Fundamentos del Software: introducción a la programación verificada

Programación segura

Enteros

Roberto Blanco[†] & Ricardo J. Rodríguez[‡]

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Universidad Zaragoza

[†]Max Plank Institute for Security and Privacy
Bochum, Alemania

[‡]Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza (España)

Septiembre 2020

Universidad de Zaragoza

Enteros

- **Creciente fuente de vulnerabilidades en programas C/C++**
- **Las comprobaciones de rango en los enteros no se ha aplicado de manera sistemática** durante el desarrollo de software
  - Todavía existen errores relacionados con los enteros
  - Algunos de ellos pueden convertirse en vulnerabilidades
- **Fallos costosos y explotables**
- Parte del **top 25 de errores más peligrosos** (MITRE 2011)

Lecturas de interés:

- Dietz, W.; Li, P.; Regehr, J. & Adve, V. “Understanding Integer Overflow in C/C++”. *Proceedings of the 34th International Conference on Software Engineering*, pp. 760–770, IEEE, 2012
- Brumley, D.; Song, D. X.; Chiueh, T.; Johnson, R. & Lin, H. “RICH: Automatically Protecting Against Integer-Based Vulnerabilities”. *Proceedings of the Network and Distributed System Security Symposium*, 2007

Universidad Zaragoza

Enteros

- **Errores numéricos relacionados con enteros**
  - Desbordamientos, subdesbordamientos, truncamientos con pérdida, problemas con conversiones

Comportamientos indefinidos

- **Conversión a/desde un tipo de dato que produce un valor fuera del rango representable**
- **Conversión de un puntero a un tipo de datos que produce un valor fuera del rango representable**
- **El valor del resultado de una operación aritmética o de conversión que no puede representarse**

Universidad Zaragoza

Enteros

```c
//Try both and see to believe:
//gcc foo.c -o foo
//gcc foo.c -o foo -O2
#include <stdio.h>
#include <limits.h>

int foo (int x){
    return ( x+1 ) > x;
}

int main()
{
    printf ("%d\n", ( INT_MAX+1 ) > INT_MAX );
    printf ("%d\n", foo ( INT_MAX ));
    return 0;
}
```

Universidad Zaragoza

Índice

Universidad Zaragoza

Tipos de datos enteros

- Conjunto finito sobre $\mathbb{Z}$
- El valor del objeto es el **valor matemático asociado a dicho objeto**
- Depende de la codificación particular de los bits almacenados en ese espacio de memoria reservado

Universidad Zaragoza

Tipos de datos enteros

- Conjunto finito sobre $\mathbb{Z}$
- El valor del objeto es el **valor matemático asociado a dicho objeto**
- Depende de la codificación particular de los bits almacenados en ese espacio de memoria reservado

- Los tipos de enteros en C incluyen **los tipos enteros que han existido desde los principios de C**
  - **Conjunto de datos enteros con y sin signo**
  - En C99 se añadió el tipo booleano (sin signo)
- **Existe una correspondencia muy cercana a la arquitectura subyacente de la máquina**

Universidad Zaragoza

Tipos de datos enteros

- **Cada tipo de dato en C tiene un número fijo de bytes para su almacenamiento**
- La constante `CHAR_BIT` (en `<limits.h>`) **almacena el número de bits en un byte**
  - **Por lo menos 8**, aunque puede ser mayor dependiendo de las implementaciones específicas
- **No todos los bits se usan para representar el valor. Los que no se usan se llaman *padding***
  - Los que se usan se llaman **anchura** del tipo de datos
- Precisión del valor: depende del número de bits usados como anchura
  - Tipos enteros con signo: anchura – 1 (hay un bit de signo)
  - Tipos enteros sin signo: anchura

Universidad Zaragoza

Tipos de datos enteros

Enteros sin signo

Representación de valores positivos ($\mathbb{Z}_{\geq 0}$)

