Purpose and Contents
Introduction
C Basics. Data Representation
Arithmetic Boundary Conditions
Type Conversions
Arithmetic Operations / Operators
Conclusions
Bibliography

Implementation Sins

C Language Issues

Adrian Colesa

Universitatea Tehnică din Cluj-Napoca
Computer Science Department

October 12, 2015

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

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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
  - 24^{th} place in Mitre's Top 25
  - `http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf`
- CWE-191: Integer Underflow (Wrap or Wraparound)
- CWE-192: Integer Coercion Error

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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 tuned into a buffer overflow
- all languages are prone to DoS and logic errors

Actual Relevance

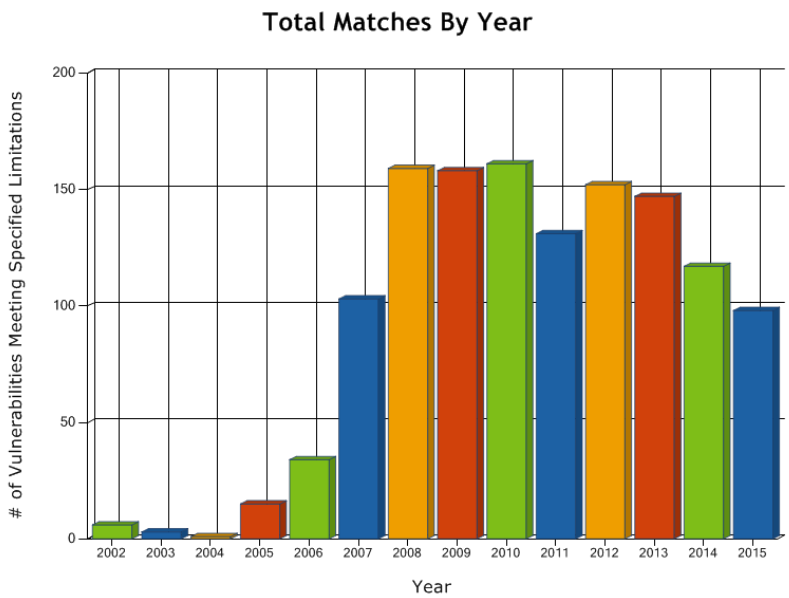

Figure: Number of Integer Overflow Vulnerabilities

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Actual Relevance (cont.)

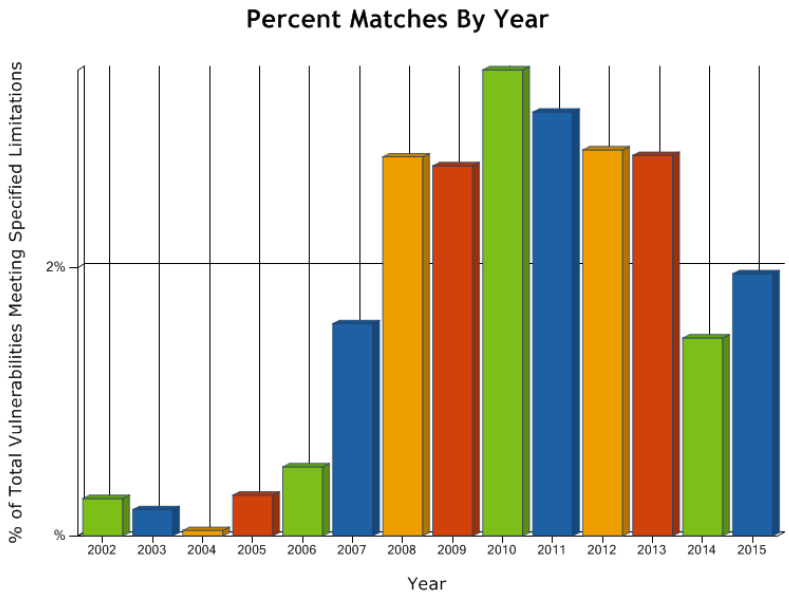

Figure: Percentage of Integer Overflow Vulnerabilities

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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*

Types (cont.)

- 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 lomg	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

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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
    - - addition, 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)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Binary Encoding (cont.)

