COMP 4384 Software Security

Module 5: *Integer Overflow Attacks*

Ahmed Tamrawi

atamrawi   atamrawi.github.io   ahmedtamrawi@gmail.com

Acknowledgment Notice

*Part of the slides are based on content from CMSC414 course by **Dave Levin** and **Niall Cooling**'s blog "When integers go bad" (https://blog.feabhas.com/2014/10/vulnerabilities-in-c-when-integers-go-bad/) and "Basic Integer Overflows" by **Phrack magazine** (http://phrack.org/issues/60/10.html)*

What does the program print?

```java
public class JavaPuzzle {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

What does the program print?

```
public class JavaPuzzle {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

It prints “5”!

What does the program print?

```
public class JavaPuzzle {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24L * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24L * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

Takeaway 1 *When working with large numbers, watch out for* ***overflow****— it's a silent killer!*


999999
P R N D L2 L1
P R N D L2 L1
000000
P R N D L2 L1

What’s Wrong with this Code?

```
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

What’s Wrong with this Code?

int
4 bytes
-2,147,483,648
to
2,147,483,647

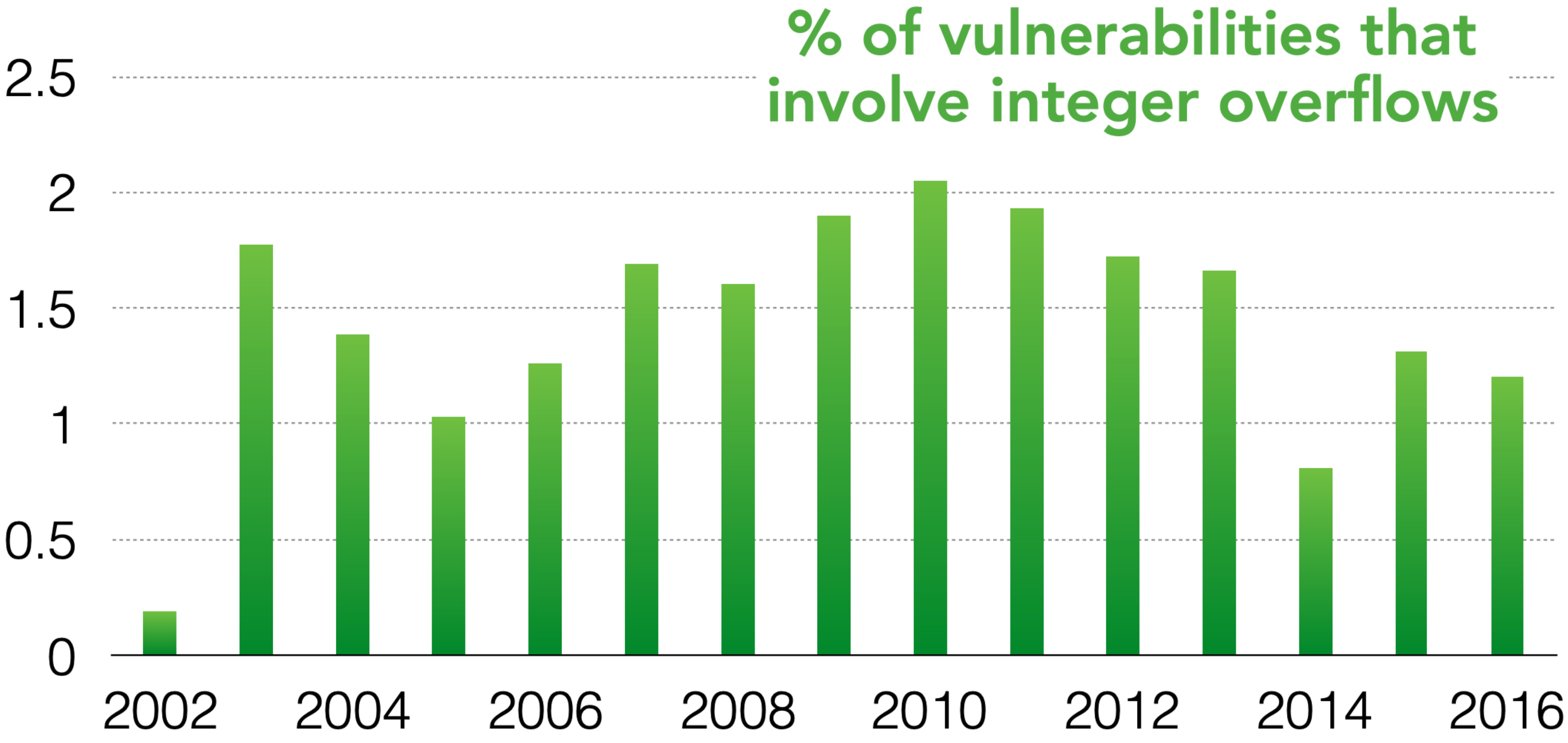
size_t
4 bytes
0 to 4,294,967,295

```
void vulnerable()
{
    size_t len;
    char *buf;
  HUGE
    len = read_int_from_network();
    buf = malloc(len + 5); Wrap-around
    read(fd, buf, len);
    ...
}
```

```
void *malloc(size_t size)
```

