TD 6: Format Strings Vulnerabilities

Get the TD tarball from
`https://dept-info.labri.fr/~thibault/SecuLog/td6.tgz`
unpack it, run `make`
When running the programs, rather use the 32bit versions, they will be simpler to hack.

1 Format Strings

1. What does the following code do? (`printf-1.c`) (guess it before executing it) Tell what the last displayed value is.

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

int main (void) {
  int value;
  fprintf(stdout, "%c 0x%08x %s a\aa %3$s %n\n", 0x41, 9, "AAAA", &value)
      ;
  fprintf(stdout, "value = %d\n", value);
  return EXIT_SUCCESS;
}
```

We will now attack the `vulnerable-1-32` program below. We will start with managing to get information from it, such as addresses and content, and eventually get it to behave differently.

Note that there is also a pie version of the program, `vulnerable-1-32-pie` . PIE means Position-Independent Executable, i.e. the code can be loaded at a random address, which is a usual security feature to make attacks harder. We will thus first play with the non-pie version, and then notice that the same works with the pie version.

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

char passwd[] = "secret_password";

#define MAGIC 0x12434241

int target = MAGIC;

void foo (char *string) {
  printf (string);
  if (target == 0x00025544)
    printf ("you have hacked it!\n");
  else if (target != MAGIC)
    printf ("you have modified the target to 0x%x!\n", target);
```

```
}

int main (void) {
  volatile int var = 0xabcd;
  char buf[128];
  while (1) {
    if (!fgets(buf, sizeof(buf), stdin))
        break;
    foo(buf);
  }
  return EXIT_SUCCESS;
}
```

2. For a start, get the program to print the return address from `foo` into `main`. Note : use gdb, put a breakpoint on printf, run the program, and once gdb breaks in printf, use `pframe` to check what the stack looks like, and compare the content of the stack with what you get with giving the program input like `%p %p %p`... and `%3$p %4$p`...

   Check first that it works with the non-pie version (so it's easy to compare with what `objdump` says) and then check that it works with the pie version as well. This means that we can get pie programs to tell the random position where they were loaded, and buffer overflow attacks such as seen in TD5 become possible.

3. Now get the same program to print the value of the `var` local variable of `main`. You can use the gdb commands `bt` and `frame` to access to the `main` frame.

4. Now get it to print some of the content of `buf`. Note that you can use `pframe/64` to print more of the stack.

   Put an integer in the first 4 bytes of `buf` and use some `%p` to get the program to print it.

   Hint : To pass the exact bytes you want, you can use e.g.

   ```
   echo -e '\x61\x62\x63\x64' | ./vulnerable-1-32
   ```

   which gives it the four bytes 0x61, 0x62, 0x63 and 0x64 (which will be printed by the program as abcd, see `man ascii`)

5. For simplicity, use the non-pie version, and get it to print the content of the `passwd` variable, thanks to some `%s` hack.

   Hint1 : To get the address of the `passwd` string, you can use

   ```
   objdump -D vulnerable-1-32 | grep passwd
   ```

   Hint2 : first make sure that you got the address correctly recorded in `buf`, and that you can print it with some `%p` , and then just switch to `%s`

   Hint3 : You can use `| hexdump -C` to print in hexadecimal the output.

6. Still with the non-pie version, now print the value of the `target` variable (but interpreted as a string, there no other easy way ; use `| hexdump -C` to print it in hexadecimal).

7. Still with the non-pie version, get it to print "`you have modified the target`".

8. Still with the non-pie version, get it to print "`you have hacked it!`".

9. Now retry the last three steps, but with PIE enabled (and ASLR left enabled). Remember step 2 to print some address inside `main`, and explain why that's helpful to access `target`.

   Note : you will thus need, during the same execution of vulnerable-1-32-pie, to get the information where the program is running, and then inject the approriate string. You can use something like this :

   ```
   (echo -e 'foobar' ; read hack ; echo -e "$hack") | ./vulnerable-1-32-pie
   ```

and use `\\x01\\x02...` to get the characters you want through. Why do we need two `\` characters ?