- `unsigned char`
- `unsigned short int`
- `unsigned int`
- `unsigned long int`
- `unsigned long long int`
  - **Lista en orden creciente**
  - Los tipos `long long` se definieron en C99, pero no en ISO/IEC 14882:2003 (C++). **Están definidos en C++11 and y otras implementaciones**
- Límites de tamaño por plataforma/compilador en cabecera <`limits.h`>
  - Conviene usar las constantes por portabilidad, en vez de valores particulares

Universidad Zaragoza

Tipos de datos enteros

Enteros sin signo

Constante	Magnitud mínima (C11)	x86-32	Máximo valor para
`UCHAR_MAX`	255 ($2^8 - 1$)	same	`unsigned char`
`USHRT_MAX`	65535 ($2^{16} - 1$)	same	`unsigned short int`
`UINT_MAX`	65535 ($2^{16} - 1$)	4294967295	`unsigned int`
`ULONG_MAX`	4294967295 ($2^{32} - 1$)	same	`unsigned long int`
`ULLONG_MAX`	18446744073709551615 ($2^{64} - 1$)	same	`unsigned long long int`

- **El valor mínimo es siempre 0** (no hay constante definida)

Universidad Zaragoza

Tipos de datos enteros

Enteros con signo

- Representación de **valores positivos y negativos** ($\mathbb{Z}$)
  - `signed char`
  - `signed short int`
  - `signed int`
  - `signed long int`
  - `signed long long int`
- Puede omitirse excepto para `char` y `signed`
- `int` puede también omitirse, a menos que sea el único tipo presente

Universidad Zaragoza

Tipos de datos enteros

Enteros con signo

- **El signo se almacena en el bit de orden más alto, e indica si el valor representado es negativo**
  - 0 es positivo (e.g., 00101001 $\rightarrow +41$)
  - 1 es negativo (e.g., 10101001 $\rightarrow -41$)

Tipos de datos enteros

Enteros con signo

- **El signo se almacena en el bit de orden más alto, e indica si el valor representado es negativo**
  - 0 es positivo (e.g., 00101001 $\rightarrow +41$)
  - 1 es negativo (e.g., 10101001 $\rightarrow -41$)
- **Diferentes formas de representación interna**:
  - **Signo y magnitud**
  - **Complemento a uno**
  - **Complemento a dos**
- Normalmente, complemento a dos es lo más usado en entornos de escritorio

Universidad Zaragoza

Tipos de datos enteros

Rangos de los enteros con signo

Constante	Magnitud mínima (C11)	x86-32	Máximo valor para
SCHAR_MIN	-127 $(-(2^7-1))$	-128	`signed char`
SCHAR_MAX	127 (2^7-1)	(igual)	`signed char`
SHRT_MIN	-32767 $(-(2^{15}-1))$	-32768	`short int`
SHRT_MAX	32767 $(2^{15}-1)$	(igual)	`short int`
INT_MIN	-32767 $(-(2^{15}-1))$	-2147483648	`int`
INT_MAX	32767 $(2^{15}-1)$	$+2147483647$	`int`
LONG_MIN	-2147483647 $(-(2^{31}-1))$	-2147483648	`long int`
LONG_MAX	$+2147483647$ $(2^{31}-1)$	(igual)	`long int`
LLONG_MIN	-9223372036854775807 $(-(2^{63}-1))$	-9223372036854775808	`long long int`
LLONG_MAX	$+9223372036854775807$ $(2^{63}-1)$	(igual)	`long long int`

- **La anchura mínima obligatoria en C11 para estos tipos es**:
  - `signed char` (8 bits), `short` (16), `int` (16), `long` (32), and `long long` (64)
- **El tamaño de los tipos puede saberse con el operador `sizeof`**
  - Incluye bits de padding (si alguno)

