C Language Issues

Purpose and Agenda

- The purpose of this lecture
  1. Presents basic aspects of C language: types, number representation, conversions
  2. Presents vulnerabilities due to bad understanding or misuse of C aspects
- Agenda
  - Introduction
  - C Basics. Data representation
  - Arithmetic Boundary Conditions
    - Overview
    - Unsigned integer boundaries
    - Signed integer boundaries
  - Type conversions
    - Overview
    - Type conversions vulnerabilities
  - Conclusions

Introduction
Overview

- subject of security research since stack-mashing attacks largely replaced by heap exploits
- root causes of many reported issues
- problem is due to limited representation space for numbers
- the nuance of the problem vary from language to language

CWE References

- CWE-682: Incorrect Calculation
- CWE-190: Integer Overflow or Wraparound
  - 24th place in Mitre’s Top 25
- CWE-191: Integer Underflow (Wrap or Wraparound)
- CWE-192: Integer Coercion Error

CWE-682: Incorrect Calculation

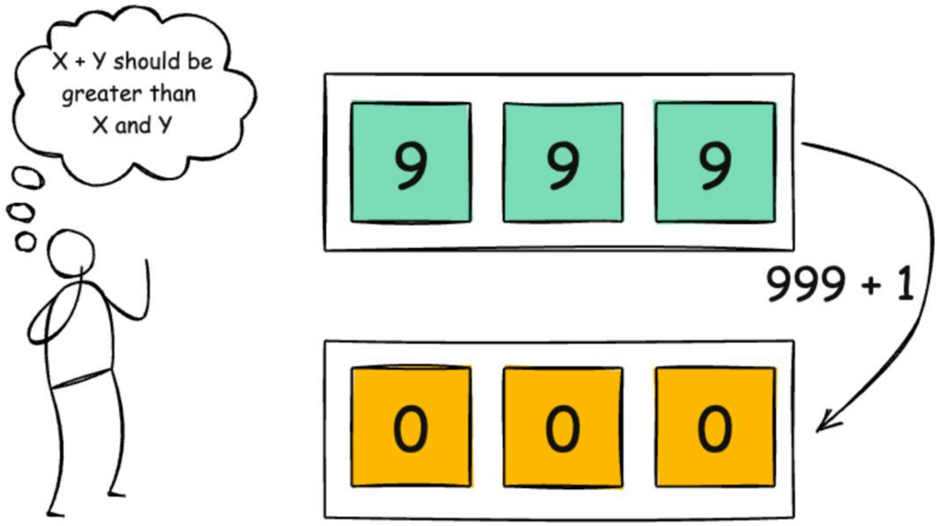
```
int *p = x;
char * second_char = (char *)(p + 1);
```

```
img_t table_ptr;
/*struct containing img data, 10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

CWE-190: Integer Overflow or Wraparound

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

```
short int bytesRec = 0;
char buf[SOMEBIGNUM];

while(bytesRec < MAXGET) {
    bytesRec += getFromInput(buf+bytesRec);
}
```



CWE-191: Integer Underflow

```
#include <stdio.h>
#include <stdbool.h>
main (void)
{
    int i;
    i = -2147483648;
    i = i - 1;
    return 0;
}
```

CWE-192: Integer Coercion Error

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;

sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;

if (numHeaders > 100) {
        ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader);
ParsePacketHeaders(packet, headers);
```

Affected Languages

- all common languages could be affected
  - the effects depends on how a language handles integers internally
- C and C++ are the most dangerous
  - most likely an integer overflow could be turned into a buffer overflow
- all languages are prone to DoS and logic errors

C Basics. Data Representation

Types

- signed and unsigned
  - precision and width
  - default specifier: signed
- basic types
  - character: char, signed char, unsigned char
  - integer (signed / unsigned)
    - short int / unsigned short int
    - int / unsigned int
    - long int / unsigned long int
    - long long int / unsigned long long int
  - floating: float, double, long double
  - bit fields
- type aliases
  - UNIX: int8_t / uint8_t, int16_t / uint16_t, int32_t / uint32_t, int64_t / uint64_t
  - WINDOWS: BYTE / CHAR, WORD, DWORD, QWORD