Note2 : the pie and the non-PIE builds are different since PIE requires different kinds of x86 instructions.

10. Make the program execute a shell... by making it call the `system` libc function with `"/bin/sh"` as parameter. Hint : to write large values to memory, you can write the 32 bits in two steps : first write a 32bit value containing the lower 16 bits, and then write two bytes after that a 32bit value containing the higher 16 bits. Note : try first with ASLR disabled :) It happens that the position of the libc allows to perform the four writes in the right order to easily make `%n` write the wanted values.
11. Now repeat the exploit with ASLR enabled. Most often the position of libc will allow to perform the four writes in the right order too.

2 Level up

Read the "GOT overwrite" section of the "Exploiting Format String Vulnerabilities" article linked from the course website. For this exercise we will replace the address of the `exit()` function by the address of the `bar()` in the global offset table (GOT), first with PIE disabled. By the end, we should trigger the message `"code execution redirected! you win!"`.

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

void bar (void) {
  printf ("code execution redirected! you win!\n");
  _exit (EXIT_SUCCESS);
}

void foo (void) {
  char buffer[512];

  fgets (buffer, sizeof (buffer), stdin);
  printf (buffer);
  exit (EXIT_SUCCESS);
}

int main (void) {
  foo ();
  return EXIT_SUCCESS;
}
```

3 Advanced

This exercise is about a remote attack based on a format string vulnerability on a distant server (`vulnerable-5.c`).
You can connect to it with `nc localhost 4242`
We first suppose to have PIE disabled, hack it!

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

```c
#include <string.h>
#include <unistd.h>

#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>

#define LISTEN_PORT 4242
#define LENGTH 1024

int cnxfd = 0;

void fail(char *msg) {
  perror(msg);
  exit(EXIT_FAILURE);
}

void bindshell(void) {
  /* Binding the process to the connexion */
  dup2(cnxfd, STDIN_FILENO);
  dup2(cnxfd, STDOUT_FILENO);
  dup2(cnxfd, STDERR_FILENO);
  /* Running the shell */
  system("/bin/sh");
}

void server(int sockfd) {
  char input[LENGTH], output[LENGTH];

  while (true) {
    struct sockaddr_in caddr;
    socklen_t clen = sizeof(caddr);
    if ((cnxfd = accept(sockfd, (struct sockaddr *) &caddr, &clen)) < 0)
      fail("accept()");
    /* Blanking memory */
    memset(input, '\0', LENGTH);
    memset(output, '\0', LENGTH);
    /* Receiving and sending back the string */
    send(cnxfd, "Waiting for data: ", 18, 0);
    recv(cnxfd, input, LENGTH - 1, 0);
    snprintf(output, sizeof(output), input);
    output[sizeof(output) - 1] = '\0';
    send(cnxfd, "Sending data: ", 14, 0);
    send(cnxfd, output, strlen(output), 0);
    /* Closing the connection */
    close(cnxfd);
  }
}

int main(void) {
```

```
    int sockfd;
    const struct sockaddr_in saddr = {
      .sin_family = AF_INET,
      .sin_addr.s_addr = htonl(INADDR_ANY),
      .sin_port = htons(LISTEN_PORT)
    };

  while (true) {
    if (!fork()) { /* Child process */
      fprintf(stdout, "run (pid = %d)\n", getpid());
      /* Socket initialization and error checking */
      if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
        fail("socket()");
      if (setsockopt(sockfd,
                     SOL_SOCKET, SO_REUSEADDR, &(int){ 1 }, sizeof(int)) <
                        0)
        fail("setsockopt()");
      if (bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
        fail("bind()");
      if (listen(sockfd, LENGTH) < 0)
        fail("listen()");
      /* Running the server */
      server(sockfd);
    } else { /* Parent process (wait for child) */
      wait(NULL);
      close(sockfd);
    }
  }
  return EXIT_SUCCESS;
}
```

Try, now, to attack the same server with PIE enabled (`vulnerable-5-xx-pie`).