Integer bugs

TOOR - Computer Security

Hallgrímur H. Gunnarsson

Reykjavík University

2012-04-30

Introduction

Integer manipulation can be critical from a security perspective.

Integers might be used in the determination of an offset or size for memory allocation, copying, concatenation, etc.

Exploitable integer bugs have been found in critical software, such as, OpenSSH, Apache, Linux Kernel, Microsoft Windows, MacOS X, Mozilla Firefox, Google Chrome, Opera, Integer Explorer, etc.

As we will see, integer bugs can be subtle and hard to spot.

Finding integer bugs requires solid understanding of the rules governing integers in C

Overview - What we will cover today

Boundary condition vulnerabilities:

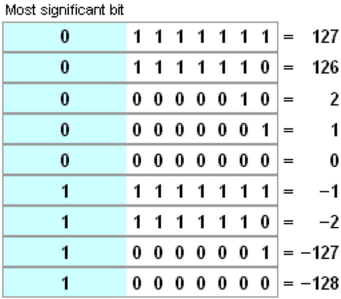
- Signed integer boundaries (overflow/underflow)
- Unsigned integer boundaries (overflow/underflow)

Type conversion vulnerabilities:

- Signed/unsigned conversions
- Sign extension
- Truncation
- Signed/unsigned comparison

Heap overflows are a common consequence of integer bugs, they will be covered by Ymir in the next lecture.

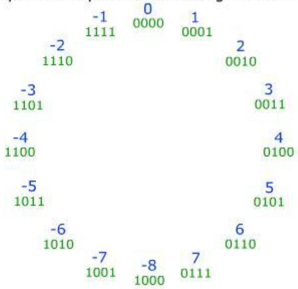
Two’s complement system



Most significant bit									
0	1	1	1	1	1	1	1	=	127
0	1	1	1	1	1	1	0	=	126
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

8-bit two's-complement integers

Two's Complement representation using 4 bit binary strings



How do we negate x in two’s complement? Calculate: ∼x + 1

Definitions:

- Overflow: Value exceeds maximum value
- Underflow: Value is less than minimum value

Arithmetic boundary conditions

Basic integer types in C have minimum and maximum values

Value range is determined by the underlying representation in memory

What happens when we cross these boundaries?

Simple arithmetic on a variable, such as addition, subtraction, or multiplication, can result in a value that cannot be stored in that variable.

Unsigned integer boundaries

Overflow example:

```
unsigned char a;
a = 0xe2;
a = a + 0x22;
```

The result (`0x104 == 260`) cannot be stored in a (beyond maximum possible value for `unsigned char`).

Underflow example:

```
unsigned char a;
a = 0;
a = a - 1;
```

The result (`0xFF == -1`) cannot be stored in a (below minimum possible value for `unsigned char`).

Unsigned integer boundaries

The C99 standards states:

> *”A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.”*

The modulo operation guarantees that the result stays within range (by ”wrapping around”).

Example:

```
r = (0xffffffff + 0x1) % 0x100000000
r = (0x100000000) % 0x100000000 = 0
```

guarantees that only the lowest 32 bits of the result are used.

Unsigned integer boundaries

Overflow example:

```
unsigned char a;
a = 0xe2;
a = a + 0x22;
```

The result (`0x104 == 260`) cannot be stored in a, so it wraps around, `0x104 % (0xFF + 1) == 0x104 % 0x100 == 4`

Underflow example:

```
unsigned char a;
a = 0;
a = a - 1;
```

The result (`0xFF == -1`) cannot be stored in a, so it wraps around, `0xFF % 0x100 == 0xFF == 255`

OpenSSH CHAP SKEY/BSDAUTH – CVE-2002-0639

Integer overflow in OpenSSH 2.9.9 through 3.3 (remote code execution): nresp is under our control

```
unsigned int nresp;
// ...

nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
    }
}
```

Signed integer boundaries

According to C99, the result of an arithmetic overflow or underflow with a signed integer is implementation defined.

On most common architectures, the results of signed arithmetic overflows are well defined.

The boundary behavior is a natural consequence of how two's complement arithmetic is implemented.

When an operation on a signed integer causes an arithmetic overflow or underflow, the resulting value ”wraps around the sign boundary”, and typically causes a change in sign.

Signed integer boundaries

Overflow example:

```c
int a = 0x7fffffff;
printf("%d\n", a);
a = a+1;
printf("%d\n", a);
```

The signed addition overflows the sign boundary, causing our value to wrap around 0x80000000 and become a negative number.