Universidad Zaragoza

Índice

Universidad Zaragoza

Errores más comunes

- **Desbordamiento**
  - Ocurre en tiempo de ejecución con los enteros con signo, cuando el resultado de una expresión entera excede el valor del tipo
  - Por ejemplo, dos enteros sin signo de 8 bits pueden necesitar 16 bits
- **Subdesbordamiento**
  - Ocurre en tiempo de ejecución con los enteros sin signo, cuando el resultado de una expresión entera es más pequeño que el mínimo valor del tipo
  - Por ejemplo, restar $0 - 1$ y almacenar el valor en un entero de 16-bits (sin signo) resultará en el valor $2^{16} - 1$ (no -1)
- **Truncamiento**
  - Ocurre en tiempo de ejecución cuando se asigna un entero con un tamaño más grande a un entero que tiene menos tamaño
  - Por ejemplo, castear un `int` a un `short` descarta los bits más altos del `int`, pudiendo perder información por el camino
- **Problemas de signo**
  - Ocurre cuando un entero con signo se interpreta como sin signo, o viceversa
  - En representación de complemento a 2, las conversiones pueden suponer que el bit de signo se interprete como valor (i.e., $2^{32} - 1 \neq -1$)

Universidad Zaragoza

Errores más comunes

Ejemplo

```
struct pixmap {
  unsigned char *p;
  int x;
  /* xsize */
  int y;
  /* ysize */
  int bpp;
};
typedef struct pixmap pix;
......
void readpgm(char *name, pix * p) {
   /* read pgm */...
   pnm_readpaminit(fp, &inpam);
  p->x=inpam.width;
  p->y=inpam.height;
   if(!(p->p=(char *)malloc(p->x*p->y)))
     F1("Error at malloc");
   for(i=0; i<inpam.height; i++){
     pnm_readpamrow(&inpam, tuplerow);
   for(j = 0; j<inpam.width; j++)
     p->p[i*inpam.width+j]=sample;
  }
}
```

```
void getComm(unsigned int len, char *src){
   unsigned int size;
   size = len - 2;
   char *comm = (char *)malloc(size + 1);
   memcpy(comm, src, size);
   return;
}
```

```
static inline u32 *decode_fh(u32 *p, struct svc_fh *fhp) {
   int size;
   fh_init(fhp, NFS3_FHSIZE);
   size = ntohl(*p++);
   if (size > NFS3_FHSIZE)
      return NULL;
   memcpy(&fhp->fh_handle.fh_base, p, size);
   fhp->fh_handle.fh_size = size;
   return p + XDR_QUADLEN(size);
}
```

```
int detect_attack(u_char *buf, int len, u_char *IV){
   static word16 *h = (word16 *) NULL;
   static word16 n = HASH_MIN_ENTRIES;
   register word32 i, j;
   word32 l;
   ...
   for(l=n; l<HASH_FACTOR(len/BSIZE); l=l<<2);
   if (h == NULL) {
      debug("Install crc attack detector.");
      n = l;
      h = (word16 *) xmalloc(n*sizeof(word16));
   } else
      for (c=buf, j=0; c<(buf+len); c+=BSIZE, j++){
         for (i = HASH(c) & (n - 1); h[i] != UNUSED; i = (i + 1) & (n - 1))
            ...;
         h[i] = j;
      }
   }
```

Errores más comunes

Normalmente, se explotan de manera indirecta

- **Ejecución de código arbitraria**
  - Por ejemplo, reserva de memoria insuficiente provocada por un desbordamiento de buffer, ataques de sobreescritura, etc.
- **Denegación de servicio**
  - Por ejemplo, excesiva reserva de memoria o bucles infinitos
- **Evitación de mecanismos de sanitización**
  - Por ejemplo, evitar una comprobación de cotas superiores que no espera números negativos
- **Errores lógicos**
  - Por ejemplo, manipular el contador de referencia forzando que un objeto se libere de forma temprana