Width, Minimum and Maximum Values

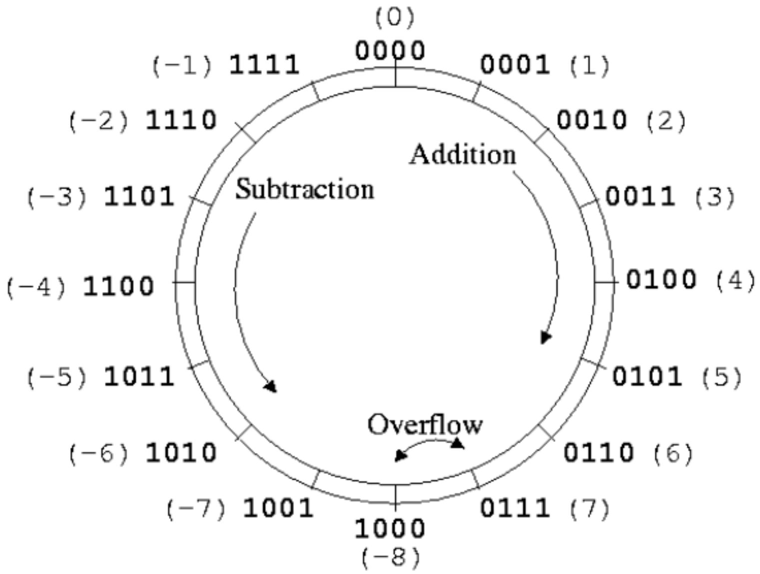
Type	Width	Minimum value	Maximum value
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32,768	32,767
unsigned short	16	0	65,535
int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
long	32	-2,147,483,648	2,147,483,647
unsigned long	32	0	4,294,967,295
long long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

$$[0, 2^n - 1]$$
$$[-2^{n-1}, 2^{n-1} - 1]$$

Binary Encoding

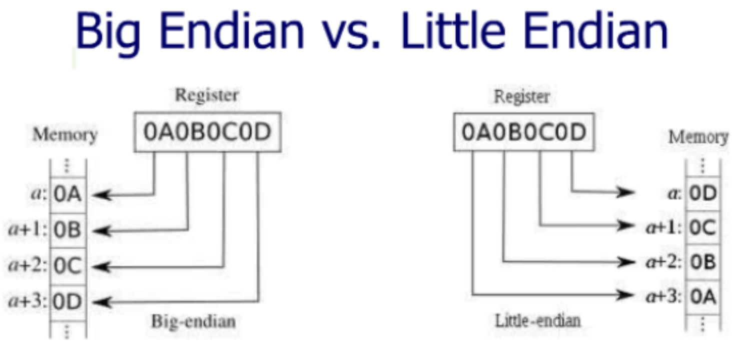
- 0 and 1 bits
- signed numbers use value bits and a sign bit
- possible arithmetic schemes
  - sign and magnitude (+: easy for humans; -: difficult for CPU)
  - one complement
    - negative numbers: invert all bits
    - +: good for CPU
    - -: borrowing complexity, two zeros
  - two complement (commonly used)
  - negative numbers: invert all bits and add 1
  - all operations works normal as for unsigned numbers
  - there is just one value for zero (0)

Binary encoding



Byte Order

- big endian: most-significant byte at smaller memory addresses
- little endian: most-significant byte at bigger memory addresses



Common Implementations

- ILP32: integer, long, pointer represented on 32 bits
- **ILP32LL**
  - integer, long, pointer represented on 32 bits, long long on 64 bits
  - de facto standard for 32-bit platforms
- **LP64**
  - long and pointer represented on 64 bits
  - de facto standard for 64-bit platforms
- ILP64: integer, long, pointer represented on 64 bits
- LLP64: long long and pointer represented on 64 bits

Arithmetic Boundary Conditions

Context and definitions

- about number ranges (minimum and maximum values)
- dependent on their binary representation
- numeric / integer overflow condition
  - the maximum value an integer can hold is (over)exceeded
  - example

```
unsigned int a;
a = 0xFFFFFFFF;
a = a + 1;      // a = 0;
```

- numeric / **integer underflow** condition
  - the minimum value an integer can hold is (under)exceeded
  - example

```
unsigned int a;
a = 0;
a = a - 1; // a = 0xFFFFFFFF
```

Security Risks of Integer Overflow / Underflow

- could lead to incorrect variables' values ⇒
  - undetermined application's behavior
  - application's integrity violation
- could lead to a cascade of faults
- give an attacker multiple possibility to influence the application's execution
- vulnerabilities are due to arithmetic operations using user controlled (directly or indirectly) numbers
- examples
  - bad lengths / limits calculated for memory allocation ⇒ buffer overflow
  - bad length / limit checking ⇒ buffer overflow

Unsigned Integer Boundaries,
Unsigned Integer Overflow

- operations are subject to the rules of modular arithmetic
  - result is "real result" modulo (max represented value + 1)
  - example: R=R%2^32
- extra bits of overflow results are truncated
- operations that could lead to overflow: addition, multiplication, shifting to left
- at the CPU level, the carry flag (CF) is set at overflow
- in case of multiplication, it could be possible at machine level to get the high bits of the
- (overflowed) result

```c
unsigned int a;
a = 0xE0000020;
a = a + 0x20000020;
// -> a = (0xE0000020 + 0x20000020) % 0x10000000
// a = 0x40
```