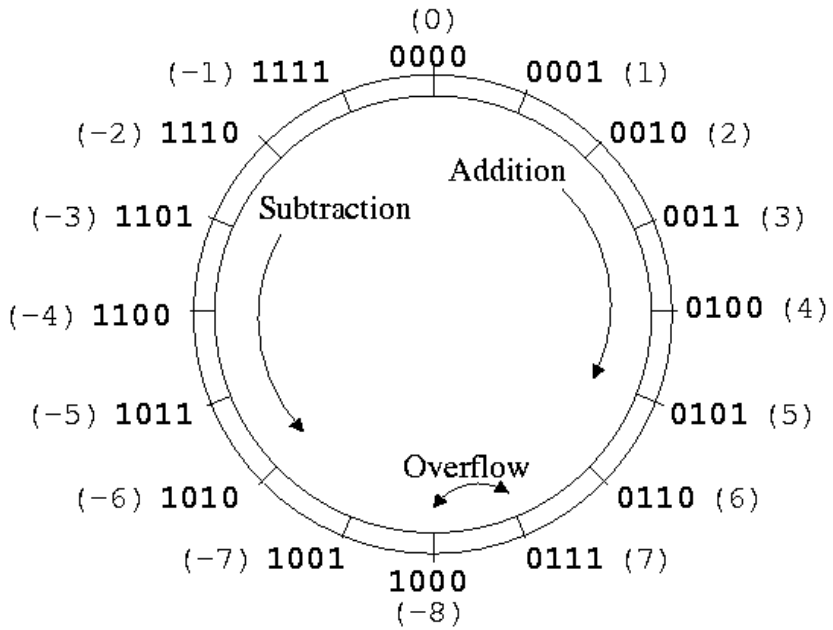

Figure: Two Complement Representation

Byte Order

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

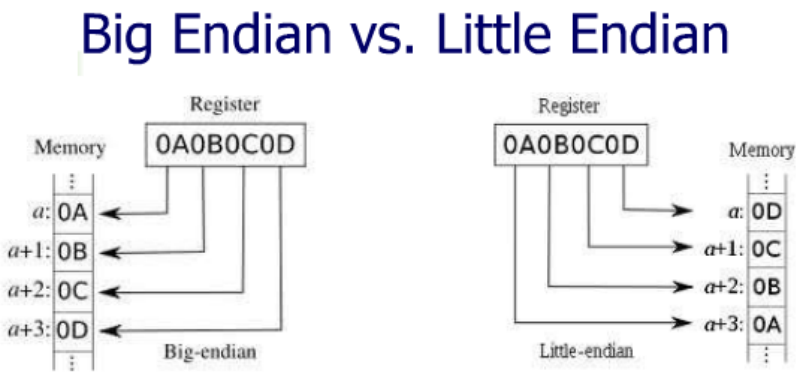

Figure: Big vs Little Endian Representation

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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;
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Context and Definitions (cont.)

- 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 $\Rightarrow$
  - 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

Security Risks of Integer Overflow / Underflow (cont.)

- bad lengths / limits calculated for memory allocation $\Rightarrow$ buffer overflow
- bad length / limit checking $\Rightarrow$ buffer overflow

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Unsigned Integer Overflow

- operations are subject to the rules of modular arithmetic
  - result is “real result” module (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

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Unsigned Integer Overflow Vulnerability 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);
}
```

Unsigned Integer Overflow Vulnerability Example (cont.)

- *n* could be overflowed by multiplication of user-controlled *height* and *width*, resulting in a relatively small number
  - example:
    $0x400 * 0x10000001 = 0x400$
    $1024 * 268435457 = 1024$
- still, the *for* loop goes for a large portion of (overflowed) memory
  - example: 1024 bytes allocated $\Rightarrow$
    1 element allocated BUT
    more elements accessed

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Unsigned Integer Overflow 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();
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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

Unsigned Integer Underflow Vulnerability 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;
```

Unsigned Integer Underflow Vulnerability Example (cont.)

```
    }

    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);
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Unsigned Integer Underflow Vulnerability Example (cont.)

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Signed Integer Overflow and Underflow

- overflow could results 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 1

```
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) {
    free(buf);
    die("bad length");
  }

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

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

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Signed Integer Vulnerability Example 1 (cont.)

```
}
```

- read signature

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- in practice read could check if *nbytes* less than the maximum signed value

Signed Integer Vulnerability Example 2

```
char* extend_heap(char *p, int crt_len, int add_size)
{
  char *new_p = p;

  if (add_size > 0) {
    if (crt_len + add_size > crt_len)
      new_p = realloc(p, crt_len + add_size);
  }

return new_p;
}
```