Universidad Zaragoza

Errores más comunes

Consecuencias

- **Rotura silenciosa**
  - Las optimizaciones del compilador pueden resultar en comportamiento no definido
- **Bombas de tiempo**
  - Hoy funciona, pero mejoras en algoritmos de optimización pueden llegar a explotarla
- **Ilusión de predictibilidad**
  - Algunos compiladores, con determinados niveles de optimización, tienen comportamiento predecible para algunas operaciones no definidas
- **Dialectos informales**
  - Algunos compiladores permiten forzar el comportamiento de complemento a 2 en desbordamientos
- **Estándares no estándares**
  - El significado de un desbordamiento cambia entre estándares (e.g., $1 << 31$)
    - Definido en implementación en ANSI C y C++98
    - Indefinido en C99 en C11

Universidad Zaragoza

Wraparound

- **Resultado de operaciones aritméticas con enteros sin signo cuyo valor es demasiado pequeño (menor que 0) o demasiado grande (supera la representación)**
- *¿Qué ocurre cuando el resultado de una operación no se puede representar mediante el tipo de entero sin signo?*
  - El resultado se reduce al **módulo del número mayor que puede ser representado**
- Por ejemplo, con **operaciones de suma y multiplicación**
  - Sumar/restar un número de n-bits requiere $n + 1$ bits de precisión
  - Multiplicar un número de n-bits requiere $2n$ bits de precisión

Universidad Zaragoza

Wraparound

Ejemplo en x86-32

```c
unsigned int ui;

ui = UINT_MAX; // 4294967295 on x86-32
ui++;
printf("ui = %u\n", ui); // prints 0

ui = 0;
ui --;
printf("ui = %u\n", ui); // prints 4294967295
```

Wraparound

- **Una expresión entera sin signo nunca se evaluará a algo menor que cero, debido al wraparound**
- O sea, **es posible codificar expresiones tal que siempre sean ciertas** (o falsas)
  - Por ejemplo: `for (unsigned i = n; --i >= 0; )`

Universidad Zaragoza

Wraparound

- **Una expresión entera sin signo nunca se evaluará a algo menor que cero, debido al wraparound**
- O sea, **es posible codificar expresiones tal que siempre sean ciertas** (o falsas)
  - Por ejemplo: `for (unsigned i = n; --i >= 0; )`
- **Errores de codificación**: *¿qué ocurrirá cuando el máximo valor del contador de eventos de tu programa se alcance?*
- Ejemplo real: 25 de diciembre, 2004, **Comair (South Africa) tuvo que parar todas sus operaciones y hacer aterrizar 1100 vuelos después de un fallo en su software de planificación de tripulaciones**
  - Fallo provocado por un contador de 16-bit que limitaba el número de cambios a 32768 en cualquier mes
  - El mal tiempo de ese mes provocó un montón de reasignaciones, superándose el valor máximo del tipo de dato

Universidad Zaragoza

Wraparound

Portablilidad

- **Si se necesita asegurar la portabilidad, pueden usarse los tipos de datos donde se especifica el tamaño** (como `uint32_t` de <`stdint.h`>)
- **Si no, la anchura del tipo de dato usado para el wraparound depende de la implementación** (o sea, puede haber comportamientos diferentes según sea la plataforma)

