Practice-exam Hacking in C 2019, time: 2h

(100 points total)

1. **(20 points)**

   (a) What should the declarations of `r`, `s` `t`, `w`, and `v` be in the code below to give them the right type?

   ```
   int a;
   int *p = &a;
   char c;
   .... r = &c;
   .... s = &r;
   .... t = **s;
   .... w = &p;
   .... v = *p;
   ```

   (b) What are the values of `x[0]`, `x[1]`, `x[2]`, and `y` after executing the code below?

   ```
   int x[3];
   x[0] = 0;
   x[1] = 8;
   x[2] = 16;
   int y = 3;
   int *p = x;
   int *q = &y;
   int **pp = &p;
   int **qq = &q;
   (*pp)++;
   (*p)++;
   *q = *p+1;
   ```

2. **(20 points)** Consider the following code fragment:

   ```
   int f()
   {
     char a[22];
     int32_t b[3];
     uint64_t c;
     int32_t i;
     ...
   }
   ```

   (a) The local variables of the function `f()` take a total of 46 bytes. What would be a good reason for a compiler to allocate more space that that for the local variables on the stack on a modern 64-bit machine?

   (b) The compiler can also reorder local variables on the stack. In what order would you expect the variables to be stored? Explain your answer.

   (c) Write some code that prints "`expected order`" if the variables are indeed in the order that you described in part b).

(d) Write a program that prints ”`big endian`” when it is compiled for and running on a big-endian architecture and ”`little endian`” when it is compiled for and running on a little-endian architecture.
Note: This program should not be longer than 8 lines of code.

3. **(20 points)**

Consider the following code

```
#include <stdio.h>
#include <stdint.h>

int main() {
  int64_t x[3];
  x[0] = 42;
  x[1] = 1 << 5;
  x[2] = 4 ^ 7;  //   ^ is bitwise XOR

  printf("%lx \n", x);
  printf("%lx \n", &x);          // (b)
  printf("%lx \n", x+2);         // (a)
  printf("%lx \n", &x+2);        // (c)
  printf("%lx \n", *(x+2) ^ 3);  // (d)
  printf("%lx \n", *x + x[2]);   // (e)
  return 0;
}
```

Recall that `%lx` prints a long in hexadecimal notation. If the first call to `printf` prints `7ffffffabc00`, what do the other calls to `printf` print?

4. **(20 points)**

```
int *terrible(){
  char *s1 = malloc(20);
  char *s2 = malloc(30);
  int p[42];
  const char *z = "hello world!";
  s1 = s2;
  free(s1);
  free(s2);
  free(z);
  return p;
}
```

Explain the 5 different errors in this code. One error occurs actually twice, so there are a total of 6 errors.

5. **(20 points)** Consider the following code vulnerable to buffer overflows:

```
int bad(const char *s, size_t len)
{
  buf a[100];
  ...
  memcpy(a, s, len);
  ...
}
```

Assume the role of an attacker who wants to inject and run shell code. Assume that the attacker controls inputs to the function `bad`. Assume that the address of `buf` is `0x7fffed6b7dd0`. Assume that the return address of `bad` is stored at address `0x7fffed6b7e3c`.

(a) What `len` argument would you use in the attack?

(b) Assume that you have 30 bytes of shell code. What other "ingredients" go into the attack string that the attacker will provide as first argument?

(c) Describe in detail what the first argument `s` will look like for the attack to work.

(d) Would the attack still work if `memcpy(a, s, len)` was replaced by `strcpy(a, s)`? Would the attack string need to change? Explain why or why not.

(e) Would a non-executable stack prevent the attack? Explain why or why not.

(f) Would stack canaries prevent the attack? Explain why or why not.