Example

```
u_char *make_table(
    unsigned int width,
    unsigned int height,
    u_char *init_row)
{
  unsigned int n;
  int i;
  u_char *buf;

  n = height * width;
  buf = (char*) malloc(n);

  if (!buf)
    return NULL;

  for (i = 0; i < height; i++)
    memcpy(&buf[i * width], init_row, width);
}
```

Example

```c
u_char *make_table(
    unsigned int width,
    unsigned int height,
    u_char *init_row)
{
  unsigned int n;
  int i;
  u_char *buf;

  n = height * width;
  buf = (char*) malloc(n);

  if (!buf)
    return NULL;

  for (i = 0; i < height; i++)
    memcpy(&buf[i * width], init_row, width);
}
```

- n could be overflowed by multiplication of user-controlled height and width, resulting in a relatively small number
  - example (on 32 bits)
    - $0x400 * 0x10000001 = 0x400$ (hexadecimal)
    - $1024 * 268435457 = 1024$ (decimal)
- still, the for loop goes for a large portion of (overflowed) memory
  - in the example
    - 1024 bytes allocated ⇒ **one element allocated**
    - BUT . . . **more than one elements accessed**

Example II (Vulnerability in OpenSSH 3.1)

```
u_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);
}
packet_check_eom();
```

Example II (Vulnerability in OpenSSH 3.1)

```
u_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);
}
packet_check_eom();
```

- *nresp* not checked, user-controlled
- *on x86 nresp is unsigned int (4 bytes)*
- *UINT_MAX = 0xFFFFFFFF*
- *overflow when nresp >= 0xFFFFFFFF/4 (0x40000000)*

Unsigned Integer Underflow

- operations are subject to the rules of modular arithmetic
- caused by an operation whose result is under the minimum representable value of 0
- underflow results in huge positive (unsigned) numbers
- operations that could lead to underflow: subtraction

Example

```
struct header {
  unsigned int len;
  unsigned int type;
};

char *read_packet (int sockfd)
{
  int n;
  unsigned int len;
  struct header hdr;
  static char buffer[1024];

  if (full_read(sockfd, (void*) &hdr, sizeof(hdr)) <= 0) {
    error("full read: %m");
    return NULL;
  }

  len = ntol(hdr.len);
  if (len > (1024 + sizeof (hdr) - 1))
    return NULL;

  if (full_read(sockfd, buffer, len - sizeof(hdr)) <= 0)
    return NULL;
  buffer[sizeof(buffer) - 1] = 0;

  return strdup(buffer);
}
```

both test pass for length = 0x7FFFFFFF

Signed Integer Overflow and Underflow

- overflow could result in a (large) negative number due to the twos complement representation
- underflow could transform a negative number into a positive one
- operations that could lead to overflow: addition, multiplication, shifting to left
- result are not so easy to be classified: depends on how the sign bit is affected