Wraparound

Operadores que producen wraparound

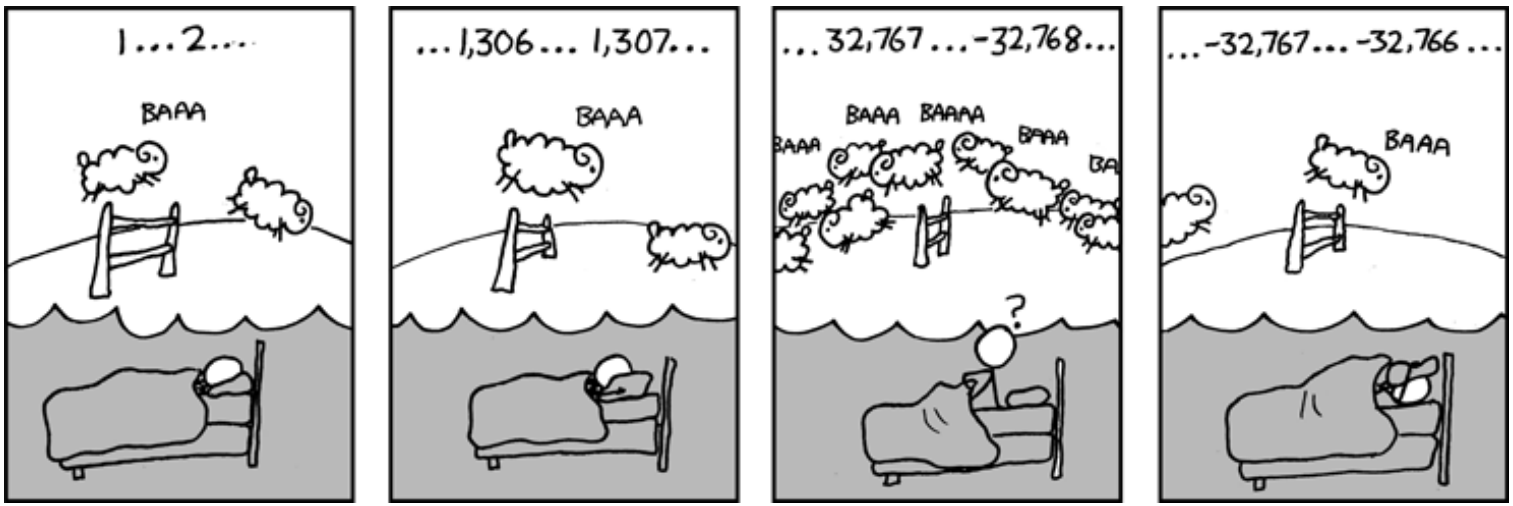
- **Suma**: +, +=, ++
- **Resta**: -, -=, --, unary -
- **Multiplicación**: *, *=
- **Desplazamiento (a la izquierda)**: <<, <<=

Regla INT30-C: *Ensure that unsigned integer operations do not wrap*

(https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap)

Universidad Zaragoza

Desbordamiento



- **Ocurre con tipos enteros con signo, cuando el resultado no puede ser representado en el tipo de dato**
- **Comportamiento no definido en C**: algunas implementaciones hacen *wrapping* (lo más común), otras rompen ejecución
  - Se asume que el wrapping es un comportamiento especificado, lo cual es falso

Desbordamiento

Ejemplo en x86-32

```c
int si;

si = INT_MIN; // -2147483648
si--;
printf("si = %d\n", si); // si equals 2147483647

si = INT_MAX; // 2147483647
si++;
printf("si = %d\n", si); // si equals -2147483648
```

Desbordamiento

Operadores que producen desbordamiento

- **Suma**: +, +=, ++
- **Resta**: -, -=, --, `unary -`
- **Multiplicación**: *, *=
- **División y módulo**: /, /=, %, %=
- **Desplazamiento (a la izquierda)**: <<, <<=

Regla INT32-C: *Ensure that operations on signed integers do not result in overflow*

(`https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow`)

Desbordamiento

Desbordamiento en `176.gcc`, SPEC CINT2000

```
/* ( unsigned ) <= 0 x7fffffff is equivalent to >= 0. */
else if ( const_op == (( HOST_WIDE_INT ) 1 << (
        mode_width - 1)) - 1)
{
    const_op = 0, op1 = const0_rtx ;
    code = GE;
}
```

- `HOST_WIDE_INT` es un entero, `mode_width` tiene valor 32
- **`(1 << 31) - 1` está indefinido**
  - C99: es ilegal desplazar un bit “1” bit al bit de signo (o sobrepasarlo)
  - La resta también puede producir un subdesbordamiento

Desbordamiento

Desbordamiento en `IntegerLib`, librería para operaciones seguras con enteros del CERT

```
int addsi ( int lhs , int rhs ) {
    errno = 0;
    if ((( lhs+rhs )^ lhs ) &(( lhs+rhs )^ rhs ))
            >> ( sizeof (int)* CHAR_BIT -1)) {
        error_handler (" OVERFLOW ERROR ", NULL , EOVERFLOW );
        errno = EINVAL ;
    }
    return lhs +rhs;
}
```