Prints out:

```
2147483647
-2147483648
```

Signed integer boundaries

Underflow example:

```
int a = 0x80000020;

printf("%d\n", a);
a = a - 200;
printf("%d\n", a);
```

The signed subtraction overflows the sign boundary, causing our value to wrap below 0x80000000 and become a positive number.

Prints out:

```
-2147483616
2147483480
```

Signed integer vulnerability

```
char *read_data(int sockfd) {
    char *buf;
    int length = network_get_int(sockfd);

    if (!(buf = (char *)malloc(MAXCHARS)))
        die("malloc");

    if (length < 0 || length + 1 >= MAXCHARS) {
        free(buf); die("bad length");
    }

    if (read(sockfd, buf, length) <= 0) {
        free(buf); die("read");
    }

    buf[value] = '\0';
    return buf;
}
```

Signed integer vulnerability

Let's assume that: `int length == 0x7fffffff == 2147483647`

```
if (length < 0 || length + 1 >= MAXCHARS) {
    free(buf);
    die("bad length");
}
```

`0x7fffffff` passes the check because `2147483647 >= 0` and `2147483647 + 1 == -2147483648 < MAXCHARS`

```
if (read(sockfd, buf, length) <= 0) {
    free(buf);
    die("read");
}
```

`read()` reads and copies data into the buffer until the connection is closed (or we reach 2147483647 bytes)

Integer conversion: widening

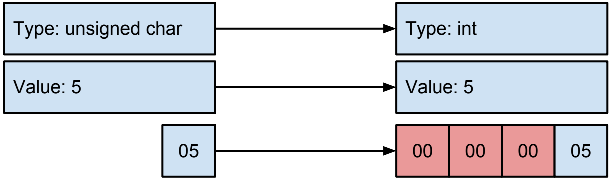
When an integer type is converted from a narrow type to a wider type, the machine copies the bit pattern and sets all the new high bits to either 0 or 1.

If the source type is unsigned, the machine uses **zero extension**, i.e. it propagates the value 0 to all high bits in the new wider type.

If the source type is signed, the machine uses **sign extension**, i.e. it propagates the sign bit fomr the source type to all unused bits in the new wider type.

Integer conversion: widening

Value-preserving conversion from unsigned char to signed int:

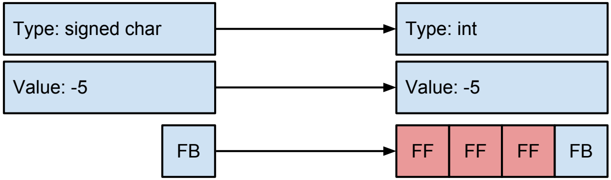


Source type is unsigned $\implies$ zero extension

The value is preserved.

Integer conversion: widening

Value-preserving conversion from signed char to int:

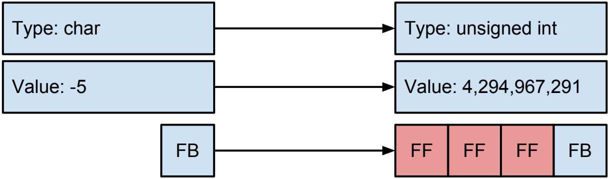


Source type is signed $\implies$ sign extension

The value is preserved.

Integer conversion: widening

Value-changing conversion from signed char to unsigned int:



Source type is signed $\Longrightarrow$ sign extension

The value changes if the source value is negative!

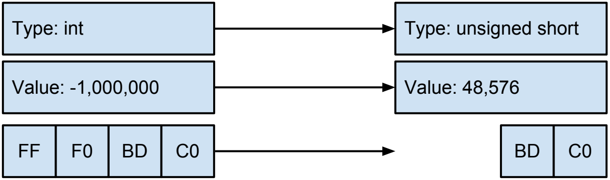
This might be surprising to some programmers, but bug-hunters love surprises. We will look at sign extension vulnerabilities later in the lecture.

Integer conversion: narrowing

When an integer type is converted from a wider type to a narrow type, the machine uses truncation.

The bits from the wider type that do not fit in the new narrower type are dropped.

All such conversions are value-changing because we are losing precision.