Signed Integer Vulnerability Example

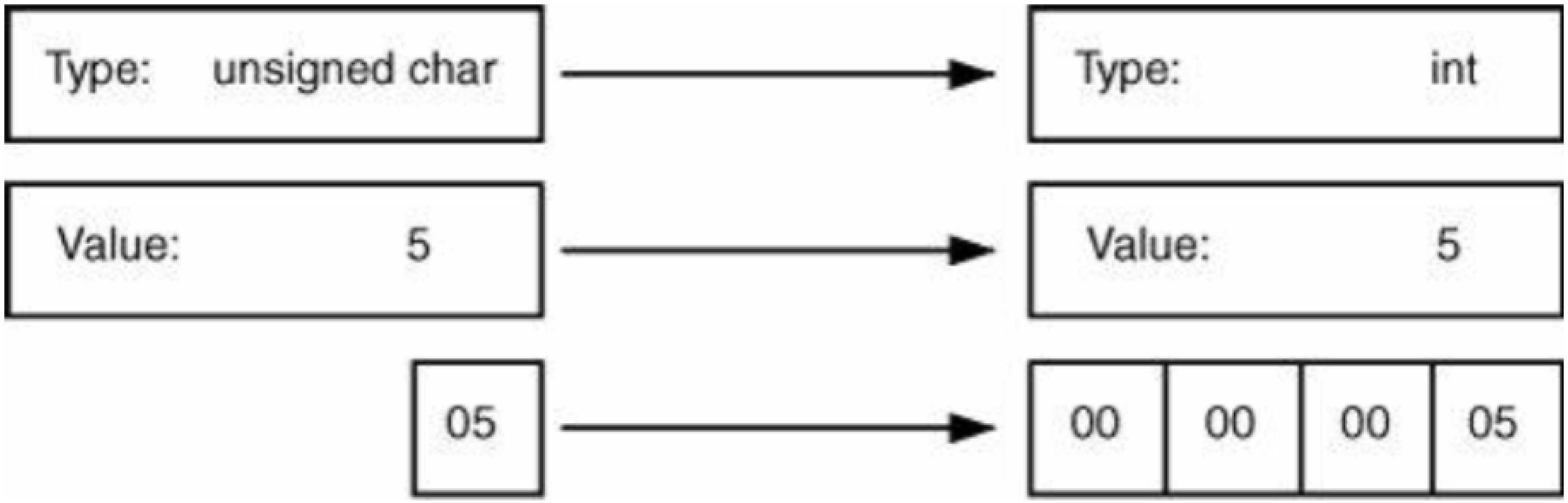
```
char* read_data(int sockfd)
{
  char *buf;
  int value;
  int length = network_get_int(sockfd);

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

  if (length < 0 || length + 1 > MAXCHARS) { // both tests pass for length = 0x7FFFFFFF
    free(buf);
    die("bad length");
  }

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

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

Type Conversions
Overview

Definition and Context

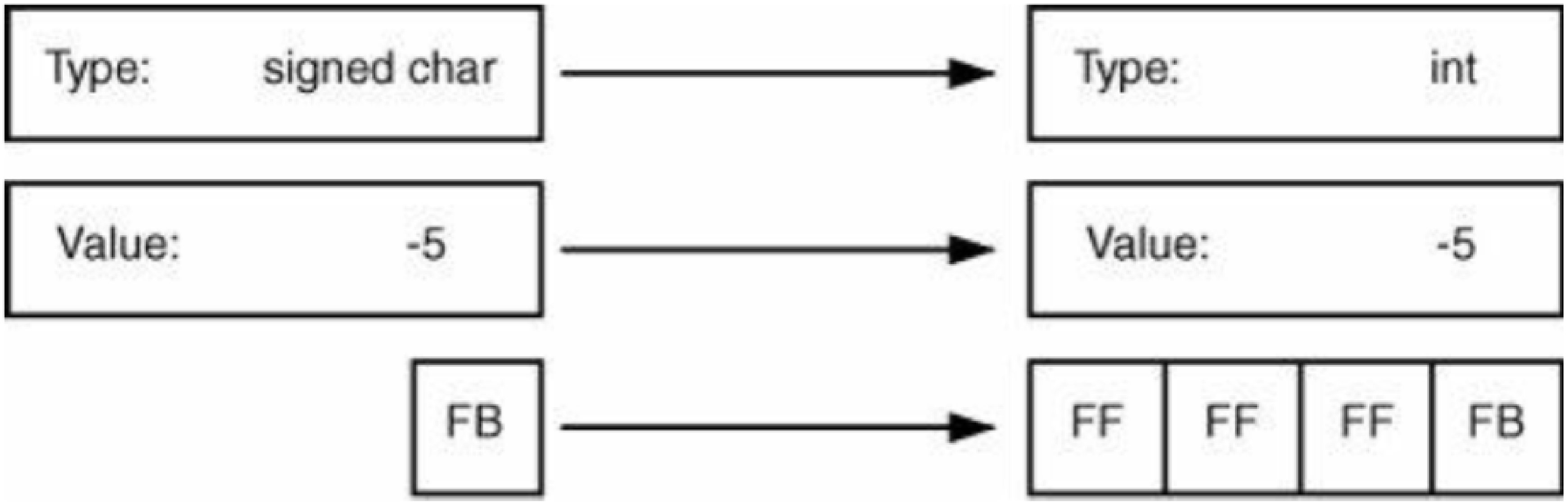
- conversion of an object of one type to another type
  - **explicit** vs. **implicit** (default) type conversion
- value-preserving vs. value-changing
  - new type can represent (or not) all possible values of the old type
- cases
  - 1. **widening** from narrow to wider types
    - zero-extension: used for unsigned numbers
    - sign-extension: used for signed numbers
    - could be value-changing
  - 2. **narrowing** by truncation
    - is value-changing
  - 3. **conversion** between signed and unsigned
    - is value-changing

Conversion Rules for Integers: Widening from Narrow to Wider Type



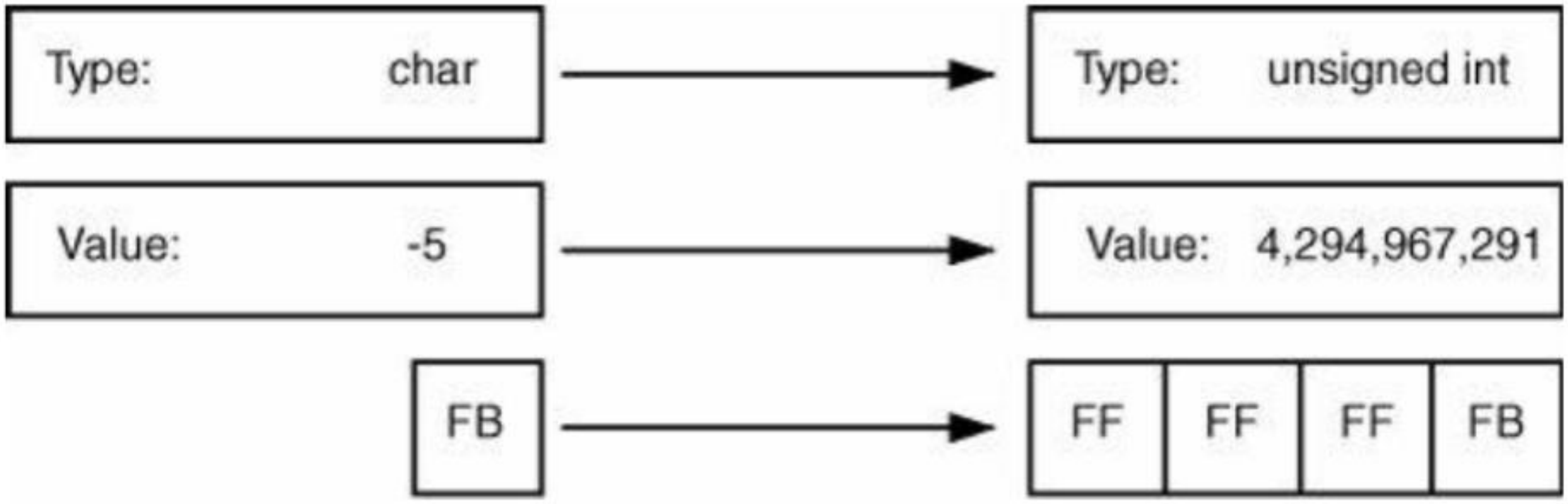
value preserving conversion: “unsigned char” to “signed int”