- **Los argumentos se suman sin ningún chequeo previo**
- **Deberían de castearse a un tipo sin signo antes de sumarse**

Truncamiento

- **Ocurre en tiempo de ejecución cuando se asigna un entero con un tamaño más grande a un entero que tiene menos tamaños**
- **Pueden perderse datos**
  - Imagina por ejemplo la suma de `c1` y `c2`: se produce un valor (290) que es mayor que el tipo `unsigned char` (considerando que `unsigned char` tiene 8 bits)

```
unsigned char sum, c1, c2;

c1 = 200;
c2 = 90;
sum = c1 + c2;
```

Truncamiento

Operadores que producen truncamiento

Cualquier operador de asignación

=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=

Problemas de signo

- Cuando algunas operaciones aritméticas no tienen el mismo tipo (a ambos lados), se producen conversiones de tipo implícitos para conseguir un mismo tipo
- *Operadores*: *, /, %, +, -, <, >, <=, >=, ==, !=, &, ^, |, y el operador ternario ? :

Ejemplo de conversión en x86-32

```
unsigned int ui = UINT_MAX;
signed char c = -1;

if (c == ui) {
     printf("-1 = 4,294,967,295?\n");
}
```

Problemas de signo

Algunas notas sobre la conversión

- **Las conversiones implícitas simplifican la programación en C**
- **Pero cuidado: se pueden perder datos**
- Hay que evitar aquellas conversiones que produzcan:
  - **Pérdida de datos** (conversión a un tipo donde el valor no puede ser representado)
  - **Pérdida de signo** (conversión de un tipo con signo a uno sin signo)

Universidad Zaragoza

Índice

Universidad Zaragoza

Otros tipos de datos enteros

- **Definidos en C11** (<stdint.h>, <inttypes.h>, and <stddef.h>)

size_t

- **Definido en <stddef.h>. Elimina el problema de portabilidad**
- **Tipo de dato entero sin signo; resultado de `sizeof`**
- El límite del valor de `size_t` está definido en la constante `SIZE_MAX`
- **Es la manera más eficiente y portable de**:
  - **Declarar una variable para guardar un tamaño**: `size_t n = sizeof(thing)`
  - **Declarar un argumento de una función que recibe un tamaño**:
    ```
    void foo(size_t thing);
    ```
    - Útil para representar el número de elementos de un vector
    - Útil si la función realiza operaciones de desplazamiento sobre un vector

intmax_t, uintmax_t

- **Tipos de datos enteros con la mayor anchura posible**
- **Puede usarse para salidas por pantalla formateadas**:

```
printf("%ju", (uintmax_t) x);
```

Universidad Zaragoza

Índice

Universidad Zaragoza

Estrategias de mitigación
Evitando errores de wraparound

- **Comprobar si va a haber wraparound antes o después de la operación**
- Los límites definidos en <limits.h> son útiles, pero **hay que usarlos bien**:

```
unsigned i, j, sum;
if (sum + i > UINT_MAX)
    // Too big error
else
    sum += i;
```

Estrategias de mitigación

Evitando errores de wraparound

```
unsigned i, j, sum;
if (i   > UINT_MAX - sum)
    // Too big error
else
    sum += i;
```

- **Ojo! Recuerda que el valor mínimo es 0**:

```
unsigned j, sum;

if (sum - j < 0) // cannot happen
    // negative wraparound detected
else
    sum -= j;
```

```
unsigned j, sum;

if (j > sum) // correct
    // negative wraparound detected
else
    sum -= j;
```

