Secure Coding Lab Experiment - 13
Rule 04. Integers (INT)

INT30-C. Ensure that unsigned integer operations do not wrap

Integer values must not be allowed to wrap, especially if they are used in any of the following ways:

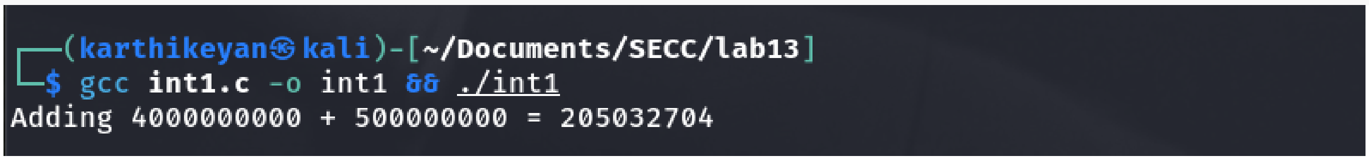
- Integer operands of any pointer arithmetic, including array indexing.
- The assignment expression for the declaration of a variable length array.
- The postfix expression preceding square brackets [ ] or the expression in square brackets [ ] of a subscripted designation of an element of an array object.
- Function arguments of type size_t or rsize_t (for example, an argument to a memory allocation function).
- In security-critical code .

Addition:

Noncompliant Code Example (Unsigned Integer Wrap)

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

```c
#include <stdio.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;

  // Debugging output
  printf("Adding %u + %u = %u\n", ui_a, ui_b, usum);
}

int main() {
  unsigned int a = 4000000000; // Example values
  unsigned int b = 500000000;
  func(a, b);
  return 0;
}
```

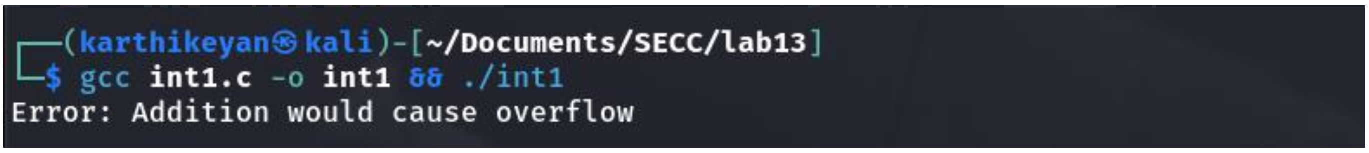


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Adding 4000000000 + 500000000 = 205032704
```

Compliant Solution (Precondition Test)

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

```
#include <stdio.h>
#include <limits.h>
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    printf("Error: Addition would cause overflow\n");
  } else {
    usum = ui_a + ui_b;
    printf("Result of addition: %u + %u = %u\n", ui_a, ui_b, usum);
  }
}

int main() {
  unsigned int a = 4000000000; // Example values
  unsigned int b = 500000000;
  func(a, b);
  return 0;
}
```

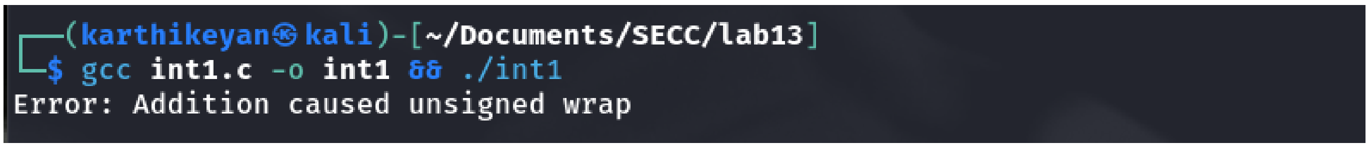


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Error: Addition would cause overflow
```

Compliant Solution (Postcondition Test)

This compliant solution performs a postcondition test to ensure that the result of the unsigned addition operation usum is not less than the first operand:

```
#include <stdio.h>
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  if (usum < ui_a) {
    printf("Error: Addition caused unsigned wrap\n");
  } else {
    printf("Result of addition: %u + %u = %u\n", ui_a, ui_b, usum);
  }
}

int main() {
  unsigned int a = 4000000000; // Example values
  unsigned int b = 500000000;
  func(a, b);
  return 0;
}
```



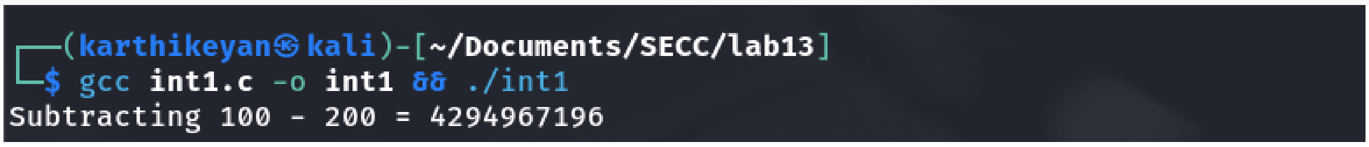
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Error: Addition caused unsigned wrap
```

Subtraction

Noncompliant Code Example (Unsigned Integer Wrap)

This noncompliant code example can result in an unsigned integer wrap during the subtraction of the unsigned operands ui_a and ui_b. If this behavior is unanticipated, it may lead to an exploitable vulnerability.

