Pilot Writeup

Author: Yaseen

Description

```
This one’s a Pilot, ‘cause it’s our first time on air!
```

Brief

```
 Exploited a buffer overflow to leak the stack address, calculated our
buffer
 location, and injected shellcode. Used a crafted payload to pivot
execution and
 spawn a shell.
```

Security checks

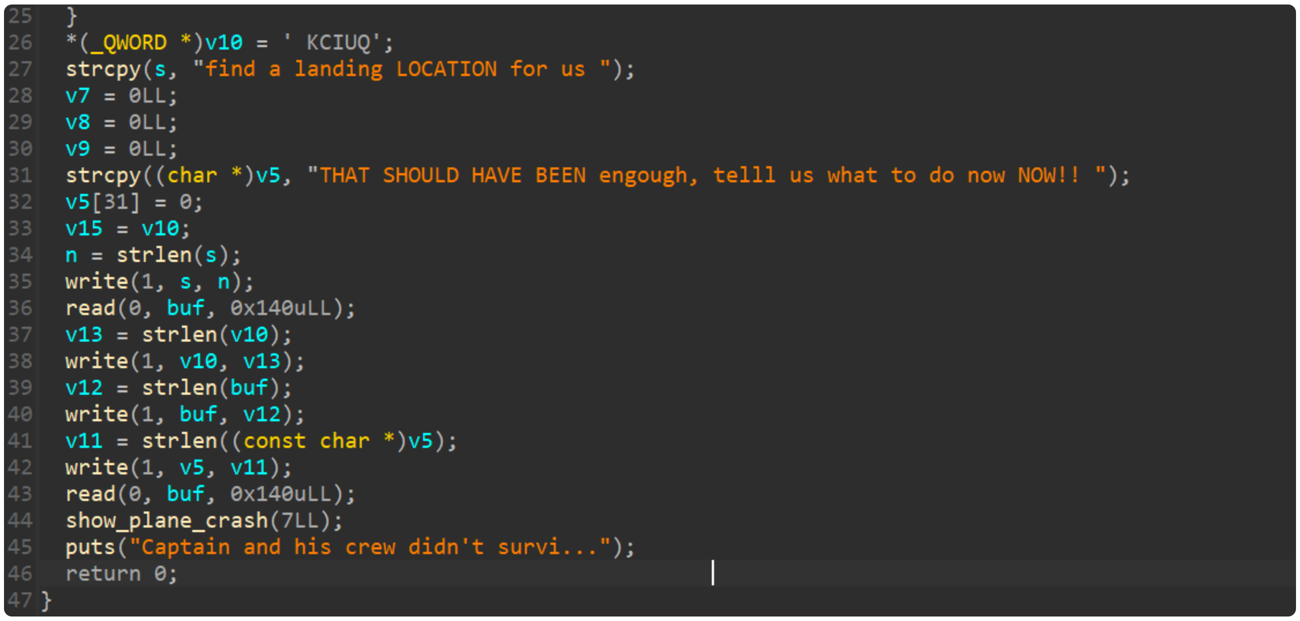
```
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        PIE enabled
    Stack:      Executable
    RWX:        Has RWX segments
    Stripped:   No
```

```
pilot: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=cab72d119e8608da9440bcf4d4281f14ede74bc9, for GNU/Linux
3.2.0, n
ot stripped
```

1. Not stripped
2. Statically Linked
3. Stack executable => We think of shell codes
4. No canary No problem !

- Running the binary, it gives us 2 prompts and it seems it prints our input back, but unfortunately it's not vulnerable to format strings.
- The asci art is being displayed twice

Reversing

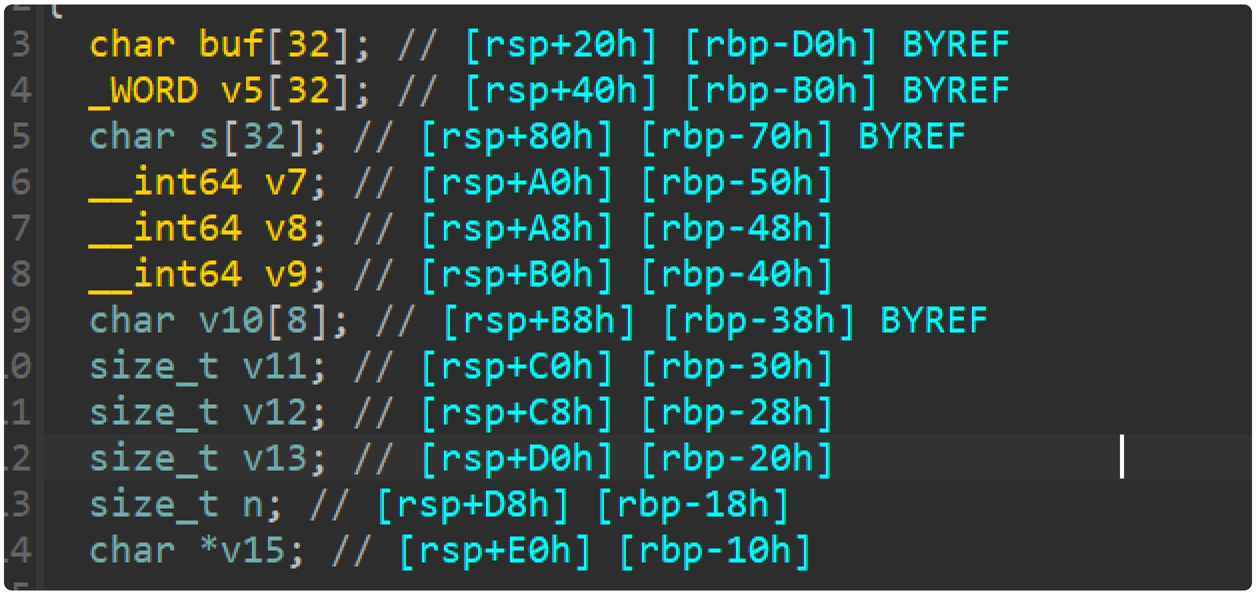


```
  }
  *(_QWORD *)v10 = ' KCIUQ';
  strcpy(s, "find a landing LOCATION for us ");
  v7 = 0LL;
  v8 = 0LL;
  v9 = 0LL;
  strcpy((char *)v5, "THAT SHOULD HAVE BEEN engough, telll us what to do now NOW!! ");
  v5[31] = 0;
  v15 = v10;
  n = strlen(s);
  write(1, s, n);
  read(0, buf, 0x140uLL);
  v13 = strlen(v10);
  write(1, v10, v13);
  v12 = strlen(buf);
  write(1, buf, v12);
  v11 = strlen((const char *)v5);
  write(1, v5, v11);
  read(0, buf, 0x140uLL);
  show_plane_crash(7LL);
  puts("Captain and his crew didn't survi...");
  return 0;
}
```

- The first part of the main is related to asci art printing, so we will ignore it and discuss the codes showed in the image.
- It first copies 2 strings to memory variables and prints each one on each input() we get. In total we have 2 promtps.
- the **buf** we input to is 32 characters max, but it allows reading up to 320 bytes (0x140) => *Buffer overflow*
- It uses **write** function to print, so it's not vulnerable to format strings
- Since we have a buffer overflow and 2 prompts, we conclude that the first prompt is needed for an address leak, and the second one to spawn our shell. Since we weren't given a library nor we know the environment of remote server, as well as the stack being executable we **exclude** `ret2libc` and think of `ret2shellcode`

Leaking

- The code reads 320 bytes into buf, which can overflow and overwrite null terminators of nearby strings like v10 or v5 on the stack. When strlen() is called on these strings, it no longer stops at the original null terminator and counts additional bytes into adjacent memory regions. The write() function then outputs this extended length, potentially leaking sensitive data from memory areas beyond the intended string boundaries. This creates an information disclosure vulnerability where an attacker can read arbitrary stack memory by controlling the input buffer overflow.
- But what do we want to leak and how to calculate the needed offset?



```
char buf[32]; // [rsp+20h] [rbp-D0h] BYREF
_WORD v5[32]; // [rsp+40h] [rbp-B0h] BYREF
char s[32]; // [rsp+80h] [rbp-70h] BYREF
__int64 v7; // [rsp+A0h] [rbp-50h]
__int64 v8; // [rsp+A8h] [rbp-48h]
__int64 v9; // [rsp+B0h] [rbp-40h]
char v10[8]; // [rsp+B8h] [rbp-38h] BYREF
size_t v11; // [rsp+C0h] [rbp-30h]
size_t v12; // [rsp+C8h] [rbp-28h]
size_t v13; // [rsp+D0h] [rbp-20h]
size_t n; // [rsp+D8h] [rbp-18h]
char *v15; // [rsp+E0h] [rbp-10h]
```

ADDRESS (pointer):

v15 - char pointer (address pointing to v10)

These are VALUES stored at addresses:

```
buf[32] - 32-byte buffer stored at address [rbp-D0h]
v5[32] - 64-byte buffer stored at address [rbp-B0h]
s[32] - 32-byte buffer stored at address [rbp-70h]
v7, v8, v9 - 8-byte values stored at addresses [rbp-50h], [rbp-48h],
[rbp-40h]
v10[8] - 8-byte buffer stored at address [rbp-38h]
v11, v12, v13, n - size_t values stored at addresses [rbp-30h] through
[rbp-18h]
```

- so v15 is our target, our input starts at [rbp-D0h] , v15 resides at [rbp-10h]
- Distance: 0xD0 - 0x10 = 0xC0 *(192 in decimal)* to leak *v15
- ret address offset is 216

Check solve.py