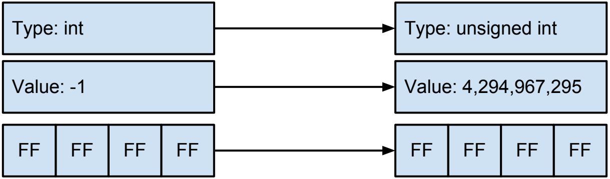


Integer conversion: signed/unsigned

When a conversion occurs between a signed type and an unsigned type of the same width, nothing is changed in the bit pattern, but the value is interpreted in the context of the new type.

The conversion is therefore value-changing.

1) signed to unsigned conversion:



Type: int → Type: unsigned int

Value: -1 → Value: 4,294,967,295

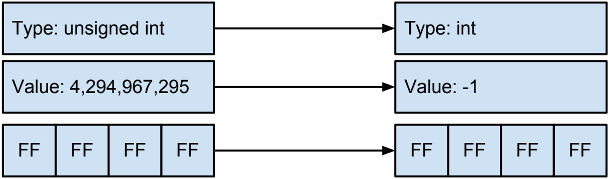
FF	FF	FF	FF	→	FF	FF	FF	FF

Integer conversion: signed/unsigned

When a conversion occurs between a signed type and an unsigned type of the same width, nothing is changed in the bit pattern, but the value is interpreted in the context of the new type.

The conversion is therefore value-changing.

2) unsigned to signed conversion:



Type: unsigned int → Type: int

Value: 4,294,967,295 → Value: -1

FF FF FF FF → FF FF FF FF

Simple conversions

Explicit casts specify an explicit type conversion, e.g.

```
    (unsigned char)x
```

`x` is converted into type `unsigned char` and the resulting type of the expression is `unsigned char`

Assignment:

```
    short int a;
    int b = -10;

    a = b;
```

The operand on the right must be converted into the type of the left operand.

(What happens in this instance?)

Simple conversions

Explicit casts specify an explicit type conversion, e.g.

```
(unsigned char)x
```

`x` is converted into type `unsigned char` and the resulting type of the expression is `unsigned char`

Assignment:

```
short int a;
int b = -10;

a = b;
```

The operand on the right must be converted into the type of the left operand.

(What happens in this instance? Truncation because of narrowing)

Simple conversions continued

Function calls:

```
void f(int a, unsigned char b) {
    // ...
}

short x = 10;
int y = 20;

f(x, y);
```

The arguments in the call are converted according to the function prototype.

What happens in this instance?

Simple conversions continued

```c
void f(int a, unsigned char b) {
    // ...
}

short x = 10;
int y = 20;

f(x, y);
```

What happens in this instance?

- `x` is widened with sign extension, value is preserved
- `y` is narrowed with truncation and interpreted as an unsigned value, value might change

Simple conversions continued

Return values:

```
char f(void) {
    int a = 42;
    return a;
}
```

`return` does a conversion of its operand to the type specified in the function definition

In this instance, a is narrowed and truncated into `char`

Integer promotions

Certain operators in C require an integer operand of type int or unsigned int, e.g. addition:

```
a + b
```

Both `a` and `b` are promoted to integers.

For such operators, C uses **integer promotion** to transform the narrower integer operands into the correct type.

Anything narrower than `int` is promoted:

- `unsigned char`, `signed char` – promoted
- `unsigned short`, `short` – promoted
- `unsigned int`, `int` – not promoted
- `long`, `unsigned long` – not promoted

Integer promotions

How do we decide whether to promote to `int` or `unsigned int` ?

We use `int` if a value-preserving transformation to an `int` can be performed.

Otherwise, we use `unsigned int`.

Integer promotions are invoked in many settings, e.g. the unary + and − operators, the bitwise shift operators, expressions in switch statements, etc.

Integer promotions are also a critical component of C's rules for handling arithmetic expressions, which are called the **usual arithmetic conversions.**

Usual arithmetic conversions

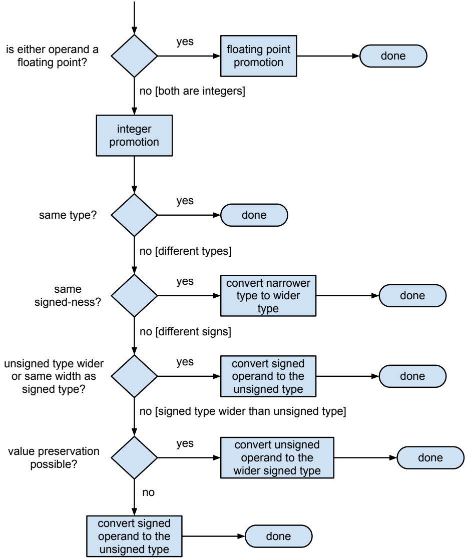
Usual arithmetic conversions occurs in the following cases:

