Hack The Boo 2025

Halloween CTF hosted by Hack The Box.

https://ctf.hackthebox.com/event/details/hack-the-boo-2025-competition-2842

- Rusted Oracle

Rusted Oracle

Written by `@mxg586` (Misha), also hosted on my blog

This rev (reverse engineering) challenge consists of an `x86_64` executable file called `rusted_oracle`

Here's the output when you run it (`INPUT` is when the program asks for input):

```
A forgotten machine still ticks beneath the stones.
Its gears grind against centuries of rust.

[ a stranger approaches, and the machine asks for their name ]
> INPUT
[ the machine falls silent ]
```

The above output shows what happens if you run the program with the wrong input, so we need to find the correct input.

When you open the program up in Ghidra, it is fairly straightforward to get the correct input for the name, you can tell from this line in the decompiled code of the main function:

```
iVar1 = strcmp(local_58,"Corwin Vell");
```

So we now know the correct name is "Corwin Vell", here's the output of the program when you run it with this input:

```
A forgotten machine still ticks beneath the stones.
Its gears grind against centuries of rust.

[ a stranger approaches, and the machine asks for their name ]
> Corwin Vell
[ the gears begin to turn... slowly... ]
```

Note that the program does not terminate after you press enter and keeps running until you press `ctrl+c`

In the Ghidra decompiled code, you can see that the main function calls a function called `machine_decoding_sequence`. In this function, it first sleeps for a random number of seconds and then runs some really obfuscated code:

```
__seconds = rand();
sleep(__seconds);
for (local_2c = 0; local_2c < 0x17; local_2c = local_2c + 1) {
    // Really obfuscated code
}
```

From looking at the code, you can see that the really obfuscated code probably prints the flag, the problem is that the `rand` function generates a random number between 0 and `RAND_MAX` which is normally at least 32767 so the program is sleeping for a random number of seconds in that range.

The probability of the program sleeping for a reasonable number of seconds is very low, so it is not a good strategy to just rerun the program until a reasonably low random number is selected. It would be useful if you could somehow avoid waiting a long time for the code to run and eventually, I had an idea.

Linux executables are usually dynamically linked and after running the `file` command you can see that this is the case for this file. This means that C functions like `rand` are not built in to the executable but rather are loaded in from your operating system when you run the file. Conveniently, all the standard Linux library functions are open source and you can modify them. So naturally, I did the only reasonable thing you can do and compiled `glibc` from source but with a modified `rand` function.

I downloaded the same version of `glibc` as my system from ftp.gnu.org/gnu/glibc/ and I found the `rand` function in the source code. Here's the function:

```
int
rand (void)
{
  return (int) __random ();
}
```

So you can see it returns the output of a function called `__random` which is an `int`. To solve this challenge, we just want the function to return a low value so I just set the body of the function to `return 1`:

```
int
rand (void)
{
  return 1;
}
```

I then compiled the source code with the prefix set to a directory in my home directory so it doesn't overwrite my actual system `glibc`.

After doing this, you just run the challenge executable like this:

```
LD_LIBRARY_PATH=/home/username/path/to/glibc/lib ./rusted_oracle
```

Here's the output after this:

```
A forgotten machine still ticks beneath the stones.
Its gears grind against centuries of rust.


[ a stranger approaches, and the machine asks for their name ]
> Corwin Vell
[ the gears begin to turn... slowly... ]
On a rusted plate, faint letters reveal themselves: HTB{sk1pP1nG-C4ll$!!1!}
```

After exactly one second after entering "Corwin Vell", the last line appears and we have the flag! It says something about skipping calls, whatever that means, but what matters is that we solved the challenge!

We can draw some valuable CTF advice from this challenge: If the program doesn't do what you want, try recompiling parts of your operating system to make it do what you want.