Signed Integer Vulnerability Example in OpenSSL 0.9.6

```
c.inf=ASN1_get_object(&c.p, &c.slen, &c.tag. &c.xtag, len -
    off);

want = (int)c.slen;
if (want > (len - off)) {
  want -= (len - off);
  if (!BUF_MEM_grow(b, len + want))
    ASN1_die_err("...");
  i = BIO_read(in, &b->data[len], want);
}
```

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

Definition and Context

- conversion of an object of one type to another type
- explicit type conversion
- implicit (default) type conversion

Conversion Rules for Integers

- cases
  - value-preserving vs. value-changing
    - new type can represent (or not) all possible values of the old type
  - widening from narrow to wider type
    - *zero-extension* is used for unsigned numbers
    - *sign-extension* is used for signed numbers

Conversion Rules for Integers (cont.)

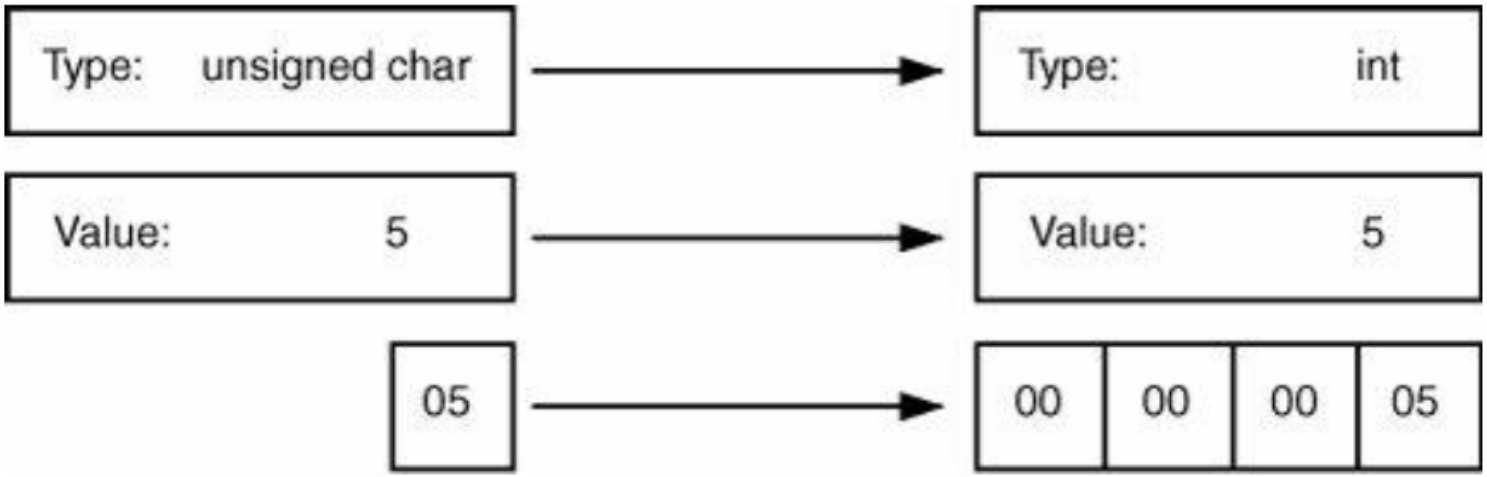

Figure: value preserving conversion: “unsigned char” to “signed int”

Conversion Rules for Integers (cont.)