- Addition
- Subtraction
- Multiplicative operators, $*$, /, %
- Relational and equality operators
- Binary bitwise operators
- Question mark operator (ternary operator)

The goal is to reconcile two types into a compatible type for an arithmetic operation

Let's look at the rules for usual arithmetic conversion

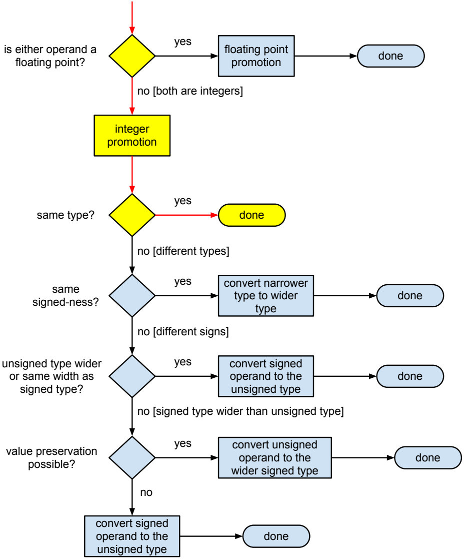
Usual arithmetic conversions


is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
unsigned char a = 255;
unsigned char b = 255;
unsigned char c = a+b;

if (c > 300)
    printf("x\n");
```

What happens?

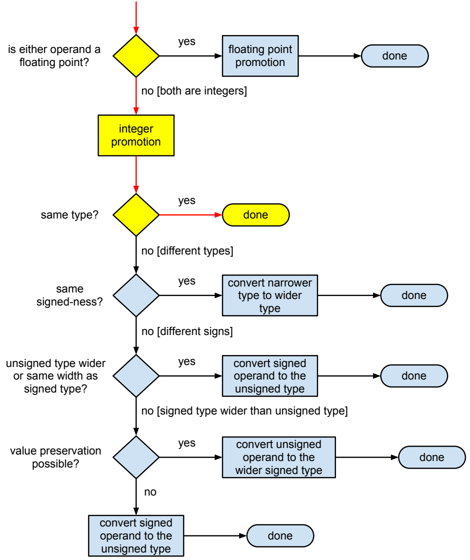

is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
unsigned char a = 255;
unsigned char b = 255;
unsigned char c = a+b;

if (c > 300)
    printf("x\n");
```

What happens?

a and b are promoted to integers, the + operation is performed, the resulting integer is truncated into the narrow type `unsigned char`

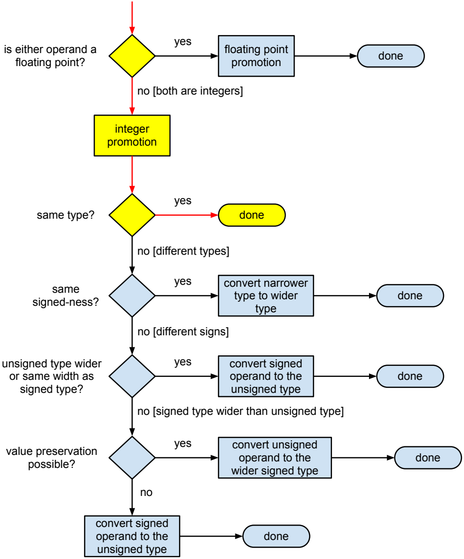


is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
unsigned char a = 255;
unsigned char b = 255;

if ((a + b) > 300)
    printf("x\n");
```

What happens?



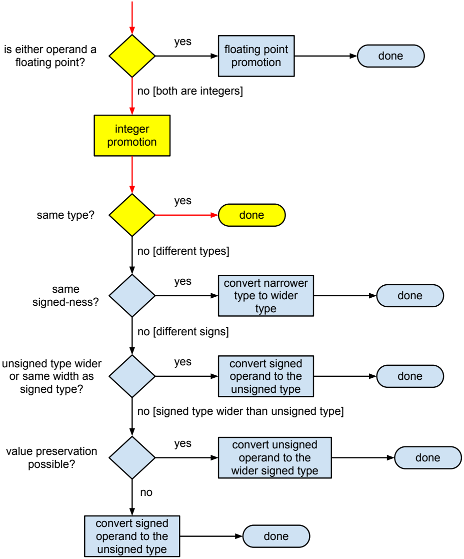
is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
unsigned char a = 255;
unsigned char b = 255;

if ((a + b) > 300)
    printf("x\n");
```

What happens?