```c
#include <stdio.h>
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff = ui_a - ui_b;

  // Debugging output
  printf("Subtracting %u - %u = %u\n", ui_a, ui_b, udiff);
}

int main() {
  unsigned int a = 100; // Example values
  unsigned int b = 200;
  func(a, b);
  return 0;
}
```

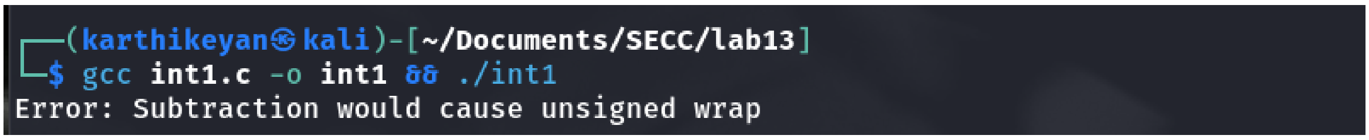


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Subtracting 100 - 200 = 4294967196
```

Compliant Solution (Precondition Test)

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

```c
#include <stdio.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff;
  if (ui_a < ui_b) {
    printf("Error: Subtraction would cause unsigned wrap\n");
  } else {
    udiff = ui_a - ui_b;
    printf("Result of subtraction: %u - %u = %u\n", ui_a, ui_b, udiff);
  }
}

int main() {
  unsigned int a = 100; // Example values
  unsigned int b = 200;
  func(a, b);
  return 0;
}
```

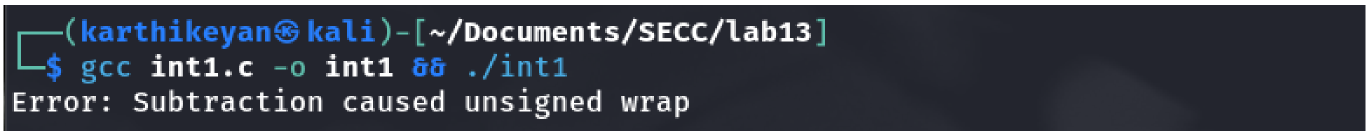


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Error: Subtraction would cause unsigned wrap
```

Compliant Solution (Postcondition Test)

This compliant solution performs a postcondition test that the result of the unsigned subtraction operation udiff is not greater than the minuend:

```
#include <stdio.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int udiff = ui_a - ui_b;
  if (udiff > ui_a) {
    printf("Error: Subtraction caused unsigned wrap\n");
  } else {
    printf("Result of subtraction: %u - %u = %u\n", ui_a, ui_b, udiff);
  }
}

int main() {
  unsigned int a = 100; // Example values
  unsigned int b = 200;
  func(a, b);
  return 0;
}
```



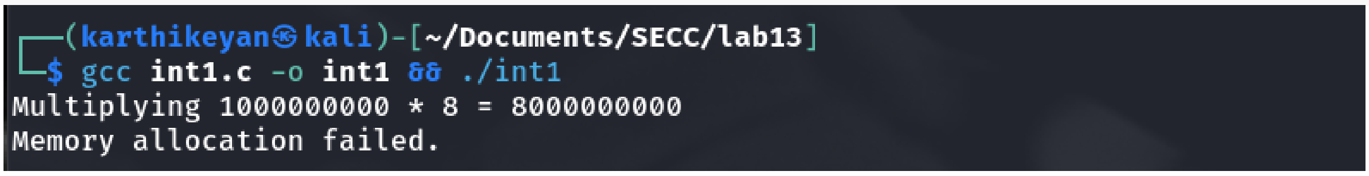
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Error: Subtraction caused unsigned wrap
```

Multiplication

Noncompliant Code Example (Unsigned Integer Wrap)

The signed int operand is converted to size_t prior to the multiplication operation so that the multiplication takes place between two size_t integers, which are unsigned.

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

void func(size_t num_vertices, size_t vertex_size) {
  size_t total_size = num_vertices * vertex_size;
  // Debugging output
  printf("Multiplying %zu * %zu = %zu\n", num_vertices, vertex_size, total_size);

  void* memory = malloc(total_size);
  if (memory) {
    printf("Memory allocation succeeded.\n");
    free(memory);
  } else {
    printf("Memory allocation failed.\n");
  }
}

int main() {
  size_t vertices = 1000000000; // Example values
  size_t size_of_vertex = 8;
  func(vertices, size_of_vertex);
  return 0;
}
```



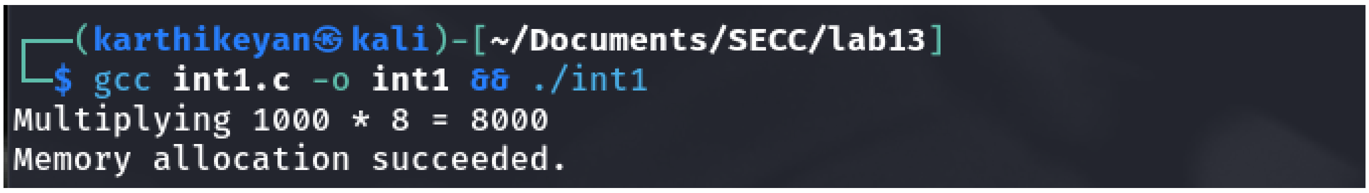
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Multiplying 1000000000 * 8 = 8000000000
Memory allocation failed.
```

Compliant Solution (Multiplication Precondition Test)

This compliant solution tests the operands of the multiplication to guarantee that there is no unsigned integer wrap:

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

void func(size_t num_vertices, size_t vertex_size) {
  if (num_vertices > SIZE_MAX / vertex_size) {
    printf("Error: Multiplication would cause overflow\n");
  } else {
    size_t total_size = num_vertices * vertex_size;
    printf("Multiplying %zu * %zu = %zu\n", num_vertices, vertex_size, total_size);

    void* memory = malloc(total_size);
    if (memory) {
      printf("Memory allocation succeeded.\n");
      free(memory);
    } else {
      printf("Memory allocation failed.\n");
    }
  }
}
```