Takeaway 2 *You have to know the semantics of your programming language to avoid these errors.*

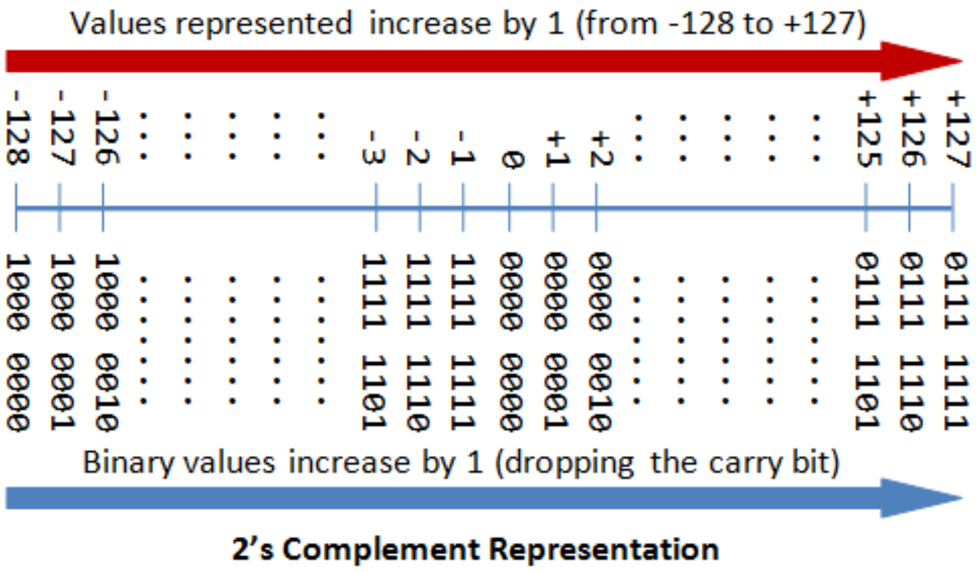
Integer Overflow Prevalence



http://web.nvd.nist.gov/view/vuln/statistics

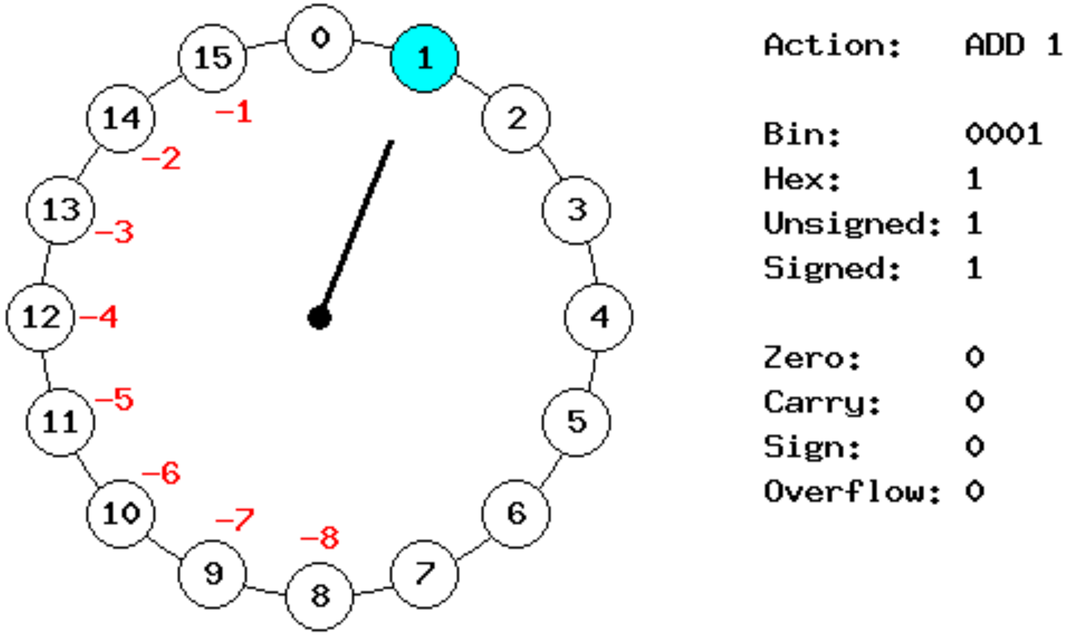
Integers

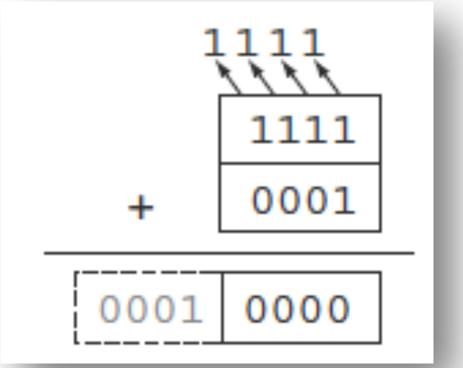
- All built-in integral types (char, short, int, long, etc.) have a limited capacity because they are represented with a **fixed number of bits**.
- In most 32-bit architectures, signed integers (those that can be either positive or negative) are expressed in what is known as **two's compliment notation**.