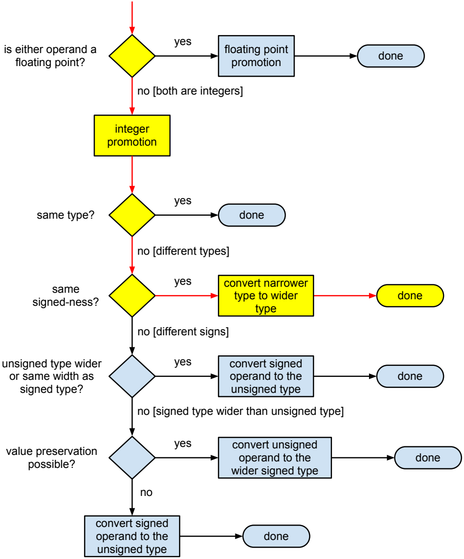
a and b are promoted to integers, the + operation is performed, the result is compared to 300


is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
int a = 10;
long b = 20;
long long c = a + b;
```

What is happening here?



is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

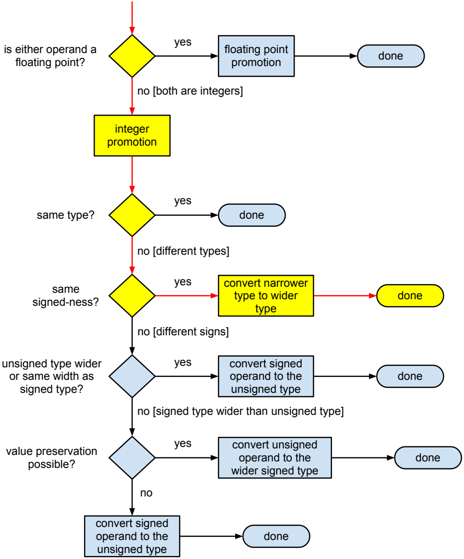
```
int a = 10;
long b = 20;
long long c = a + b;
```

What is happening here?

Two things.

a is converted into the wider type of b before + is performed.

The resulting type, a long int, is then converted into a long long by assignment to c.



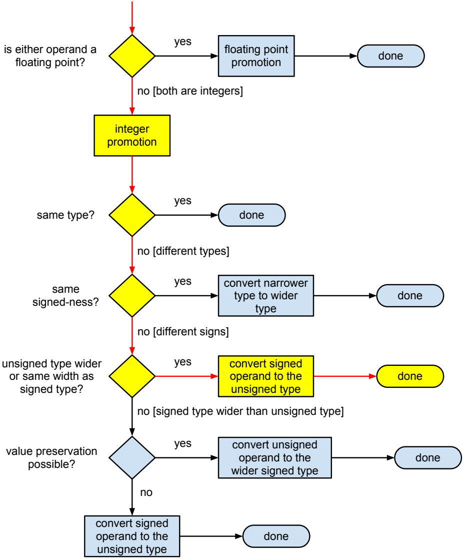
is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
int len = -5;

if (len < sizeof(int))
    printf("x\n");
```

Does it print x?

Why / why not?



is either operand a floating point?

yes

floating point promotion

done

no [both are integers]

integer promotion

same type?

yes

done

no [different types]

same signed-ness?

yes

convert narrower type to wider type

done

no [different signs]

unsigned type wider or same width as signed type?

yes

convert signed operand to the unsigned type

done

no [signed type wider than unsigned type]

value preservation possible?

yes

convert unsigned operand to the wider signed type

done

no

convert signed operand to the unsigned type

done

Usual arithmetic conversions

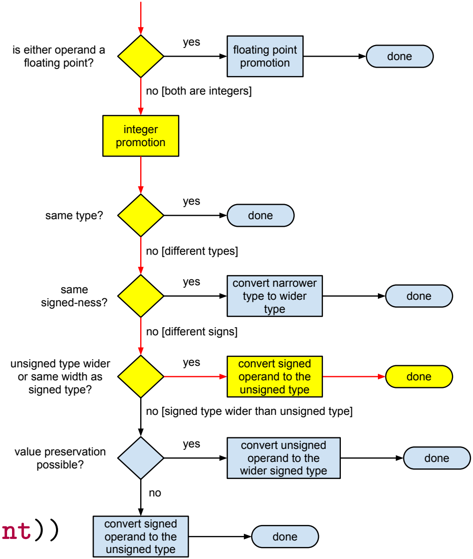
```
int len = -5;

if (len < sizeof(int))
    printf("x\n");
```

`sizeof(T)` has type `size_t`, which is an unsigned integer type.