Conversion Rules for Integers: Widening from Narrow to Wider Type (cont.)



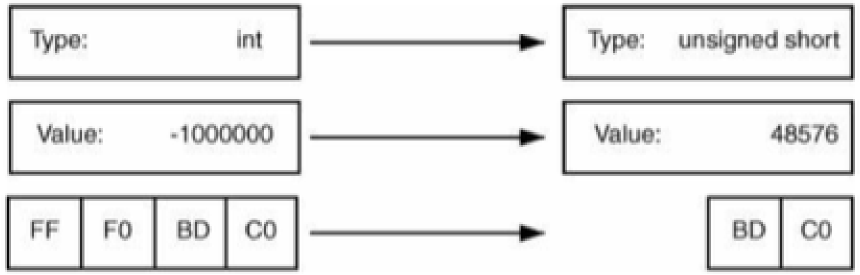
value preserving conversion: “signed char” to “signed int”

Conversion Rules for Integers: Widening from Narrow to Wider Type (cont.)

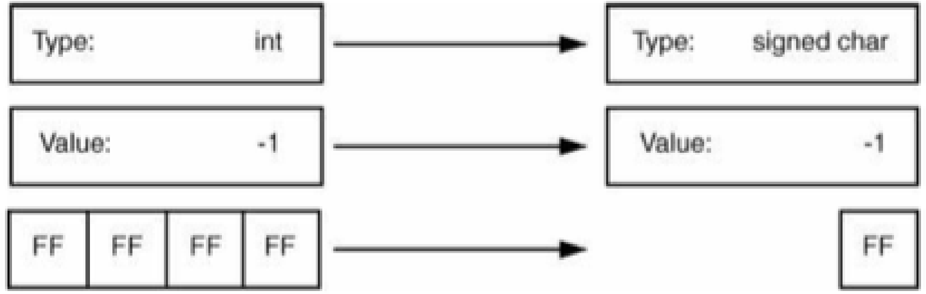


value changing conversion: “signed char” to “unsigned int”

Conversion Rules for Integers: Narrowing (by truncation)

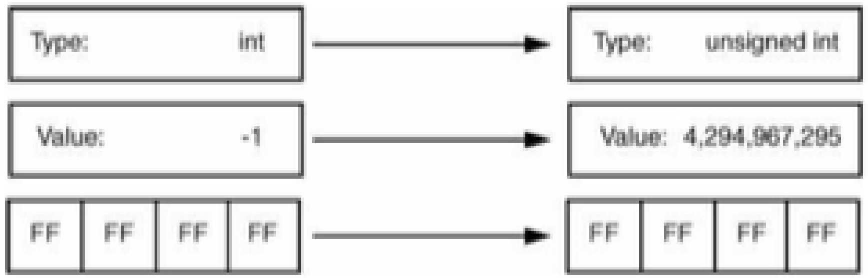


truncation: “signed int” to “unsigned short”

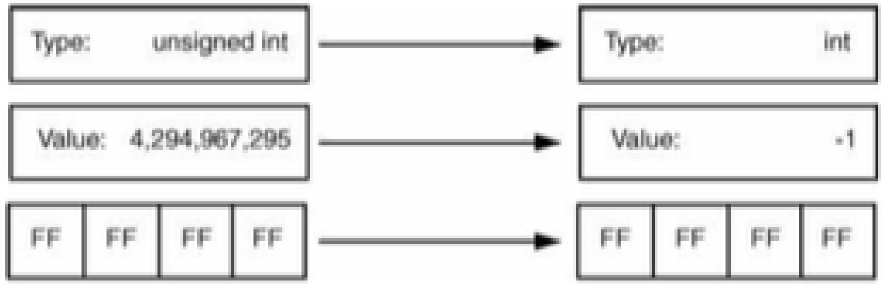


truncation: “signed int” to “signed char”

Conversion Rules for Integers: Between Signed and Unsigned



conversion: “signed int” to “unsigned int”



conversion: “unsigned int” to “signed int”

Conversion Rules for Integers: Rules and Effects

- narrower signed → wider unsigned
  - sign extension ⇒ **value-changing**
- narrower signed → wider signed
  - sign extension ⇒ **value-preserving**
- narrower unsigned → wider (any)
  - zero extension ⇒ **value-preserving**
- wider (any) → narrower (any)
  - truncation ⇒ **value-changing**
- signed ↔ unsigned (of the same width)
  - bits preserved, but the value is otherwise interpreted ⇒ **value-changing**

Simple conversions

- (type)casts: `(unsigned char)var`
- assignments