Integers

- Unlike integers in mathematics, program variables have a fixed range and “wrap around” when they go above their **maximum** value or below their **minimum** value; a very large positive number becomes a very large negative number, or vice versa.





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	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

https://www.tutorialspoint.com/cprogramming/c_data_types.htm

What are the potential underlaying problems of **fixed-sized representation** of numbers?

- Arithmetic Overflow
- Arithmetic Underflow
- Promotion/extension
- Demotion/narrowing
- Sign conversion

Arithmetic Overflow

- When an attacker can take advantage of this behavior, the program is said to contain an integer overflow vulnerability.
- Integer overflow can lead to any number of problems, but in C and C++, an integer overflow is most frequently used as a **lead-in to a buffer overflow exploit**.
  - The buffer overflow might occur when the wrapped-around variable is used to **allocate memory**, **bound a string operation**, or **index into a buffer**.
- Integer overflow can also occur in Java, but because Java enforces **memory safety properties**, integer overflow is not as easy to exploit.

Example 1: *Unsigned Overflow Vulnerability*

```c
#include <stdio.h>

int main(void) {
	unsigned short a = 65000;
	unsigned short b = 540;
	unsigned short c = 0;

	c = a + b;
	printf("Result is %u + %u = %u\n", a, b, c);
	return 0;
}
```

overflow.c

unsigned short
2 bytes
0 to 65,535

Example 1: *Unsigned Overflow Vulnerability*

```c
#include <stdio.h>

int main(void) {
	unsigned short a = 65000;
	unsigned short b = 540;
	unsigned short c = 0;

	c = a + b;
	printf("Result is %u + %u = %u\n", a, b, c);
	return 0;
}
```

unsigned short
2 bytes
0 to 65,535

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc overflow.c -o overflow
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./overflow
Result is 65000 + 540 = 4
```

```
65000 => 0xfde8 => b'1111 1101 1110 1000
  540 => 0x021c => b'0000 0010 0001 1100
                 b'1 0000 0000 0000 0100
```

Example 2: *Arithmetic Underflow Vulnerability*

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

int main(void) {
        assert(sizeof(short)==2);

        unsigned short us = 0;
                 short ss = SHRT_MIN; // -32768

        us -= 1;
        ss -= 1;

        printf("%u %d\n", us, ss);
        return 0;
}
```

underflow.c

unsigned short
2 bytes
0 to 65,535

short
2 bytes
-32,768 to 32,767

Example 2: *Arithmetic Underflow Vulnerability*

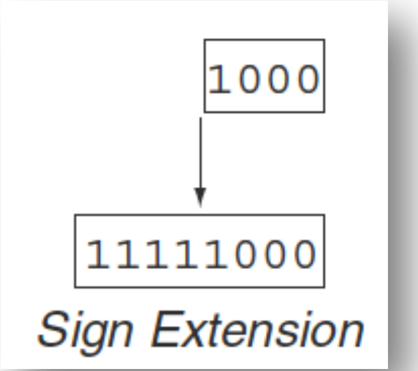
```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(short)==2);

        unsigned short us = 0;
                 short ss = SHRT_MIN; // -32768

        us -= 1;
        ss -= 1;

        printf("%u %d\n", us, ss);
        return 0;
}
```

