1. Table

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **offset** | | | | | **total size** | **alignment requirement** | | |
|  | a | b | c | d |  | | |  |
| P1 | 0 | 4 | 8 | 12 | 16 | | | 4 |
| P2 | 0 | 8 | 12 | 16 | 24 | | | 8 |
| P3 | 0 | 2 |  |  | 12 | | | 2 |
| P4 | 0 | 32 |  |  | 40 | | | 8 |
| P5 | 0 | 24 |  |  | 64 | | | 8 |

1. Table

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **no stack protector** | **stack protector** |
| gcc flag |  | -fno-stack-protector | -f-stack-protector | |
| len | assembly for allocating stack | subq $16, %rsp | subq $16, %rsp | |
| stack size in decimal | 16 | 16 | |
| assembly for freeing stack | addq $16, %rsp | addq $16, %rsp | |
|  |  |  |  | |
| lptoa | assembly for allocating stack | Subq $16, %rsp | Subq $16, %rsp | |
| stack size in decimal | 16 bytes | 16 bytes | |
| assembly for freeing stack | addq $16, %rsp | addq $16, %rsp | |
| "char \*s" address relative to rsp after entering lptoa | $32(%rsp) | $32(%rsp) | |
| "long \*p" address relative to rsp after entering lptoa | $40(%rsp) | $40(%rsp) | |
| "val" address relative to rsp after entering lptoa | $8(%rsp) | $8(%rsp) | |
|  |  |  |  | |
| longlen | assembly for allocating stack | Subq $32, %rsp | Subq $48, %rsp | |
| stack size in decimal | 32 bytes | 48 bytes | |
| assembly for freeing stack | Addq $32, %rsp | Addq $48, %rsp | |
| "x" address relative to rsp after entering longlen | $32(%rsp) | $48(%rsp) | |
| "v" address relative to rsp after entering longlen | 8(%rsp) | 8(%rsp) | |
| "buf" address relative to rsp after entering longlen | (%rsp) | 16 (%rsp) | |
| canary register name |  | %fs:48 | |
| canary address relative to rsp |  | 24(%rsp) | |
| canary value |  | Random value | |
| assembly for erasing canary value |  | xorq %fs:48, %rax | |
| assembly for canary cross check |  | jne \_\_stack\_chk\_fail | |

1. Open-ended question
   1. So, first we compute the number of bytes needed to hold the pointer to n long \* (pointers). Since it needs to allocate space for pointers, we multiply n by 8 and the extra 30 is added to make sure that even after rounding down to a multiple of 16 we have enough space with padding to hold the pointer to n pointers.
   2. We take s2 and add 15 to then do a bitwise -16 to round down to multiple of 16. This makes sure we are exactly at the byte we need to at and is also 16-byte aligned.
   3. The extra space e₁ equals s₁ minus (p + 8·n). Its size depends on the remainder of (8·n + 30) modulo 16. When n is odd, 8·n leaves a remainder of 8, so 8·n + 30 is 38 (or 6 modulo 16), yielding a minimum e₁ (approximately 16 bytes). When n is even, 8·n is a multiple of 16 so that 8·n + 30 leaves a remainder of 14 modulo 16, resulting in a maximum e₁ (about 24 bytes).
   4. Since s₂ is obtained by subtracting a multiple of 16 from s₁, it maintains the original alignment residue. The subsequent computation for p – adding 15 to s₂ and then rounding down to a multiple of 16 – guarantees that p is exactly 16-byte aligned, satisfying the system’s alignment requirements.