```
short int v1;
int v2 = -10;
v1 = v2;
```

- function calls
  - prototype-based

```
int dostuff(int x, unsigned char y);

void func(void)
{
  char a=42;
  unsigned short b=43;
  long long int c;
  c=dostuff(a, b);
}
```

Simple Conversions (cont.)

- return based

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

Integer Promotions (Widening Conversions to Int)

- narrower integer types → int
- used for (when)
  - certain operators require an integer operand
  - handling of usual arithmetic conversions
- integer conversion rank (rank integer types by their width from low to high)
  - 1. long long int, unsigned long long int
  - 2. long int, unsigned long int
  - 3. unsigned int, int
  - 4. unsigned short, short
  - 5. char, unsigned char, signed char
- any place an int or unsigned int can be used, any integer type with a lower integer conversion rank can also be used
- when the variable type is wider than the int, promotion does nothing
- when the variable type is narrower than the int
  - if value-preserving transformation to an int ⇒ promote
  - otherwise: a value-preserving conversion to an unsigned int is performed

Integer Promotion Applied

- unary + operator performs integer promotion on its operand
- unary - operator performs integer promotion on its operand and then does a negation
  - regardless of whether the operand is signed after promotion, a twos complement negation is done
  - the Leblancian paradox: the twos complement negative of 0x80000000 is the same number 0x80000000
  - vulnerable code example

```c
int bank1[1000], bank2[1000];

void hashbank (int index, int value)
{
  int *bank = bank1;

  if (index < 0) {
    bank = bank2;
    index = -index;

  }

  // this will write at bank2[-648]
  // for index = 0x80000000
  bank[index % 1000] = value;
}
```

Integer Promotion Applied (cont.)

- unary ~ operator operator performs integer promotion on its operand and then does a ones complement
- bit-wise shift operator
  - performs integer promotion on both arguments
  - the type of the result is the same as the promoted type for the left argument

```
char a = 1;
char c = 16;
int bob;
bob = a << c;
```

- switch statement performs integer promotion

Usual Arithmetic Conversions

- used in evaluation of C expressions where arguments are of different types
- $\Rightarrow$ they must be reconciled in a compatible type

Usual Arithmetic Conversions. Rule 1

- ***floating points take precedence***
- if one of argument has a floating point type $\Rightarrow$ the other argument is converted to a floating point type
- if one floating point argument is less precise than the other $\Rightarrow$ less precise to more precise

Usual Arithmetic Conversions. Rule 2

- if no argument is float ⇒ apply integer promotion
  - all operands are promoted to integers, if needed
  - example 1 (comparison work OK, even if seems to be an overflow)

```
unsigned char term1 = 255;
unsigned char term2 = 255;

if ((term1 + term2) > 300)
  do_something();
```

Usual Arithmetic Conversions. Rule 2 (cont.

example 2 vulnerable (*do_something()* will be executed!)