```
int main() {
  size_t vertices = 1000000000; // Example values
  size_t size_of_vertex = 8;
  func(vertices, size_of_vertex);
  return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int1.c -o int1 && ./int1
Multiplying 1000 * 8 = 8000
Memory allocation succeeded.
```

INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data

Integer conversions, whether implicit or explicit (using a cast), must ensure that there is no loss or misinterpretation of data. This requirement is especially crucial for integer values sourced from untrusted origins, particularly in the following contexts:

- **Pointer Arithmetic**: Any integer operands used in pointer arithmetic, including array indexing, must be handled carefully.
- **Variable Length Arrays**: The assignment expression when declaring a variable-length array must be guaranteed to be valid.
- **Array Subscripts**: The postfix expression preceding square brackets [] or the expression within the brackets for accessing elements of an array must not lead to misinterpretation.
- **Function Arguments**: Arguments of type size_t or rsize_t, such as those passed to memory allocation functions, must be validated.

Additionally, this rule applies to arguments passed to specific library functions that convert integers to unsigned char, including:

- *memset()*
- *memset_s()*
- *fprintf() and related functions (for the length modifier c, where the int argument is converted to unsigned char)*
- *fputc()*
- *ungetc()*
- *memchr()*

It also includes functions that convert integers to char, such as:

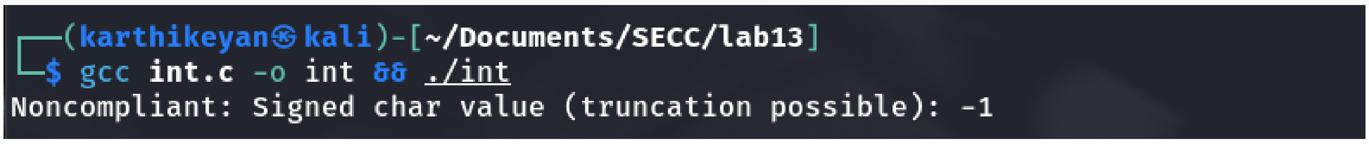
- *strchr()*
- *strrchr()*
- *All functions listed in <ctype.h>*

Ensuring safe conversions in these scenarios helps prevent undefined behavior and vulnerabilities in programs.

Noncompliant Code Example (Unsigned to Signed)

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type.

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

void func(void) {
   unsigned long int u_a = ULONG_MAX;
   signed char sc;
   sc = (signed char)u_a; /* Cast eliminates warning */
   printf("Noncompliant: Signed char value (truncation possible): %d\n", sc);
}

int main(void) {
   func();
   return 0;
}
```



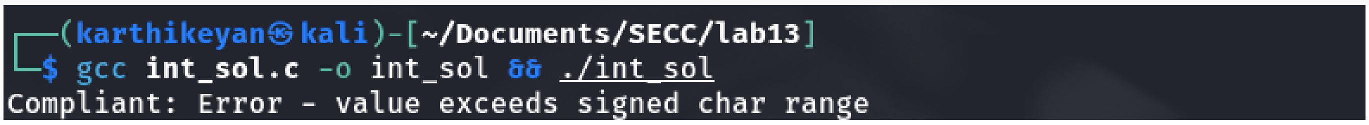
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Noncompliant: Signed char value (truncation possible): -1
```

Compliant Solution (Unsigned to Signed)

Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type:

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

void func(void) {
   unsigned long int u_a = ULONG_MAX;
   signed char sc;
   if (u_a <= SCHAR_MAX) {
      sc = (signed char)u_a;  /* Cast eliminates warning */
      printf("Compliant: Signed char value: %d\n", sc);
   } else {
      printf("Compliant: Error - value exceeds signed char range\n");
   }
}

int main(void) {
   func();
```

```
    return 0;
}
```

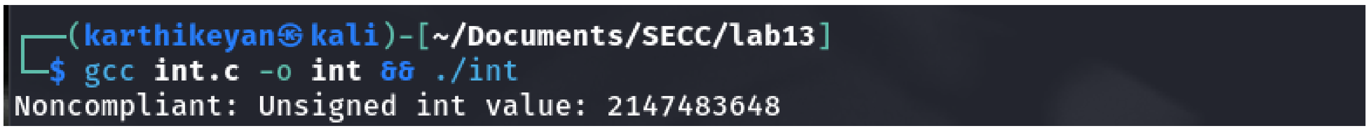


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Compliant: Error - value exceeds signed char range
```

Noncompliant Code Example (Signed to Unsigned)

Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of a signed type to a value of an unsigned type. This non-compliant code example results in a negative number being misinterpreted as a large positive number.

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

void func(signed int si) {
    /* Cast eliminates warning */
    unsigned int ui = (unsigned int)si;
    printf("Noncompliant: Unsigned int value: %u\n", ui);
}

/* Testing with INT_MIN */
int main(void) {
    func(INT_MIN);
    return 0;
}
```

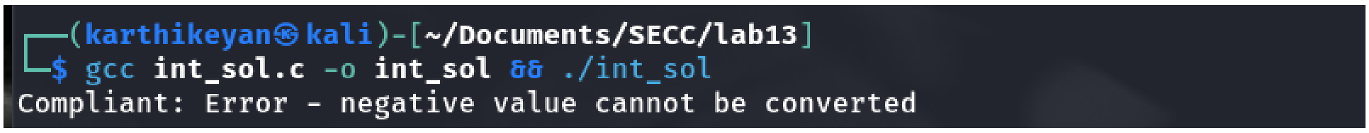


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Noncompliant: Unsigned int value: 2147483648
```

Compliant Solution (Signed to Unsigned)

Validate ranges when converting from a signed type to an unsigned type. This compliant solution converts a value of a signed int type to a value of an unsigned int type:

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

void func(signed int si) {
    unsigned int ui;
    if (si < 0) {
        printf("Compliant: Error - negative value cannot be converted\n");
    } else {
        ui = (unsigned int)si;  /* Cast eliminates warning */
        printf("Compliant: Unsigned int value: %u\n", ui);
    }
}

/* Testing with INT_MIN + 1 */
int main(void) {
    func(INT_MIN + 1);
    return 0;
}
```