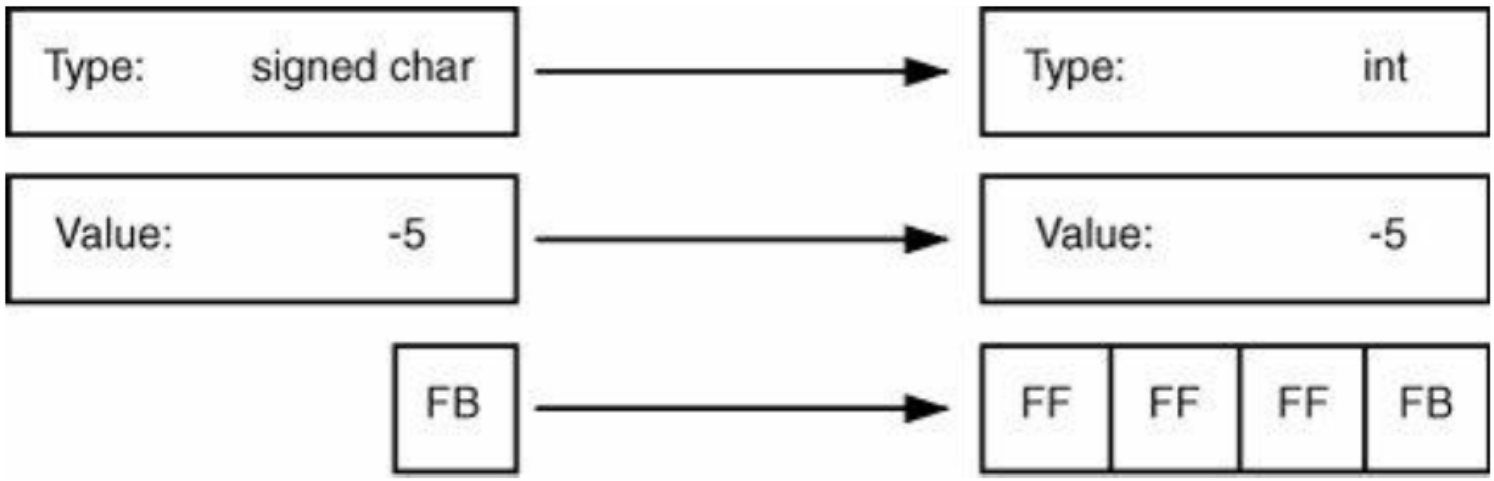

Figure: value preserving conversion: “signed char” to “signed int”

Conversion Rules for Integers (cont.)

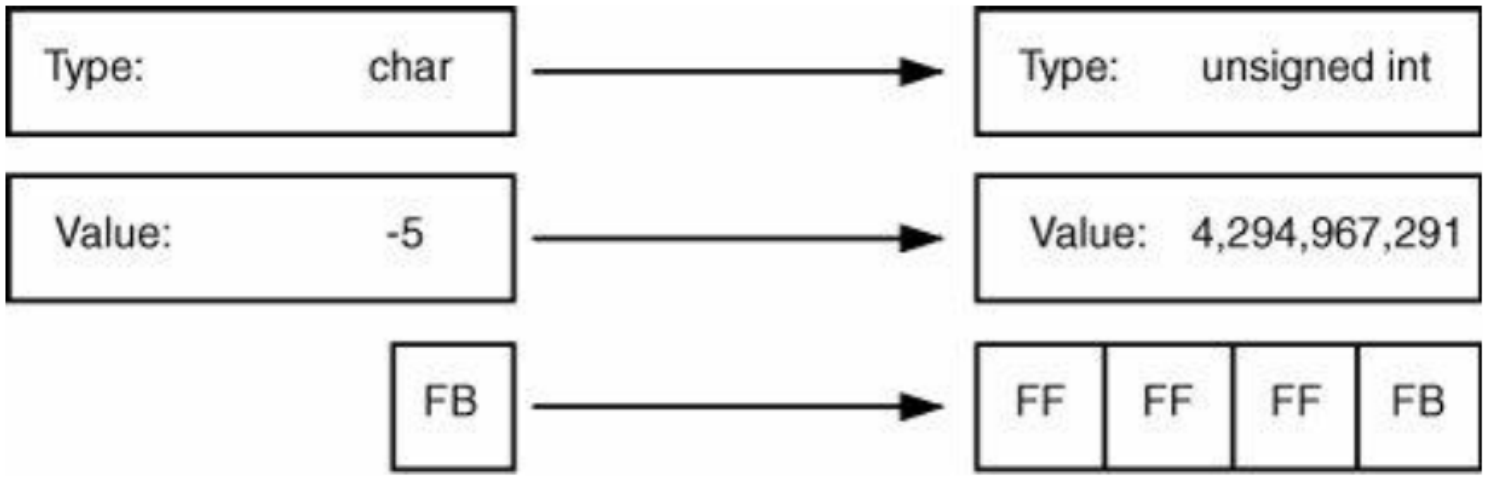

Figure: value changing conversion: “signed char” to “unsigned int”

Conversion Rules for Integers (cont.)