As a result, `len` is converted to an unsigned integer type.

The actual comparison is:

```
if (4294967291 < sizeof(int))
    printf("x\n");
```



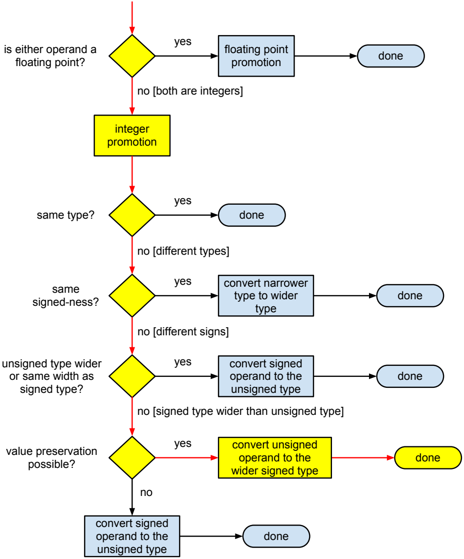
is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Usual arithmetic conversions

```
long long int a = 10;
unsigned int b = 5;
(a+b);
```

The signed argument, a long long int, can represent all the values of the unsigned argument, an unsigned int.

As a result, the compiler converts both operands to the signed operand's type: long long int.


is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

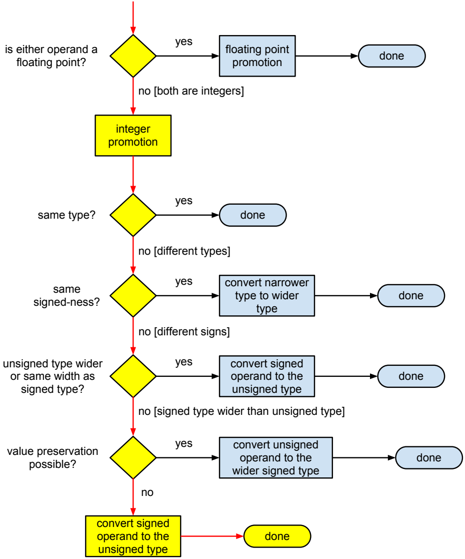
Usual arithmetic conversions

```
unsigned int a = 10;
long int b = 20;
(a+b);
```

On x86, long and int are both 4 bytes in size.

long has higher rank than int, so the signed type (long int) cannot hold all the values of the unsigned type (unsigned int).

As a result, the compiler converts both operands to unsigned long int.



is either operand a floating point?
yes
floating point promotion
done
no [both are integers]
integer promotion
same type?
yes
done
no [different types]
same signed-ness?
yes
convert narrower type to wider type
done
no [different signs]
unsigned type wider or same width as signed type?
yes
convert signed operand to the unsigned type
done
no [signed type wider than unsigned type]
value preservation possible?
yes
convert unsigned operand to the wider signed type
done
no
convert signed operand to the unsigned type
done

Type conversion vulnerabilities

Implicit type conversions can be a source of many bugs.

We will look at the following cases:

- Signed/unsigned conversions
- Sign extension
- Truncation
- Signed/unsigned comparison

Signed/Unsigned conversions

Many library routines that take a size parameter use an argument of type size_t:

size_t is an unsigned integer type that is the same size as a pointer

```
ssize_t read(int fd, void *buf, size_t count);
void *memcpy(void *dest, const void *src, size_t n);
void *malloc(size_t size);
int snprintf(char *s, size_t size, const char *fmt, ...);
char *strncpy(char *dest, const char *src, size_t n);
char *strncat(char *dest, const char *src, size_t n);
```

What happens when we use a signed integer for the size in a call?