```
unsigned short a = 1;

if ((a - 5) < 0)
  do_something();
```

example 2 correct (*do_something()* will NOT be executed)

```
unsigned short a = 1;
a = a - 5;

if (a < 0)
  do_something();
```

Usual Arithmetic Conversions. Rule 3

- same type after integer promotion
  - if after integer promotion operands are of the same type, nothing else is done

Usual Arithmetic Conversions. Rule 4

- same sign, different types
  - if after integer promotion operands have the same sign, but different widths
  - ⇒ the narrower is converted to the wider type
  - example (everything is OK)

```
int t1 = 5;
long int t2 = 6;
long long int res;

res = t1 + t2;
```

Usual Arithmetic Conversions. Rule 5

- unsigned type wider than or same width as signed type
  - the narrower signed type is converted to the wider (or equal width) unsigned type
  - example (wrong comparison ⇒ do_something() will NOT be executed!)

```
int t = -5;

if (t < sizeof(int))  // i.e. "4294967291 < 4"
  do_something();
```

Usual Arithmetic Conversions. Rule 6

- signed type wider than unsigned type, value preservation possible
  - the narrower unsigned type is converted to the wider signed type
  - example 3 (everything is OK)

```
long long int a = 10;
unsigned int b = 5;

(a+b);
```

Usual Arithmetic Conversions. Rule 7

- signed type wider than unsigned type, value preservation impossible
  - when narrower unsigned type’s values cannot be represented by the wider signed type, both are converted to the unsigned type corresponding to the signed type
  - example (it is assumed the “int” and “long int” are of the same width)

```
unsigned int a = 10;
long int b = 20;

(a+b); // the result is of "unsigned long" type
```

Usual Arithmetic Conversion Applied

- addition
- subtraction
- multiplicative operators
- relational and equality operators
- binary bit-wise operators
- question mark operator

Type Conversion Vulnerabilities

Signed/Unsigned Conversions

- example 1 — vulnerable because:
  - no validation of f
  - signed f is converted to a large unsigned int, leading to buffer overflow

```
int copy (char *dst, char *src, unsigned int len)
{
  while (len--)
    *dst++ = *src++;
}

int f = -1;
copy (d, s, f);
```

- lesson learned
  - never let negative (“signed int”) numbers go into libc functions that use “size_t”, which is an “unsigned int”
  - examples of such functions: read, snprintf, strncpy, memcpy, strncat, malloc

Signed/Unsigned Conversions (cont.)

- example 2 — vulnerable because:
  - wrong validation of len
  - could lead to buffer overflow, when len is negative

```
int len, sockfd, n;
char buf[1024];

len = get_user_len(sockfd);

if (len < 1024)
  read (sockfd, buffer, len); // len converted to "unsigned
       int"
```

- lesson learned
  - **never use signed variables for sizes**
  - if signed variables are used, **check also if positive**, besides checking for upper limits

Sign Extension

- in certain cases sign extension is a value-changing conversion with unexpected results
  - when converting from a smaller signed type to a larger unsigned type
- example of vulnerable code for both initial and patched versions

```
char len;

len = get_len();
// snprintf(dst, len, "%s", src); // initial: bad for
    negative len
snprintf(dst, (unsigned int)len, "%s", src); // solution: bad
     due to sign extension
```

Sign Extension (cont.)

- do not forget that “char” and “short” are signed
- vulnerable example (var 1): no max limit checked for count

```
char *indx;
int count;
char nameStr[MAX_LEN]; // 256
...
memset(nameStr, 0, sizeof(nameStr));
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx;

while (count) {
  (char*)indx++;
  strncat(nameStr, (char*)indx, count);
  indx += count;
  count = (char) *indx;
  strncat (nameStr, ".", sizeof(nameStr) - strlen(nameStr));
}
nameStr[strlen(nameStr)-1] = 0;
```

Sign Extension (cont.)

- vulnerable example (var 2): no check for negative count, converted to “unsigned int” due to return type of “strlen(nameStr)”

```
...