- narrowing: uses truncation (is value-changing)

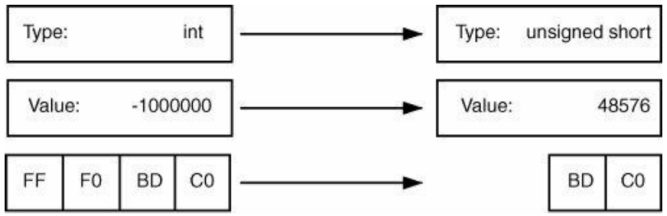

Figure: truncation: “signed int” to “unsigned short”

Conversion Rules for Integers (cont.)

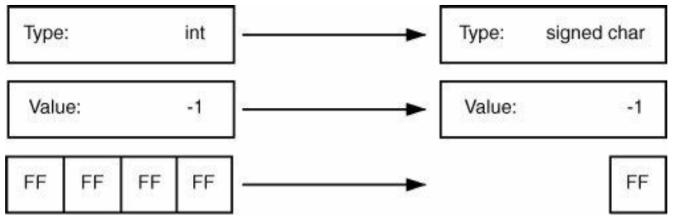

Figure: truncation: “signed int” to “signed char”

Conversion Rules for Integers (cont.)

- conversion between signed and unsigned (is value-changing)

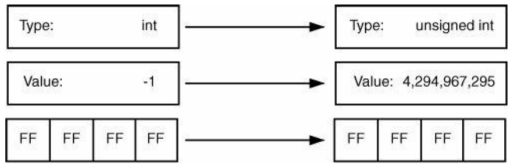

Figure: truncation: “signed int” to “unsigned int”

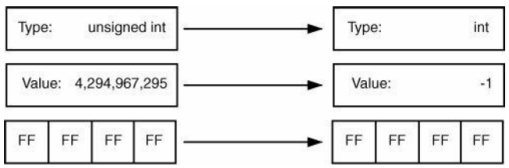

Figure: truncation: “unsigned int” to “signed int”

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Conversion Rules for Integers (cont.)

- rules
  - narrower signed $\rightarrow$ wider unsigned: sign extension $\Rightarrow$ value-changing
  - narrower signed $\rightarrow$ wider signed: sign extension $\Rightarrow$ value-preserving
  - narrower unsigned $\rightarrow$ wider (any): zero extension $\Rightarrow$ value-preserving
  - wider (any) $\rightarrow$ narrower (any): truncation $\Rightarrow$ value-changing
  - signed $\leftrightarrow$ unsigned (of the same width): bits preserved, but the value is otherwise interpreted $\Rightarrow$ value-changing

Conversion Rules for Floating Point and Complex Types

- real $\rightarrow$ integer: fractional part is discarded
- integer $\rightarrow$ real: round up or down is performed if needed
- result is undefined when source range greater than destination range
- conversions between floating point types
  - promotion causes no change in value
  - demotion can cause value changing (rounding numbers is possible)

Simple Conversions

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

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

- function calls
  - prototypes-based

Simple Conversions (cont.)

```
int dostuff(int jim, unsigned char bob);

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

- return-based

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

Simple Conversions (cont.)

Integer Promotions (Widening Conversions to Int)

- narrower integer type $\rightarrow$ int
- used for
  - certain operators require an integer operand
  - handling of 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

Integer Promotions (Widening Conversions to Int) (cont.)

- 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 $\Rightarrow$ 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

Integer Promotion Applied (cont.)

```
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

Integer Promotion Applied (cont.)

- function invocations perform *default argument promotion* in cases the function declaration does not exists or does not specify argument types (like for the K&R semantics)

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
  - $\Rightarrow$ 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

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Signed/Unsigned Conversions

- example 1: *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);
```

- **warning 1**

Signed/Unsigned Conversions (cont.)

- 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*

- example 2: 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"
```

Signed/Unsigned Conversions (cont.)

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 maximum 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));
```

Sign Extension (cont.)

```
}
nameStr[strlen(nameStr)-1] = 0;
```

- vulnerable example (var 2): no checked for negative count, converted to “unsigned int” due to “strlen(nameStr)” result

```
...

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;
```

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

Sign Extension (cont.)

```
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.)

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

Truncation

- a 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;
```

- vulnerability example 1 (NFS)

Truncation (cont.)