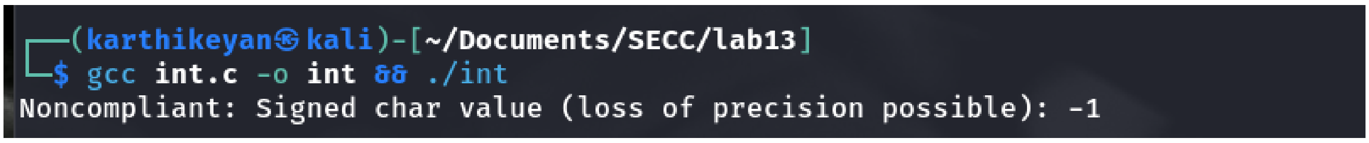


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Compliant: Error - negative value cannot be converted
```

Noncompliant Code Example (Signed, Loss of Precision)

A loss of data (truncation) can occur when converting from a value of a signed integer type to a value of a signed type with less precision. This noncompliant code example results in a truncation error on most implementations:

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

void func(void) {
    signed long int s_a = LONG_MAX;
    signed char sc = (signed char)s_a; /* Cast eliminates warning */
    printf("Noncompliant: Signed char value (loss of precision possible): %d\n", sc);
}

int main(void) {
    func();
    return 0;
}
```



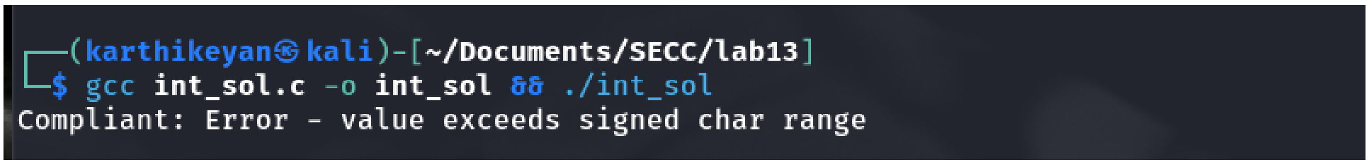
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Noncompliant: Signed char value (loss of precision possible): -1
```

Compliant Solution (Signed, Loss of Precision)

Validate ranges when converting from a signed type to a signed type with less precision. This compliant solution converts a value of a signed long int type to a value of a signed char type:

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

```
void func(void) {
    signed long int s_a = LONG_MAX;
    signed char sc;
    if ((s_a < SCHAR_MIN) || (s_a > SCHAR_MAX)) {
        printf("Compliant: Error - value exceeds signed char range\n");
    } else {
        sc = (signed char)s_a; /* Use cast to eliminate warning */
        printf("Compliant: Signed char value: %d\n", sc);
    }
}

int main(void) {
    func();
    return 0;
}
```

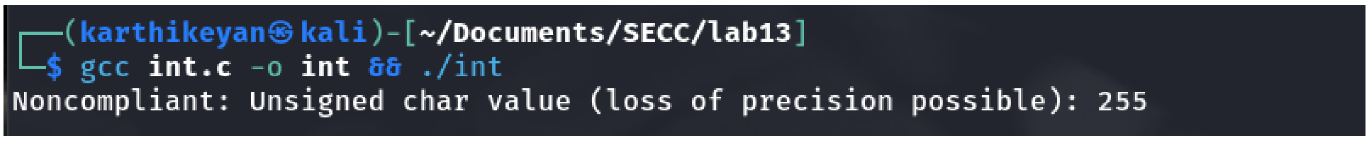


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Compliant: Error - value exceeds signed char range
```

Noncompliant Code Example (Unsigned, Loss of Precision)

A loss of data (truncation) can occur when converting from a value of an unsigned integer type to a value of an unsigned type with less precision. This noncompliant code example results in a truncation error on most implementations:

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

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    unsigned char uc = (unsigned char)u_a; /* Cast eliminates warning */
    printf("Noncompliant: Unsigned char value (loss of precision possible): %u\n", uc);
}

int main(void) {
    func();
    return 0;
}
```

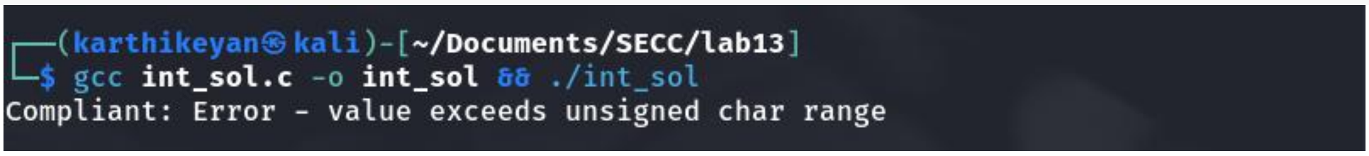


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Noncompliant: Unsigned char value (loss of precision possible): 255
```

Compliant Solution (Unsigned, Loss of Precision)

Validate ranges when converting a value of an unsigned integer type to a value of an unsigned integer type with less precision. This compliant solution converts a value of an unsigned long int type to a value of an unsigned char type:

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

void func(void) {
    unsigned long int u_a = ULONG_MAX;
    unsigned char uc;
    if (u_a > UCHAR_MAX) {
        printf("Compliant: Error - value exceeds unsigned char range\n");
    } else {
        uc = (unsigned char)u_a; /* Cast eliminates warning */
        printf("Compliant: Unsigned char value: %u\n", uc);
    }
}

int main(void) {
    func();
    return 0;
}
```

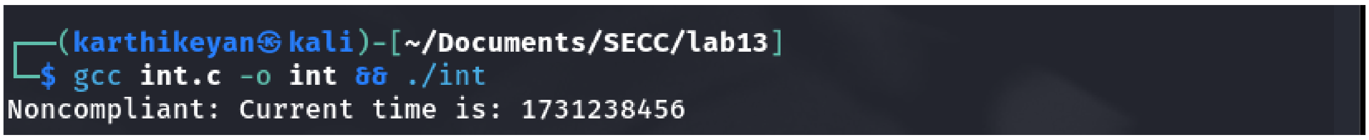


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Compliant: Error - value exceeds unsigned char range
```

Noncompliant Code Example (time_t Return Value)

The time() function returns the value (time_t)(-1) to indicate that the calendar time is not available. The C Standard requires that the time_t type is only a real type capable of representing time. (The integer and real floating types are collectively called real types.) It is left to the implementor to decide the best real type to use to represent time. If time_t is implemented as an unsigned integer type with less precision than a signed int, the return value of time() will never compare equal to the integer literal -1.

```
#include <stdio.h>
#include <time.h>

void func(void) {
    time_t now = time(NULL);
    if (now != -1) {
        printf("Noncompliant: Current time is: %ld\n", now);
    } else {
        printf("Noncompliant: Error - time not available\n");
    }
}

int main(void) {
    func();
    return 0;
}
```

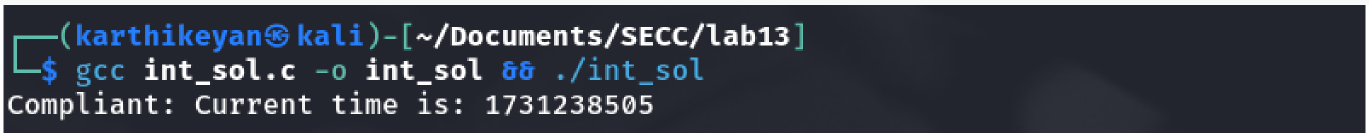


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Noncompliant: Current time is: 1731238456
```

Compliant Solution (time_t Return Value)

To ensure the comparison is properly performed, the return value of time() should be compared against -1 cast to type time_t:

```
#include <stdio.h>
#include <time.h>

void func(void) {
    time_t now = time(NULL);
    if (now != (time_t)-1) {
        printf("Compliant: Current time is: %ld\n", now);
    } else {
        printf("Compliant: Error - time not available\n");
    }
}

int main(void) {
    func();
    return 0;
}
```



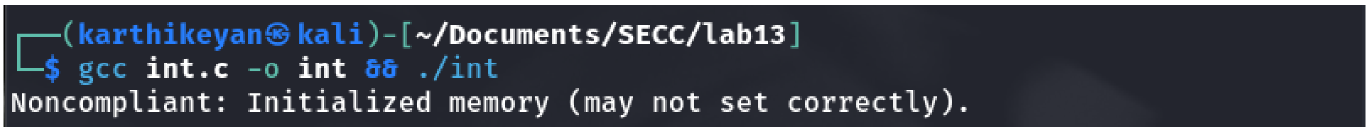
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Compliant: Current time is: 1731238505
```

Noncompliant Code Example (memset())

If the second argument is outside the range of a signed char or plain char, then its higher order bits will typically be truncated. Consequently, this noncompliant solution unexpectedly sets all elements in the array to 0, rather than 4096:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>
```

```
int *init_memory(int *array, size_t n) {
    return memset(array, 4096, n);
}

int main(void) {
    int arr[10];
    init_memory(arr, sizeof(arr));
    printf("Noncompliant: Initialized memory (may not set correctly).\n");
    return 0;
}
```

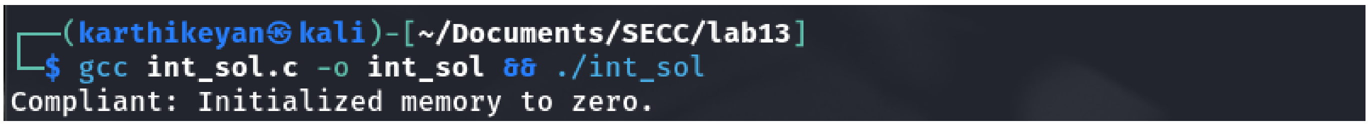


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Noncompliant: Initialized memory (may not set correctly).
```

Compliant Solution (memset())

In general, the memset() function should not be used to initialize an integer array unless it is to set or clear all the bits, as in this compliant solution:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>