Signed/Unsigned conversions

Many library routines that take a size parameter use an argument of type size_t:

size_t is an unsigned integer type that is the same size as a pointer

```
ssize_t read(int fd, void *buf, size_t count);
void *memcpy(void *dest, const void *src, size_t n);
void *malloc(size_t size);
int snprintf(char *s, size_t size, const char *fmt, ...);
char *strncpy(char *dest, const char *src, size_t n);
char *strncat(char *dest, const char *src, size_t n);
```

What happens when we use a signed integer for the size in a call?
We get an implicit conversion to unsigned int

Signed comparison vulnerability

```
int read_user_data(int sockfd) {
    int length, sockfd, n;
    char buffer[1024];

    length = get_user_length(sockfd);

    if (length > 1024)
        return -1;

    if (read(sockfd, buffer, length) < 0)
        return -1

    return 0;
}
```

Signed comparison vulnerability

```
if (length > 1024)
    return -1;
```

Constraint: length can be `[INT_MIN, 1024]`

```
if (read(sockfd, buffer, length) < 0)
    return -1
```

`read()` expects a length of type size_t (unsigned type), implicit conversion occurs.

A small negative value will become a big positive value, e.g. -100 will become 4294967196.

Unbounded read into local buffer $\Longrightarrow$ stack overflow!

Sign extension vulnerability

Remember: sign extension occurs when a signed integer type is widened

When is the type widened?

- Explicit cast, assignment, function call, etc.
- Arithmetic conversions (e.g. integer promotion)

Example:

```
char len;

len = get_len_field();
snprintf(dst, len, "%s", src);
```

What happens when `len < 0` ?

Sign extension vulnerability

Code:

```
char len;

len = get_len_field();
snprintf(dst, len, "%s", src);
```

`snprintf` takes a `size_t` (unsigned integer type) length argument

`len` is signed and will be widened via sign extension into a larger unsigned type (!)

-100 becomes 4294967196

Sign extension vulnerability

Okay, let's fix it by casting `len` to an unsigned value before passing it.

```
char len;

len = get_len_field();
snprintf(dst, (unsigned int)len, "%s", src);
```

Has it been fixed?

Sign extension vulnerability

Okay, let's fix it by casting `len` to an unsigned value before passing it.

```
char len;

len = get_len_field();
snprintf(dst, (unsigned int)len, "%s", src);
```

Has it been fixed?

No, sign extension also occurs during the conversion from char to unsigned int.

More advanced example

```c
unsigned short read_length(int sockfd) { ... }
int read_packet(int sockfd) {
    short length;
    char *buffer;

    length = read_length(sockfd);

    if (length > 1024)
        return -1;

    buffer = (char *)malloc(length + 1);

    if ((n = read(sockfd, buffer, length) < 0)) {
        free(buffer); return -1;
    }

    buffer[n] = '\0';
    return 0;
}
```

More advanced example

Let's say `read_length()` returns `0xffff`, i.e. 65535

```
    length = read_length(sockfd);
```

Here unsigned short is converted into signed short, so `length == 0xffff == -1`

```
    if (length > 1024)
        return -1;
```

`length` passes the length check because `-1 < 1024`

```
    buffer = (char *)malloc(length + 1);
```

`length + 1` causes integer promotion, `length == 0xffff` is sign-extended to `0xffffffff`, and `length+1` wraps around to 0

`malloc(0)` happily allocates a buffer of size 0

More advanced example

`buffer` is now a heap-allocated buffer of size 0

```
if ((n = read(sockfd, buffer, length) < 0)) {
    free(buffer); return -1;
}
```

`read()` takes a `size_t` for third parameter, `length` is converted from a `signed short` to a `size_t`, sign extension occurs and converts the number into a large positive number.

The call to read allows us to read a large number of bytes into the buffer, resulting in a heap overflow.

Example of a real sign extension vulnerability: CVE-2003-0694, sendmail prescan, remote code execution

Truncation vulnerability

Most famous truncation bug: CVE-2001-0144, SSH deattack, remote code execution

Extremely subtle bug that requires a clever exploit

Too complex to go into details here, but it will be on the student presentation list!

The deattack code in SSH was introduced as a security measure to prevent packet injection into a SSH session.