while (count) {
  if (strlen(nameStr) + count < (MAX_LEN -1)) { // pass for
        5 + (-1) = 4, due to overflow
    ...
    strncat(nameStr, (char*)indx, count);  // count taken as
         a huge unsigned no
    ...
  }
}
nameStr[strlen(nameStr)-1] = 0;
```

Sign Extension (cont.)

- vulnerable example (var 3): all casts superfluous, so same as previous

```
...

while (count) {
  if ((unsigned int)strlen(nameStr) + (unsigned int) count <
       (MAX_LEN -1)) { // pass for 5 + (-1), due to overflow
    ...
    strncat(nameStr, (char*)indx, count);
    ...
  }
}
nameStr[strlen(nameStr)-1] = 0;
```

Sign Extension (cont.)

- vulnerable example (var 4): due to the explicit (char) typecast

```
unsigned char *indx;
unsigned int count;
unsigned char nameStr[MAX_LEN];
...
indx = (char*) (pkt + tt_offset);
count = (char) *indx;   // this is still vulnerable to negative no.

while (count) {
  if (strlen(nameStr) + count < (MAX_LEN -1)) { // does not pass initially,
       when strlen() is 0
    indx++;
    strncat(nameStr, indx, count);
    indx += count;
    count = *indx;
    strncat (nameStr, ".", sizeof(nameStr) - strlen(nameStr));
  } else { die("error"); }
}
nameStr[strlen(nameStr)-1] = 0; // writes at nameStr[-1]
```

Sign Extension (cont.)

- example of signed extension vs. no sign extension
  - the C code

```c
// case 1
unsigned int no;
char c=5;
no = c;
```

```c
// case 2
unsigned int no;
unsigned char c;
no =  c;
```

  - the resulted assembly code

```
// case 1
mov [ebp+var_5], 5
movsx eax, [ebp+var_5]

mov [ebp+var_4], eax
```

```
// case 2
mov [ebp+var_5], 5
xor eax, eax
mov al, [ebp+var_5]
mov [ebp+var_4], eax
```

  - audit hint: look for movsx instruction in assembly code (sign extension)

Truncation

- larger type converted into a smaller one as a result of an assignment, type cast or function call
- truncation example

```
int g = 0x12345678;
short int h;
h = g; // h = 0x5678;
```

Truncation (cont.)

- vulnerability example: return of strlen is “size_t”, which will be truncated to a “short int”

```
unsigned short int f;
char mybuf[1024];
char *userstr = getuserstr();

f = strlen(userstr); // f get 464 for a strlen of
     66,000
if (f < sizeof(mybuf) - 5) // pass for strlen of
    66,000
  die ("string too long");
strcpy(mybuf, userstr);
```

Comparisons

- comparing integers of different types (widths)
- due to integer promotion, which could lead to value changing
- vulnerability example: due to len being promoted to a signed integer, then unsigned integer (e.g. len = 1, len = -1)

```
int read_pkt(int sockfd)
{
  short len;
  char buf[MAX_SIZE];

  len = newtwork_get_short (sockfd);

  if (len - sizeof(short) <= 0 || len > MAX_SIZE) { // first condition always true
    error ("bad length supplied");
    return -1;
  }

  if (read(sockfd, buf, len - sizeof(short)) < 0) {
    error ("read");
    return -1;
  }

  return 0;
}
```

Conclusions

- integer overflow / underflow could lead to application undetermined behaviour
- they often lead to buffer overflow vulnerabilities
- low-level languages (C/C++) most vulnerable, but most languages affected
- type conversion is a particular aspect of integer overflow
  - integer promotion – truncation
  - etc.

Recommendation for Code Developers

- check all (user controlled) application's inputs before using them
- recheck the math that manipulates input numbers
- do not use signed integers as unsigned parameters
- write clear code, not using "smart" tricks
- annotate the code with the exact casts that happen in an operation (just to understand clearly the results)
- use safe types, when possible (e.g. SafeInt https://safeint.codeplex.com/)
- activate useful (ALL) compiler warnings regarding type mismatch
  - Visual Studio: -W4
  - gcc: -Wall, -Wsign-compare, -ftrapv

Recommendation for Code Auditors (Reviewers)

- monitor all application’s inputs
- look for places that write into buffers
- look for explicit casts on input numbers or numbers influenced by inputs
- check the math that manipulates input numbers
- use, if possible, static analysis tools

Bibliography

- “The Art of Software Security Assessments”, chapter 6, “C Language Issues”, pp. 203 – 296
- “The 24 Deadly Sins of Software Security”, Sin 7. Integer Overflows, pp. 119 – 142
- DataRepresentation, http://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html