int *init_memory(int *array, size_t n) {
    return memset(array, 0, n);
}

int main(void) {
    int arr[10];
    init_memory(arr, sizeof(arr));
    printf("Compliant: Initialized memory to zero.\n");
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Compliant: Initialized memory to zero.
```

INT32-C. Ensure that operations on signed integers do not result in overflow

Signed integer overflow results in undefined behavior, giving implementations flexibility in handling it. For example, some may define signed integers as modulo, trap overflows, or optimize code based on the assumption that overflows won't occur. This can lead to different behaviors depending on context, such as treating local variables as non-overflowing while allowing global variables to wrap.

To prevent issues, it's essential to ensure that operations on signed integers, especially from untrusted sources, do not lead to overflow. Key areas to monitor include:

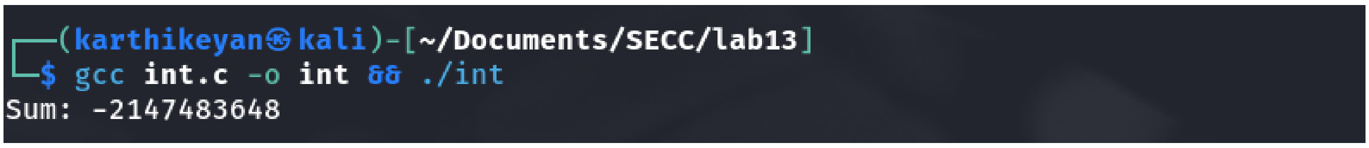
- Pointer arithmetic and array indexing.
- Variable-length array declarations.
- Array element access using subscripted designations.
- Function arguments of types size_t or rsize_t in memory allocation.

Addition

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the addition of the signed operands si_a and si_b:

```
#include <stdio.h>

void func(int si_a, int si_b) {
   int sum = si_a + si_b; // This can cause overflow
   printf("Sum: %d\n", sum);
}

int main() {
   func(2147483647, 1); // Example that may cause overflow
   return 0;
}
```



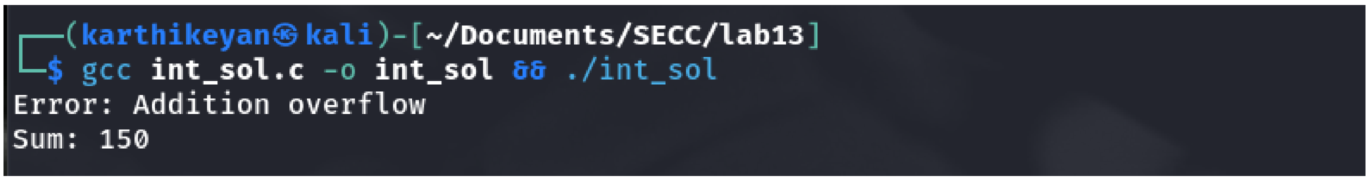
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Sum: -2147483648
```

Compliant Solution

This compliant solution ensures that the addition operation cannot overflow, regardless of representation:

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

void func(int si_a, int si_b) {
   int sum;
   if ((si_b > 0 && si_a > INT_MAX - si_b) || (si_b < 0 && si_a < INT_MIN - si_b)) {
      printf("Error: Addition overflow\n");
   } else {
      sum = si_a + si_b;
      printf("Sum: %d\n", sum);
```

```
    }
}

int main() {
    func(2147483647, 1); // Example that may cause overflow
    func(100, 50);       // Safe addition
    return 0;
}
```



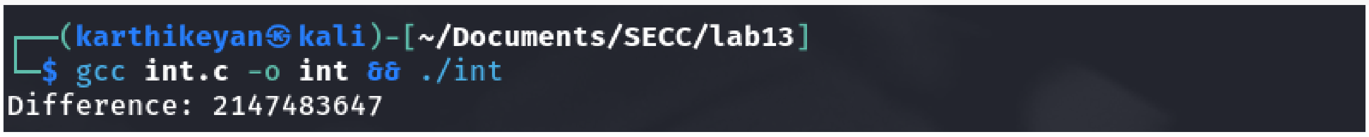
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Error: Addition overflow
Sum: 150
```

Subtraction

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the subtraction of the signed operands si_a and si_b:

```
#include <stdio.h>

void func(int si_a, int si_b) {
    int diff = si_a - si_b; // This can cause overflow
    printf("Difference: %d\n", diff);
}

int main() {
    func(-2147483648, 1); // Example that may cause overflow
    return 0;
}
```

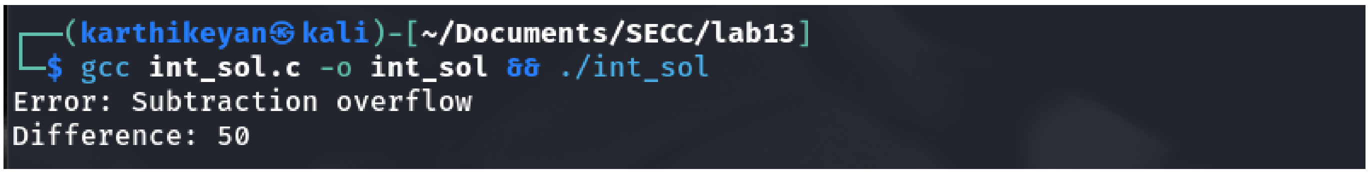


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Difference: 2147483647
```

Compliant Solution

This compliant solution tests the operands of the subtraction to guarantee there is no possibility of signed overflow, regardless of representation:

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

void func(int si_a, int si_b) {
    int diff;
    if ((si_b > 0 && si_a < INT_MIN + si_b) || (si_b < 0 && si_a > INT_MAX + si_b)) {
        printf("Error: Subtraction overflow\n");
    } else {
        diff = si_a - si_b;
        printf("Difference: %d\n", diff);
    }
}

int main() {
    func(-2147483648, 1); // Example that may cause overflow
    func(100, 50);        // Safe subtraction
    return 0;
}
```



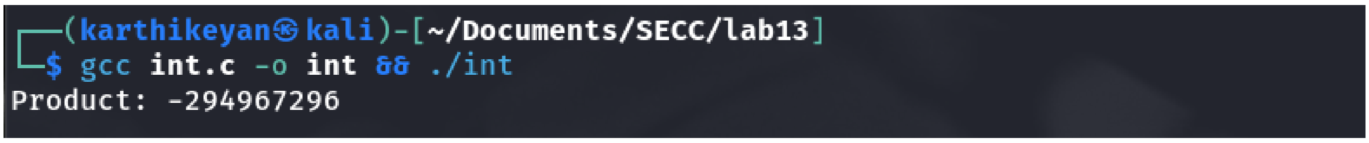
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Error: Subtraction overflow
Difference: 50
```

Multiplication

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the multiplication of the signed operands si_a and si_b:

```
#include <stdio.h>

void func(int si_a, int si_b) {
    int result = si_a * si_b; // This can cause overflow
    printf("Product: %d\n", result);
}

int main() {
    func(2000000000, 2); // Example that may cause overflow
    return 0;
}
```

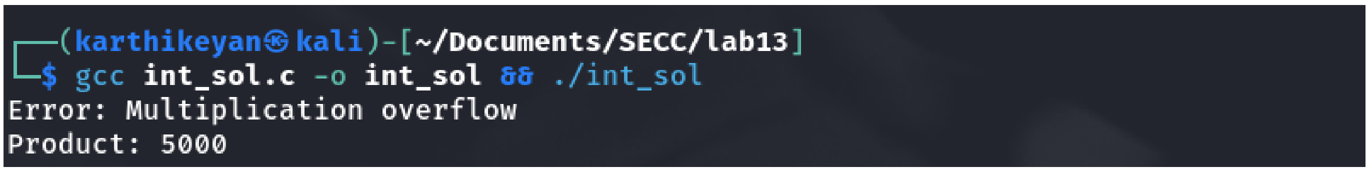


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Product: -294967296
```

Compliant Solution

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

```c
void func(int si_a, int si_b) {
    int result;
    if (si_a > 0) {
        if (si_b > 0) {
            if (si_a > INT_MAX / si_b) {
                printf("Error: Multiplication overflow\n");
                return;
            }
        } else {
            if (si_b < INT_MIN / si_a) {
                printf("Error: Multiplication overflow\n");
                return;
            }
        }
    } else {
        if (si_b > 0) {
            if (si_a < INT_MIN / si_b) {
                printf("Error: Multiplication overflow\n");
                return;
            }
        } else {
            if ((si_a != 0) && (si_b < INT_MAX / si_a)) {
                printf("Error: Multiplication overflow\n");
                return;
            }
        }
    }
    result = si_a * si_b;
    printf("Product: %d\n", result);
}


int main() {

    func(2000000000, 2); // Example that may cause overflow

    func(100, 50);       // Safe multiplication

    return 0;

}
```



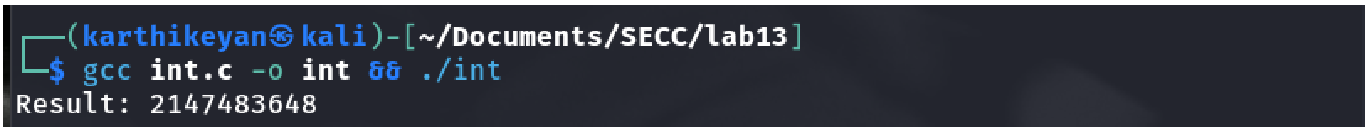
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Error: Multiplication overflow
Product: 5000
```

Division

Noncompliant Code Example

This noncompliant code example prevents divide-by-zero errors in compliance with INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors but does not prevent a signed integer overflow error in two's-complement.

```
#include <stdio.h>

void func(long s_a, long s_b) {
    long result = s_a / s_b; // This can cause divide by zero or overflow
    printf("Result: %ld\n", result);
}

int main() {
    func(-2147483648, -1); // Example that may cause overflow
    return 0;
}
```

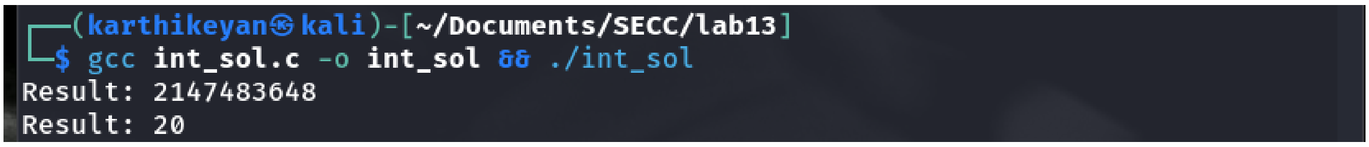


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Result: 2147483648
```

Compliant Solution

This compliant solution eliminates the possibility of divide-by-zero errors or signed overflow:

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

void func(long s_a, long s_b) {
    long result;
    if (s_b == 0 || (s_a == LONG_MIN && s_b == -1)) {
        printf("Error: Division by zero or overflow\n");
    } else {
        result = s_a / s_b;
        printf("Result: %ld\n", result);
    }
}
int main() {

    func(-2147483648, -1); // Example that may cause overflow

    func(100, 5);          // Safe division

    return 0;

}
```



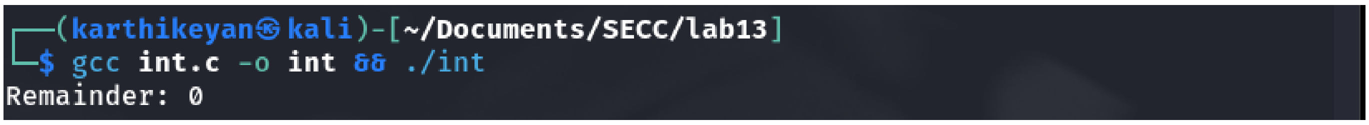
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Result: 2147483648
Result: 20
```