```
void assume_privs(unsigned short uid)
{
  seteuid(uid);
  setuid(uid);
}

int become_user(int uid)
{
 if (uid == 0)  // 65536 would pass
  die ("root not allowed");

 assume_privs (uid); // 65536 would get 0
}
```

- vulnerability example 2: return of *strlen* is "`size_t`", which will be truncated to a "short int"

Truncation (cont.)

```
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 an signed integer, then unsigned integer (e.g. len = 1)

Comparisons (cont.)

```
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;
}
```

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The *sizeof* Operator

- misuse: use it for pointers instead of referenced data
- comparison of *sizeof* result (unsigned) with signed numbers

Unexpected Results

- right shift, division, modulo on negative numbers could generate big (unsigned) numbers
- modulo operation is often used when dealing with fixed-size arrays (e.g. hash tables), so a negative result could index before the beginning of the array
- difficult to locate on the source code
- better to look after certain assembly code instructions
  - signed *sar* vs. unsigned *shr*
  - signed *idiv* vs. unsigned *div*

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Overview

- very architecture and compiler dependent
- how is the result of two pointer subtraction interpreted?
- pointers could be converted into (signed) integers
- the compiler makes no guarantee that the resulting pointer or integer is correctly assigned or points to a valid object

Pointer Arithmetic

- general
  - operations are done relative to the size of the pointer's target (type)
  - pointers are similar with arrays
- addition
  - can add an integer to a pointer
  - cannot add a pointer to a pointer (which to use as a base, which as an index?)
  - subscript operator falls under the category of pointer addition
- subtraction

Pointer Arithmetic (cont.)

- similar with addition
- subtracting one pointer from another is allowed!
- the resulting type is not a pointer, but a `ptrdiff_t` (signed integer)

- comparison: works like usual
- conditional operator
  - could have pointers as the last two operands
  - has to reconcile their types like in case of arithmetic operands

Vulnerabilities

- generally, wrongly interpret pointer arithmetic
- example (passed over the array limit)

```
int buf[1024];
int *b = buf;

while (havedata() && b < buf + sizeof(buf)) {
  *b++ = parseint(getdata());
}
```

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Order of Evaluation

- compiler dependent
- there are no guarantees related to the order operators are evaluated, excepting `&&`, `||`, `?:`, and `,`
- ambiguous side effects

Structure Padding

- padding could be used to align different structure members
- example

```
struct ex {
 int a;
 unsigned short b;
 unsigned char c;
};
```

- vulnerability example

Structure Padding (cont.)

```
    struct netdata {
      unsigned int query_id;
      unsigned short header_flags;
      unsigned int seq_no;
    };

    int packet_check (unsigned char *buf, size_t len)
    {
struct netdata *n = (struct netdata*) buf;

if (ntohl(n->seq_no) <= last_seq)
  return PARSE_REPLAYATTACK;

return PARESE_SAFE;
    }
```

Precedence

- example 1

```
if (len & 0x80000000 != 0)
  if (len < 1024)
    memcpy(dst, src, len);
```

- example 2

```
if (len = getlen() > 30)
  snprintf(dst, len - 30, "%s", src);
```

- example 3

```
int a = b + c >> 3;
```

Macros/Preprocessor

- example 1 (bad evaluation)

```
#define SQUARE(x) x*x
y = SQUARE(z + t);
```

- example 2

```
#define SQUARE(x) ((x)*(x))
y = SQUARE(j++);
z = SQUARE(getint());
```

Outline

1. Introduction
2. C Basics. Data Representation
3. Arithmetic Boundary Conditions
   - Overview
   - Unsigned Integer Boundaries
   - Signed Integer Boundaries
4. Type Conversions
   - Overview
   - Type Conversion Vulnerabilities
5. Arithmetic Operations / Operators
   - Operators
   - Pointer Arithmetic
   - Other C Nuances
6. Conclusions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Brief Review

- integer overflow / underflow could lead to application undetermined behavior
- 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.

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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/`)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Recommendation for Code Developers (cont.)

- activate useful (ALL) compiler warnings regarding type mismatch
  - Visual Studio: -W4
  - gcc: -Wall, -Wsign-compare, -ftrapv

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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

1. "The Art of Software Security Assessments", chapter 6, "C Language Issues", pp. 203 – 296
2. "The 24 Deadly Sins of Software Security", Sin 7. Integer Overflows, pp. 119 – 142
3. Data Representation, `http://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA