buffer);. By supplying an input of around 1100 "A"s, you overflow the buffer so that printf() attempts to read beyond allocated memory, causing a segmentation fault and awarding the flag.

(8) Challenge 5.0

Code:

```
python3 -c 'import struct; print("A"*48 +
struct.pack("<Q", 0x401156).decode("latin1"))' > input.txt
/challenge/run input.txt
```

Explanation:

Here the goal is to hijack execution to call win() (which prints "Great Job"). The input overflows the 40-byte buffer and 8 extra bytes (to cover saved RBP), then overwrites the return address with the address of win() (0x401156). In a normal scenario this would redirect control flow; however, in testing this payload did not yield the flag—further work (such as targeting exit handler pointers) is needed to bypass the exit(1) call.

(9) Challenge 5.1

Code:

```
python3 -c 'print("A" * 100)' > input.txt
```

```
/challenge/run input.txt
```

Explanation:

The program uses strcpy() to copy user input into a local buffer of only 40 bytes. Since strcpy() doesn't check for bounds, providing more than 40 bytes overflows the buffer and overwrites saved registers on the stack. This causes a segmentation fault (crash) when the function tries to return, which is exactly what the challenge checks for. Sending 100 “A”s ensures a reliable crash and triggers the flag.

(10) Challenge 8.0:

Code:

```
echo -n
"asdkjfakjsldfhalsdfhasjhfdkajsdhflkajshdfkhasldkjhalskdjf
h" > input.txt
/challenge/run input.txt
```

Explanation:

The program reads up to 256 bytes from stdin, then checks a CRC32 over input[6:] and, if valid, copies sVar1 bytes (from the first two input bytes) into a malloced buffer sized (input_len - 6). If sVar1 is larger than the allocated size, this causes a heap buffer overflow and crashes the program. By crafting input that passes the CRC check and sets sVar1 to a large value (e.g., 0x7300), I triggered the crash.