Software Security

– Final Examination (3h, all documents allowed) –

1 Code Security Audit (8 points)

Find at least two security flaws in the following functions : `count_digits()` and `add_record()`.
For each security issue, name the type of flaw, explain how to exploit it and the ways to mitigate it and suggest a way to fix it in the code.

Note that `UINT_MAX` is the maximum value taken by `unsigned int` variables.

```
...
FILE *global_file_handle;

/* Get the number of chars (i.e. digits) needed to represent the given
 * number as an ASCII string */
unsigned int count_digits(unsigned int number) {
  unsigned int n = 0, left = number;
  while (left != 0)
    {
      left = left / 10;
      n++;
    }
  return n;
}

void add_record(const char *name, unsigned int salary) {
  char buffer[256];
  unsigned int len = strlen(name);
  unsigned int num_digits = count_digits(salary);

  /* 5 extra bytes required: ':', ' ', '$', '\n' and final '\0' */
  if (len + num_digits + 5 > UINT_MAX)
    err(EXIT_FAILURE, "error: integer overflow");

  len = len + num_digits + 5;

  if (len > sizeof(buffer))
    err(EXIT_FAILURE, "error: string too long");

  /* Output formatted string to buffer */
  sprintf(buffer, "%s: $%u\n", name, salary);

  /* Write buffer to file */
  fprintf(global_file_handle, buffer);
}
...
```

2 Exploiting a Format String Bug in Solaris CDE (12 points)

Read the article "*Exploiting a Format Sring Bug in Solaris CDE*" by Marco Ivaldi, Phrack #70, 2021. Then, answer the following questions.

Questions

1. fake question to avoid shifting...
2. Tell why the attacked code displayed in the article looks unstructured and has weird variable names.
3. In section 2, spot the two bugs that are mentioned in the text and, for each of them, explain how they work and what result you can expect once you know how to trigger the code.
4. From section 3.2, give the name and the basic usage of the vulnerable program that will be targeted. Also, explain why do we need an access to a configured printer (possibly a fake one) and a graphical server (X11, Xorg, XQuartz or Wayland).
5. Still from section 3.2, explain why and how do we "*fake*" `lpstat` to the vulnerable program.
   Note : `lpstat` is used to gather information about the printers connected to the printing server.
6. From section 3.3, explain the structure of the payload and how we inject it in memory. Explain also why do we need to use `buf2` to avoid getting out of the stack section.
7. From section 3.4, explain how we can get control of the execution flow and execute our shellcode through the format string exploit on an x86 architecture.
8. Still on section 3.4, explain why the previous method is not possible on a SPARC architecture and explain the alternative way proposed by the author. Detail the content of the hostile format string.
9. Section 3.5 is about where the shellcode will be stored. Explain :
   — Why addresses with NULL-bytes are a problem ?
   — Why is it often needed to have `gdb` attached to the process to make it work ?
   — Finally, explain where did land the shellcode and why here ?
10. Section 3.6 is about forging a shellcode, explain what it does and why should it fit into 36 bytes ?
    Note : In SPARC, '`ba`' is a "branch" instruction which is equivalent to a '`jmp`' in x86.
11. Finally, explain what should be done to fix the original program and decrease the attack surface.
12. As a conclusion, try to tell under what conditions it worth spend some time developing an exploit for governmental or criminal usage ? What conditions could be ideally met on the security flaw or on the exploitation technique used ? Explain your answers.