The code searches for repeated blocks to determine whether any manipulation is occurring. To make the code fast, they used a hashing table and hashed each block. The hash table was dynamically resized and the bug occurs in the resizing code.

ssh deattack

```c
int
detect_attack(unsigned char *buf, u_int32_t len,
              unsigned char *IV) {
    static u_int16_t *h = (u_int16_t *) NULL;
    static u_int16_t n = HASH_MINSIZE / HASH_ENTRYSIZE;
    register u_int32_t i, j;
    u_int32_t l;
    register unsigned char *c;
    unsigned char *d;
    // ..
    if (h == NULL) {
        n = l;
        h = (u_int16_t *)xmalloc(n * HASH_ENTRYSIZE);
    } else {
        if (l > n) {
            n = l;
            h = (u_int16_t *)xrealloc(h, n * HASH_ENTRYSIZE);
        }
    }
```

ssh deattack

Amazingly the discoverer of the bug, Michael Zalewski, found it by just trying a long username:

```
$ ssh -l long_user_name ...
```

Comparisons

Remember: comparison operators invoke the usual arithmetic conversions on the operands so that the comparison can be made on compatible types.

The promotions and conversions can lead to value changes, and the comparison might not be operating as the programmer intended.

Let's look at a few examples of how comparisons can go wrong.

Comparison vulnerability

```
#define MAX_SIZE 1024;

int read_packet(int sockfd)
{
    short length;
    char buf[MAX_SIZE];

    length = network_get_short(sockfd);

    if (length - sizeof(short) <= 0 || length > MAX_SIZE)
        return -1;

    if (read(sockfd, buf, length - sizeof(short)) < 0)
        return -1;

    return 0;
}
```

Comparison vulnerability

```
if (length - sizeof(short) <= 0 || length > MAX_SIZE)
    return -1;
```

`sizeof(short)` has type `size_t`, which is an unsigned integer type.

In the expression `(length - sizeof(short))`, `length` is first promoted into a `signed int` and then converted to an unsigned integer type as part of the usual arithmetic conversions.

The resulting type of the expression is an unsigned integer type. As a result, the subtraction can never be less than 0 and the check is ineffective.

A value of e.g. `length == 1` would pass the test. A negative value, e.g. `length == -1 == 0xffff`, would also pass the test.

Comparison vulnerability

```c
if (length - sizeof(short) <= 0 || length > MAX_SIZE)
    return -1;
```

In the expression `length > MAX_SIZE`, `length` is promoted to a signed int via sign extension. A negative value would pass the test.

As a result, if `length == 0xffff` then it would pass both tests!

```c
if (read(sockfd, buf, length - sizeof(short)) < 0)
    return -1;
```

`length - sizeof(short)` becomes an unsigned int expression, a value of `length == 0xffff` would be sign extended to `0xffffffff` and then converted into an unsigned value:

```c
if (read(sockfd, buf, 4294967295 - sizeof(short)) < 0)
```

Signed comparison vulnerability

```
int read_data(int sockfd)
{
    char buf[1024];
    unsigned short max = sizeof(buf);
    short length;

    length = get_network_short(sockfd);

    if (length > max)
        return -1;

    if (read(sockfd, buf, length) < 0)
        return -1

    return 0;
}
```

Signed comparison vulnerability

```
if (length > max)
    return -1;
```

length and max are short integers (of different signed-ness), they both get promoted to signed integers. A negative value in length will evade the length check against max because length will be sign-extended from e.g. 0xffff to 0xffffffff and -1 < max.

```
if (read(sockfd, buf, length) < 0)
    return -1
```

The call to read() supplies length as a parameter, and it will be promoted to an integer and then converted to an unsigned integer type. If length = 0xffff then we will get:

```
if (read(sockfd, buf, 4294967295) < 0)
```

Unsigned comparison vulnerability

```
int get_int(char *data) {
    unsigned int n = atoi(data);
    if (n < 0 || n > 1024) return -1;
    return n;
}

int main(void) {
    unsigned long n;
    char buf[1024];

    if (argc < 2) return 0;
    n = get_int(argv[1]);
    if (n < 0) return 1;

    memset(buf, 'A', n);
    return 0;
}
```

Unsigned comparison vulnerability

If n is out of range then get_int returns −1. In main() the return value from get_int is stored in n, an unsigned long.

```
unsigned long n;
char buf[1024];

if (argc < 2) return 0;
n = get_int(argv[1]);
```

Given −1 the value of n becomes 0xffffffff, so the following check is never true:

```
if (n < 0) return 1;
```

resulting in:

```
memset(buf, 'A', 4294967295);
```

Questions

Questions?

References

This lecture was heavily based on chapter 6 in *The Art of Software Security Assessment* by Mark Dowd, John McDonald and Justin Schuh.

Basic Integer Overflows by blexim, Phrack issue #60