Estrategias de mitigación

Forma correcta de wraparound `175.vpr`, SPEC CINT2000

```
# define IA 1103515245 u
# define IC 12345 u
# define IM 2147483648 u

static unsigned int c_rand = 0;

/* Creates a random integer [0... imax ] ( inclusive ) */
int my_irand ( int imax ) {
    int ival ;
    /* c_rand = ( c_rand * IA + IC) % IM; */
    c_rand = c_rand * IA + IC ; // Use overflow to wrap
    ival = c_rand & (IM - 1); /* Modulus */
    ival = ( int ) (( float ) ival * ( float ) ( imax + 0.999)
        / ( float ) IM);
    return ival ;
}
```

Estrategias de mitigación

Desbordamiento en operaciones matemáticas

Detección de desbordamiento en una operación con enteros con signo s_1 y s_2

- **Test de precondición**:

$$((s_1 > 0) \wedge (s_2 > 0) \wedge (s_1 > (\texttt{INT\_MAX} - s_2))) \vee$$
$$((s_1 < 0) \wedge (s_2 < 0) \wedge (s_1 < (\texttt{INT\_MAX} - s_2)))$$

- **Comprobar la flag *OF* de la CPU tras la operación**
- **Test de postcondición con extensión de tipos**
  - Convertir s_1 y s_2 a un tipo de dato más grande
  - Realizar la operación
  - Comprobar si el resultado está entre los límites permitidos del tipo de datos original

Universidad Zaragoza

Estrategias de mitigación

Soluciones software

- Funciones incorporada en run-time, en GCC **(para desbordamientos en operaciones con signo)**
  - Flag de compilación: `-ftrapv`
  - Sólo considera suma, resta y multiplicación con signo
- Compilador VC++ .NET 2003: **incorpora comprobación en tiempo de ejecución si hay truncamiento con pérdida de datos**
- Compilador VC++ .NET 2005 compiler: **desbordamientos en el operador `::new`**
- **Extensiones de compilador Big Loop Integer Protection (BLIP)**
- **SafeInt C++**
- **IntSafe**
  - Librería con funciones matemáticas y conversiones optimizadas y seguras
- **GNU Multiple Precision Arithmetic Library** (GMP)
  - Permite usar enteros muy grandes, con lo que es prácticamente imposible tener desbordamientos o subdesbordamientos

Estrategias de mitigación

Other solutions

Otros lenguajes de programación

- Los lenguajes de tipado seguro normalmente solucionan los problemas de desbrodamiento/subdesbordamiento mediante dos técnicas:
  - **Insertando código en tiempo de ejecución que eleva excepciones**
  - **Extendido el tipo de los enteros automáticamente**

Cuidado: algunos lenguajes de tipado seguro, como Java y OCaml, no protegen frente a desbordamientos

Universidad Zaragoza

Estrategias de mitigación

Other solutions

Otros lenguajes de programación

- Los lenguajes de tipado seguro normalmente solucionan los problemas de desbrodamiento/subdesbordamiento mediante dos técnicas:
  - **Insertando código en tiempo de ejecución que eleva excepciones**
  - **Extendido el tipo de los enteros automáticamente**

Cuidado: algunos lenguajes de tipado seguro, como Java y OCaml, no protegen frente a desbordamientos

Otra posibles técnicas

- **Analizadores estáticos**. Aproximaciones incompletas
- **Sistemas con tipos más expresivos**

Estrategias de mitigación

Auditoría de código

- **Verifica que los rangos de los tipos se comprueban correctamente**
- **Restringir los valores enteros a un rango válido, según sea el uso del valor**
- **Declarar apropiadamente el signo de los enteros**
- **Comprobar cotas superiores e inferiores de los rangos**
- **Todas aquellas operaciones sobre enteros de fuentes externas, mejor usar una librería de funciones de enteros seguras**

Universidad Zaragoza
1542

Fundamentos del Software: introducción a la programación verificada

Programación segura

Enteros

Roberto Blanco[†] & Ricardo J. Rodríguez[‡]

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Universidad Zaragoza

[†]Max Plank Institute for Security and Privacy
Bochum, Alemania

[‡]Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza (España)

Septiembre 2020

Universidad de Zaragoza