unsigned short
2 bytes
0 to 65,535

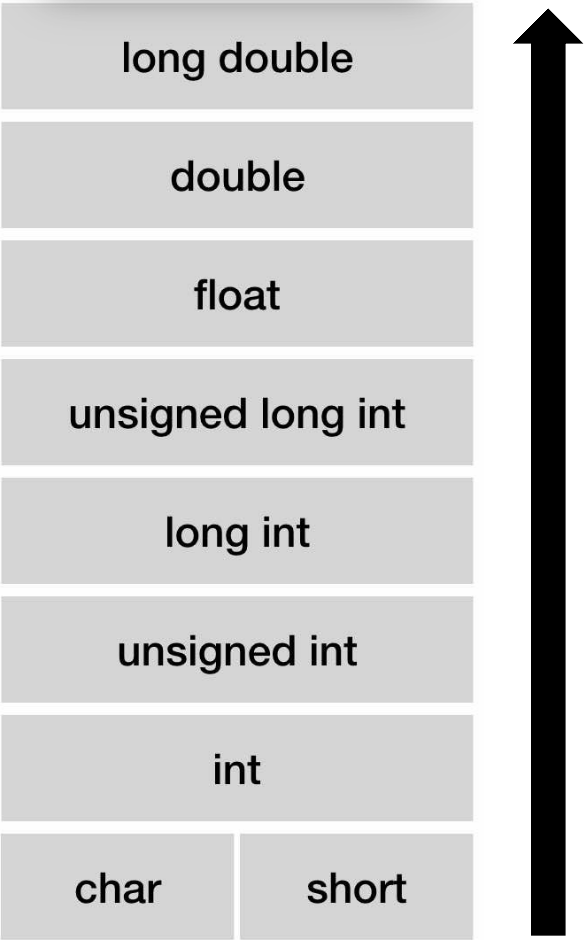
short
2 bytes
-32,768 to 32,767

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc underflow.c -o underflow
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./underflow
65535 32767
```

Integer Promotion/Extension



- Type promotion occurs when we convert from a **small sized integer** to **a larger one**, e.g. from **`short`** to **`int`**.
- For example, when a signed integer is converted from a smaller number of bits to a larger number of bits, the extra bits are *filled in so that the new number retains the same sign*.
  - Negative number casted to signed larger data type, its signed value will remain the same. (`1000` -> `1111 1000`)
  - Negative number casted to unsigned larger data type will increase significantly because its most significant bits will be set. (`1000` -> `1111 1000`)



Example 1: *Integer Promotion/Extension*

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(short)==2);

        short ss = SHRT_MIN;
        int si = ss;

        printf("%d %d\n", ss, si);
        printf("%x %x\n", ss, si);

        return 0;
}
```

signedPromotion.c

short
2 bytes
-32,768 to 32,767

int
4 bytes
-2,147,483,648
to
2,147,483,647

Example 1: *Integer Promotion/Extension*

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(short)==2);

        short ss = SHRT_MIN;
        int si = ss;

        printf("%d %d\n", ss, si);
        printf("%x %x\n", ss, si);

        return 0;
}
```

short
2 bytes
-32,768 to 32,767

int
4 bytes
-2,147,483,648
to
2,147,483,647

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc signedPromotion.c -o signedPromotion
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./signedPromotion
-32768 -32768
8000 ffff8000
```

Example 2: *Integer Promotion/Extension*

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(short)==2);

        short ss = SHRT_MIN;
        unsigned int si = ss;

        printf("%d %u\n", ss, si);
        printf("%x %x\n", ss, si);

        return 0;
}
```

unsignedPromotion.c

short
2 bytes
-32,768 to 32,767

unsigned int
4 bytes
0 to 4,294,967,295

Example 2: *Integer Promotion/Extension*

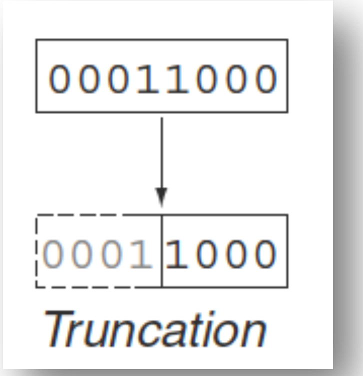
```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(short)==2);

        short ss = SHRT_MIN;
        unsigned int si = ss;

        printf("%d %u\n", ss, si);
        printf("%x %x\n", ss, si);

        return 0;
}
```

short
2 bytes
-32,768 to 32,767

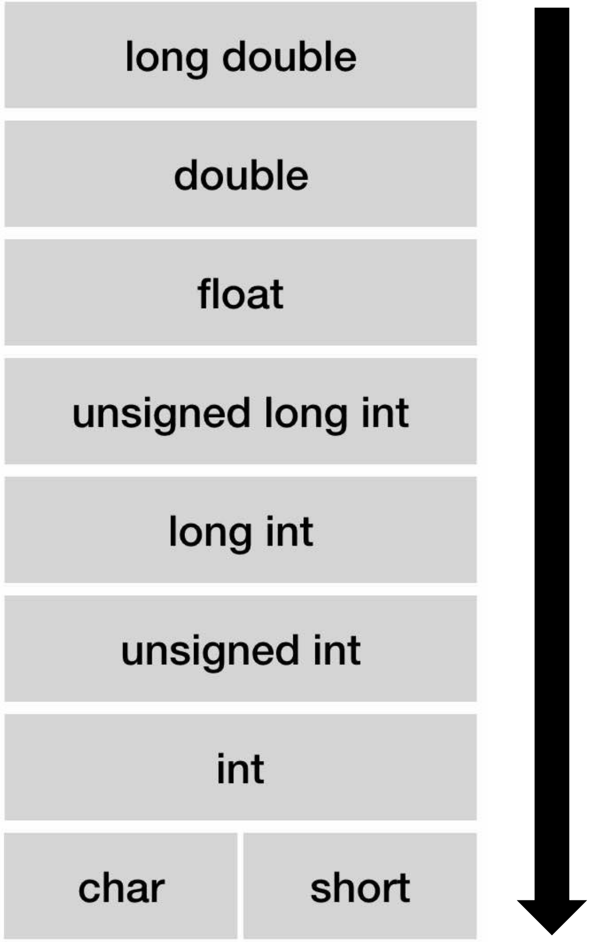
unsigned int
4 bytes
0 to 4,294,967,295

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc unsignedPromotion.c -o unsignedPromotion
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./unsignedPromotion
-32768 4294934528
ffff8000 ffff8000
```

Integer Demotion/Narrowing



- Integer truncation errors occur when an integer data type with a larger number of bits is converted to a data type with fewer bits.
- Narrowing occurs through *truncating the bits to the target type's size.*
  - For example, going from `int` to `short` will result in the bottom 16-bits of the 32-bit int being copied to the `short`.
  - For **unsigned numbers**, this may result is a loss of information (i.e. large numbers being truncated to small numbers).
  - For **signed numbers**, narrowing can result in unexpected change of sign.



Example 1: *Integer Demotion/Narrowing*

signed char
1 byte
-128 to 127

short
2 bytes
-32,768 to 32,767

int
4 bytes
-2,147,483,648
to
2,147,483,647

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

#define MAGIC_NUMBER 0xFFFF7F8F

int main(void) {
        assert(sizeof(short)==2);

        unsigned int ui = MAGIC_NUMBER;
        unsigned short us = ui;
        unsigned char uc = us;

        int si = MAGIC_NUMBER;
        short ss = si;
        signed char sc = ss;

        printf("%10u %5hu %4hhu\n", ui, us, uc);
        printf("%10x %5hx %4hhx\n", ui, us, uc);
        printf("%10d %5hd %4hhd\n", si, ss, sc);
        printf("%10x %5hx %4hhx\n", si, ss, sc);
        return 0;
}
```

narrowing.c

unsigned char
1 byte
0 to 255

unsigned short
2 bytes
0 to 65,535

unsigned int
4 bytes
0 to 4,294,967,295

signed char
1 byte
-128 to 127

short
2 bytes
-32,768 to 32,767

int
4 bytes
-2,147,483,648
to
2,147,483,647

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

#define MAGIC_NUMBER 0xFFFF7F8F

int main(void) {
        assert(sizeof(short)==2);

        unsigned int ui = MAGIC_NUMBER;
        unsigned short us = ui;
        unsigned char uc = us;

        int si = MAGIC_NUMBER;
        short ss = si;
        signed char sc = ss;

        printf("%10u %5hu %4hhu\n", ui, us, uc);
        printf("%10x %5hx %4hhx\n", ui, us, uc);
        printf("%10d %5hd %4hhd\n", si, ss, sc);
        printf("%10x %5hx %4hhx\n", si, ss, sc);
        return 0;
}
```

unsigned char
1 byte
0 to 255

unsigned short
2 bytes
0 to 65,535

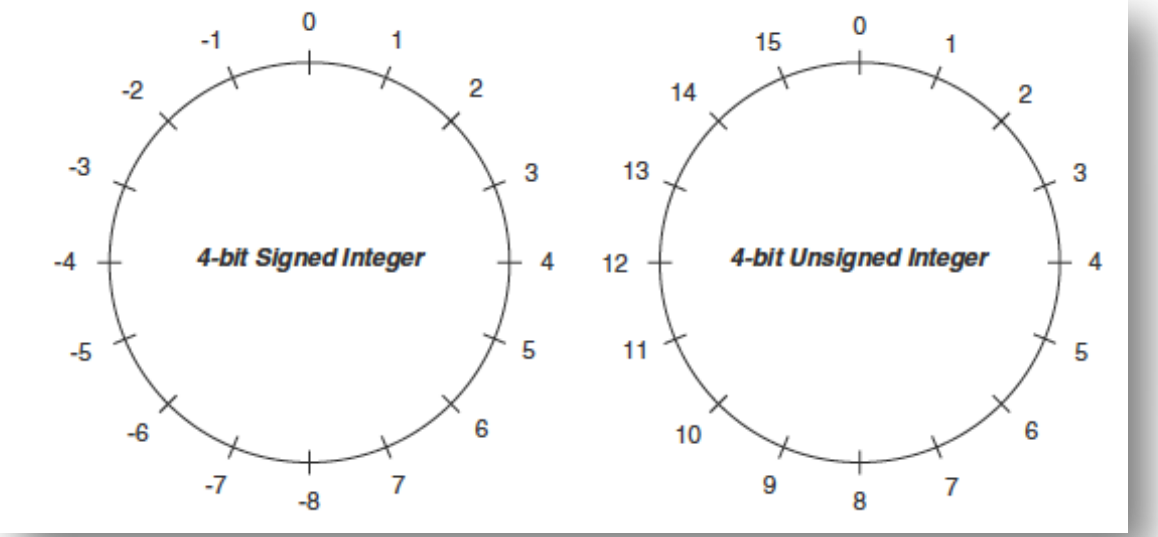
unsigned int
4 bytes
0 to 4,294,967,295

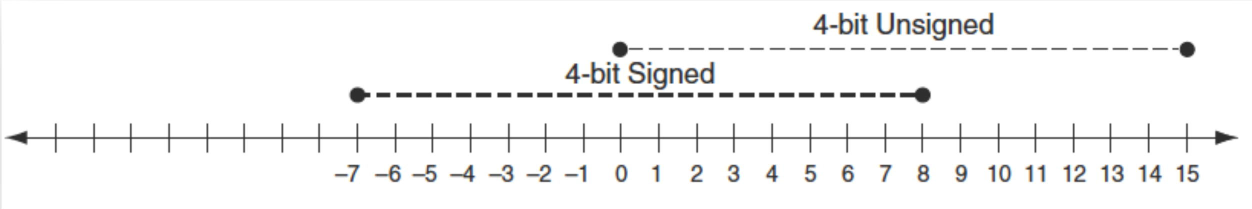
```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc narrowing.c -o narrowing
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./narrowing
4294934415 32655  143
  ffff7f8f  7f8f   8f
    -32881 32655 -113
  ffff7f8f  7f8f   8f
```

Note the change in sign for ***short***

Sign Conversion

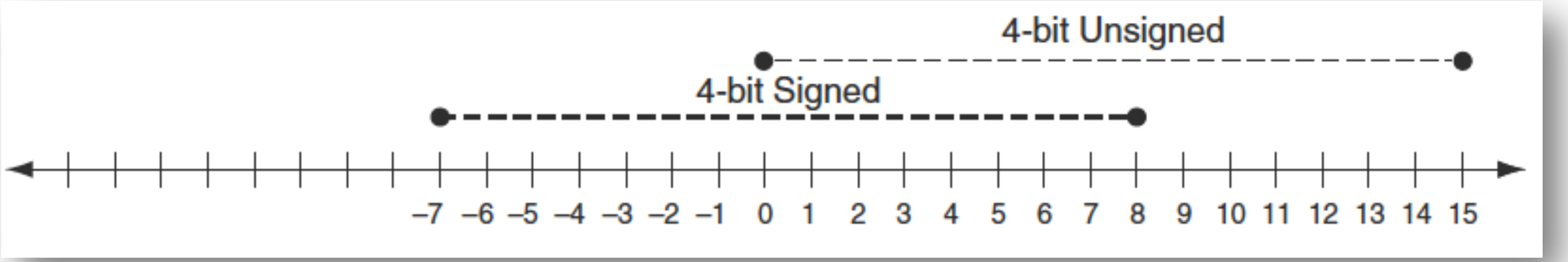
- Both **signed** and **unsigned** data types are capable of representing the same number of values because they have the same number of bits available to them.
  - However there is only **partial overlap** between the range of numbers that the two types can express.





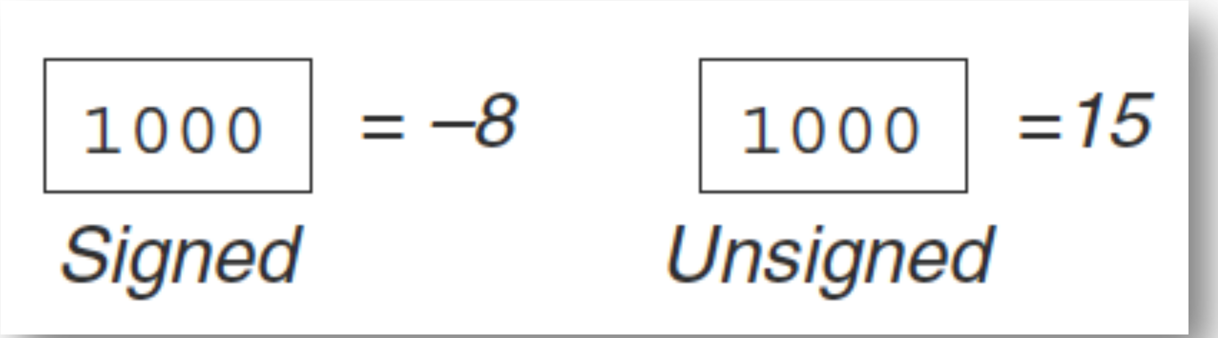
Sign Conversion

- The result of this **partial overlap** is that some values can be converted from an *unsigned data type to a signed data type* and vice versa without a **change** in meaning, while **others cannot**.
- Intuitively, this is the case for signed-to-unsigned conversions because a negative value cannot be represented as an unsigned data type.



Sign Conversion

- In the case of positive values, the problem is that the largest 50% of unsigned values require setting the high-order bit.
- The same bit pattern interpreted as a signed quantity will be negative.
  - If the most-significant-bit (MSB) is a zero (0) then there are no issues with the conversion in either direction.
  - If, however, the MSB is a 1 then a change in sign and value will occur.



Example 1: *Sign Conversion*

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

int main(void) {
        assert(sizeof(short)==2);

        unsigned short us = 0x8080;
        short ss = us;

        printf("%6hu %6hd\n", us, ss);
        printf("%6hx %6hx\n", us, ss);
        return 0;
}
```

conversion.c

unsigned short
2 bytes
0 to 65,535

short
2 bytes
-32,768 to 32,767

Example 1: *Sign Conversion*

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

int main(void) {
        assert(sizeof(short)==2);

        unsigned short us = 0x8080;
        short ss = us;

        printf("%6hu %6hd\n", us, ss);
        printf("%6hx %6hx\n", us, ss);
        return 0;
}
```

unsigned short
2 bytes
0 to 65,535

short
2 bytes
-32,768 to 32,767

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc conversion.c -o conversion
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./conversion
 32896 -32640
  8080   8080
```

Arithmetic Conversion/Promotion

- So far, we have mostly focused on types of the same size (e.g. `short` and `unsigned short`), but if we have arithmetic or logic operations a pattern called the *usual arithmetic conversions* are applied.
- This means, that for **arithmetic and logic operations**, integer types shorter than an `int` are promoted to an `int` for the operation.
  - The promotions can sometimes lead to **unexpected consequences**, such as signed values being interpreted as unsigned and vice versa.

Example 1: *Arithmetic Conversion/Promotion*

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(unsigned char)==1);

        unsigned char uc1 = 0xff;
        unsigned char uc2 = 0;

        if(~uc1 == uc2) {
                printf("%hhx == %hhx\n", ~uc1, uc2);
        } else {
                printf("%hhx != %hhx\n", ~uc1, uc2);
        }
        return 0;
}
```

promotion.c

`unsigned char`
`1 byte`
`0 to 255`

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
        assert(sizeof(unsigned char)==1);

        unsigned char uc1 = 0xff;
        unsigned char uc2 = 0;

        if(~uc1 == uc2) {
                printf("%hhx == %hhx\n", ~uc1, uc2);
        } else {
                printf("%hhx != %hhx\n", ~uc1, uc2);
        }
        return 0;
}
```

unsigned char
1 byte
0 to 255

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc promotion.c -o promotion
promotion.c:13:30: warning: format specifies type 'unsigned char' but the argument has type 'int' [-Wformat]
    printf("%hhx == %hhx\n", ~uc1, uc2);
            ~~~~             ^~~~
            %x
promotion.c:15:30: warning: format specifies type 'unsigned char' but the argument has type 'int' [-Wformat]
    printf("%hhx != %hhx\n", ~uc1, uc2);
            ~~~~             ^~~~
            %x
2 warnings generated.
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./promotion
0 != 0
```

Example 2: *Arithmetic Conversion/Promotion*

As `uc1` has been promoted to the unsigned integer `0x000000ff`, when complimented it results in `0xffffff00`, as shown and thus not equal to zero.

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

int main(void) {
        assert(sizeof(unsigned char)==1);

        unsigned char uc1 = 0xff;
        unsigned char uc2 = 0;

        if(~uc1 == uc2) {
                printf("%08x == %08x\n", ~uc1, uc2);
        } else {
                printf("%08x != %08x\n", ~uc1, uc2);
        }
        return 0;
}
```

promotion2.c

unsigned char
1 byte
0 to 255

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc promotion2.c -o promotion2
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./promotion2
ffffff00 != 00000000
```

INT_MIN

- There is one other anomaly to be aware of based around INT_MIN. When using 2's compliment the number range of an integer is not symmetrical, i.e. the range is:

  -2147483648..2147483647

- All negative values, apart from INT_MIN, have a positive representation. Unfortunately we cannot represent -2147483648 as a positive signed number.
- This leads to the strange behavior that the absolute of INT_MIN and -INT_MIN both are likely to yield INT_MIN.

Example 1: INT_MIN

int
4 bytes
-2,147,483,648
to
2,147,483,647

```
#include <stdio.h>
#include <limits.h>
#include <assert.h>
#include <stdlib.h>

int main(void) {
        assert(sizeof(int)==4);

        int intMin = INT_MIN;

        printf("%d %d %d\n", intMin, abs(intMin), -intMin);
        return 0;
}
```

intMin.c

```
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ gcc intMin.c -o intMin
local-admins-MacBook-Pro:module-05 ahmedtamrawi$ ./intMin
-2147483648 -2147483648 -2147483648
```

Example 1: *Exposing Integer Overflow Vulnerability for Privilege Escalation Attack*

- Suppose a network service keeps track of the number of connections it has received since it has started, and only grants access to the **first five users**.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // ...
  // ...
  // Does nothing to check overflow conditions
  connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

Example 1: *Exposing Integer Overflow Vulnerability for Privilege Escalation Attack*

- An attacker could compromise the above system by making a huge number of connections until the connections **counter overflows** and wraps around to zero.
- At this point, the attacker will be authenticated to the system, which is clearly an undesirable outcome.

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // . . .
  // . . .
  // Does nothing to check overflow conditions
  connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

Example 1: *Exposing Integer Overflow Vulnerability for Privilege Escalation Attack*

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // . . .
  // . . .
  // Does nothing to check overflow conditions
  connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // . . .
  // . . .
  // Prevents overflow conditions
  if(connections < 5)
    connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
   return 1;
}
```

Example 2: *Integer Underflow Vulnerability*

- The most common root problem using integer-based attacks is where the implementation of an algorithm has mixed signed and unsigned values.
- Good targets are where standard library functions, such as `malloc` or `memcpy` have been used, as in both cases they take parameters of type `size_t` (unsigned integer data type).

```
int copySize;
// do work, copySize calculated…
if (copySize > MAX_BUF_SZ) {
    return -1;
}
memcpy(&d, &s, copySize*sizeof(type));
```

Defense Against the Dark Arts

- In short, it can be very **difficult** to protect ourselves against building programs which accidentally or deliberately use the undefined or implementation defined integer behavior.
- Nevertheless, there are several things we can do:
  - Education
  - Use your compiler flags
  - Follow a Security based coding standard
  - Enforce the Coding Standard using a Static Analysis (SA) Tool

Defense Mechanisms: *Education*

- Assuming you've made it this far without skipping the content then you already, hopefully, have a better understanding of the potential issues and vulnerabilities associated with using integers; *spread the word*.
- Further reading includes:
  - *Secure Coding in C and C++* / Robert C. Seacord — 2nd ed. (cert.org/books/secure-coding)
  - *Hacking : the art of exploitation* / Jon Erickson. — 2nd ed. (www.nostarch.com/hacking2.htm)

Defense Mechanisms: *Compiler Flags*

- Some compilers support **compiler flags** that affect the behavior of integers.
- For example, it is not uncommon for `gcc` programmers to utilize these flags:

`-ftrapv`

This option generates traps for signed overflow on addition, subtraction, multiplication operations. The options `-ftrapv` and `-fwrapv` override each other, so using `-ftrapv -fwrapv` on the command-line results in `-fwrapv` being effective. Note that only active options override, so using `-ftrapv -fwrapv -fno-wrapv` on the command-line results in `-ftrapv` being effective.

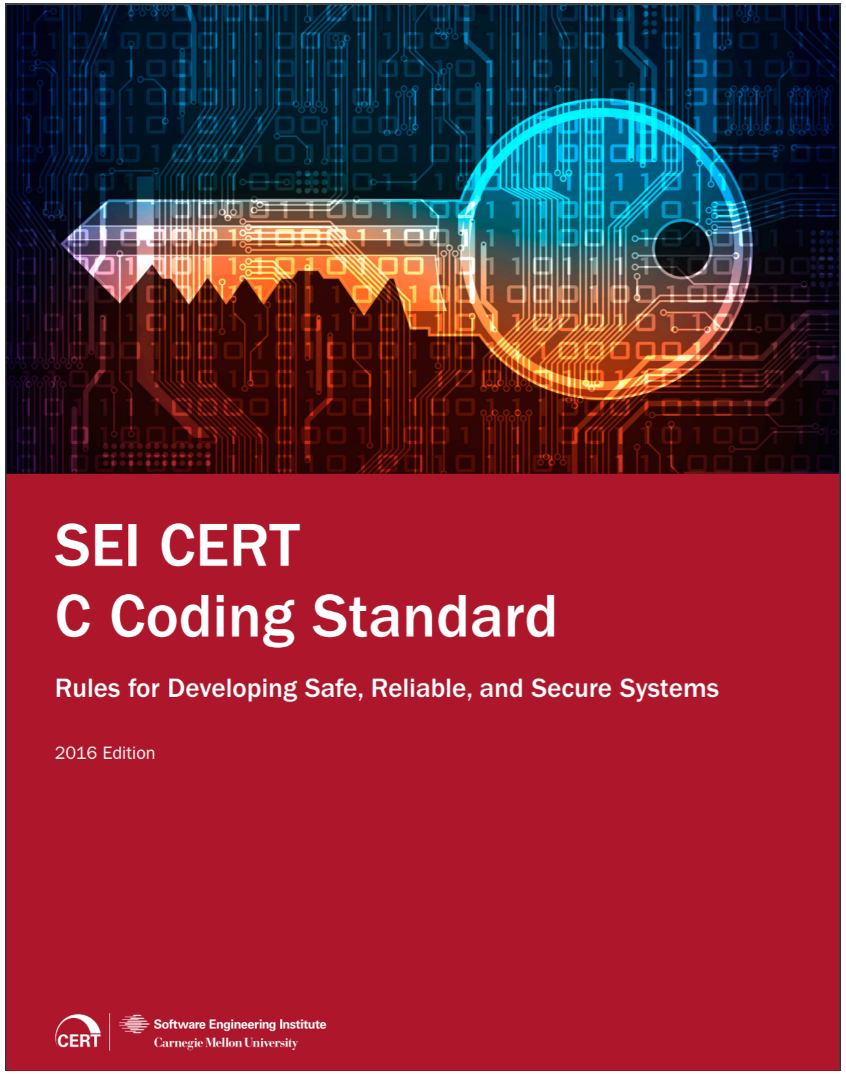
`-fwrapv`

This option instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. This flag enables some optimizations and disables others. The options `-ftrapv` and `-fwrapv` override each other, so using `-ftrapv -fwrapv` on the command-line results in `-fwrapv` being effective. Note that only active options override, so using `-ftrapv -fwrapv -fno-wrapv` on the command-line results in `-ftrapv` being effective.

`-fwrapv-pointer`

This option instructs the compiler to assume that pointer arithmetic overflow on addition and subtraction wraps around using twos-complement representation. This flag disables some optimizations which assume pointer overflow is invalid.

Defense Mechanisms: *Security Standards*



5.1.1.1 Noncompliant Code Example

This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands `ui_a` and `ui_b`. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability.

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

5.1.1.2 Compliant Solution (Precondition Test)

This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap:

```
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    /* Handle error */
  } else {
    usum = ui_a + ui_b;
  }
  /* ... */
}
```

Defense Mechanisms: *Software Analysis*

- It is so important that any coding standard is enforced through automation; ideally it is a natural part of a Continuous Integration (CI) strategy.
  - Source code is checked after a clean build but before tests are executed.
- Importantly for embedded systems we want consistency of checking across compilers, so you'll need to seek out analyzers that understand your compiler's dialect.
- Static analyzers supporting the CERT standard:
  - ParaSoft: https://www.parasoft.com/solutions/compliance/cert/
  - Coverity: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html