Remainder

Noncompliant Code Example

Many hardware architectures implement remainder as part of the division operator, which can overflow. Overflow can occur during a remainder operation when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to −1. It occurs even though the result of such a remainder operation is mathematically 0. This noncompliant code example prevents divide-by-zero errors in compliance with INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors but does not prevent integer overflow:

```
#include <stdio.h>

void func(long s_a, long s_b) {
   long result = s_a % s_b; // This can cause divide by zero or overflow
   printf("Remainder: %ld\n", result);
}

int main() {
   func(-2147483648, -1); // Example that may cause overflow
   return 0;
}
```



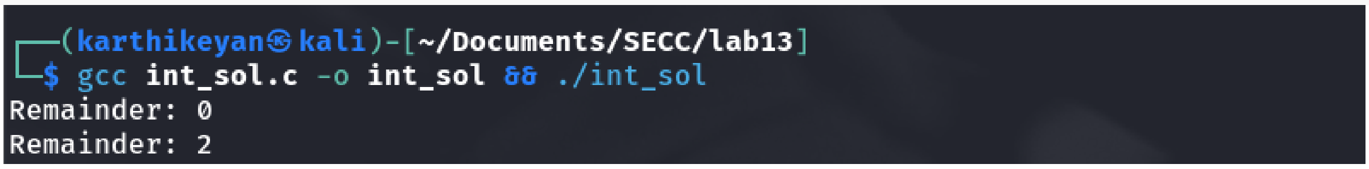
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Remainder: 0
```

Compliant Solution

This compliant solution also tests the remainder operands to guarantee there is no possibility of an overflow:

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

void func(long s_a, long s_b) {
   long result;
   if (s_b == 0 || (s_a == LONG_MIN && s_b == -1)) {
      printf("Error: Division by zero or overflow\n");
   } else {
      result = s_a % s_b;
```

```
        printf("Remainder: %ld\n", result);
    }
}

int main() {
    func(-2147483648, -1); // Example that may cause overflow
    func(100, 7);          // Safe remainder
    return 0;
}
```



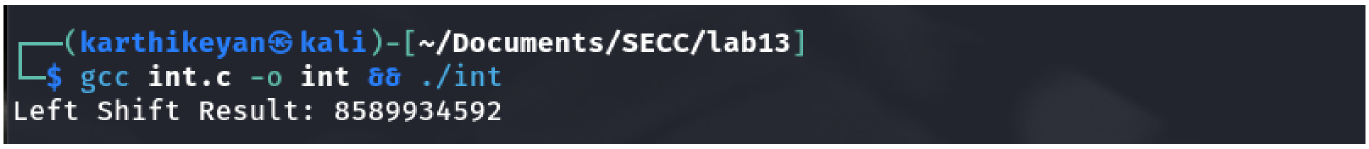
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Remainder: 0
Remainder: 2
```

Left-Shift Operator

Noncompliant Code Example

This performs a left shift, after verifying that the number being shifted is not negative, and the number of bits to shift is valid. The PRECISION() macro and popcount() function provide the correct precision for any integer type. (See INT35-C. Use correct integer precisions.) However, because this code does no overflow check, it can result in an unrepresentable value.

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

void func(long si_a, long si_b) {
    long result = si_a << si_b; // This can cause overflow
    printf("Left Shift Result: %ld\n", result);
}

int main() {
    func(1, 33); // Example that may cause overflow
    return 0;
}
```



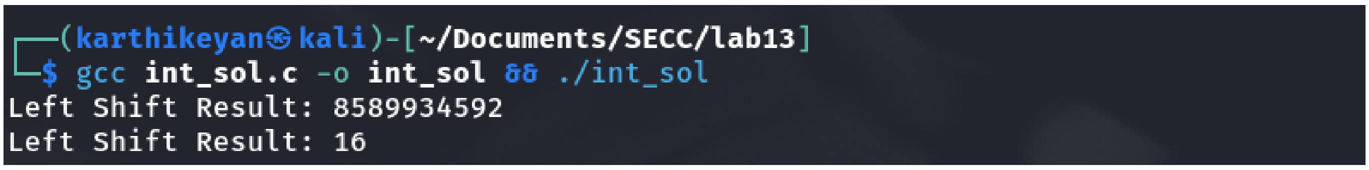
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Left Shift Result: 8589934592
```

Compliant Solution

This compliant solution eliminates the possibility of overflow resulting from a left-shift operation:

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

```
void func(long si_a, long si_b) {
    long result;
    if (si_a < 0 || si_b < 0 || si_b >= sizeof(long) * 8 || si_a > (LONG_MAX >> si_b)) {
        printf("Error: Left shift overflow\n");
    } else {
        result = si_a << si_b;
        printf("Left Shift Result: %ld\n", result);
    }
}

int main() {
    func(1, 33); // Example that may cause overflow
    func(4, 2);  // Safe left shift
    return 0;
}
```



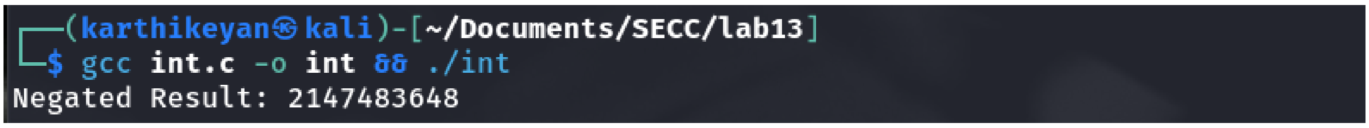
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Left Shift Result: 8589934592
Left Shift Result: 16
```

Unary Negation

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the unary negation of the signed operand s_a:

```
#include <stdio.h>
void func(long s_a) {
    long result = -s_a; // This can cause overflow
    printf("Neglated Result: %ld\n", result);
}

int main() {
    func(-2147483648); // Example that may cause overflow
    return 0;
}
```

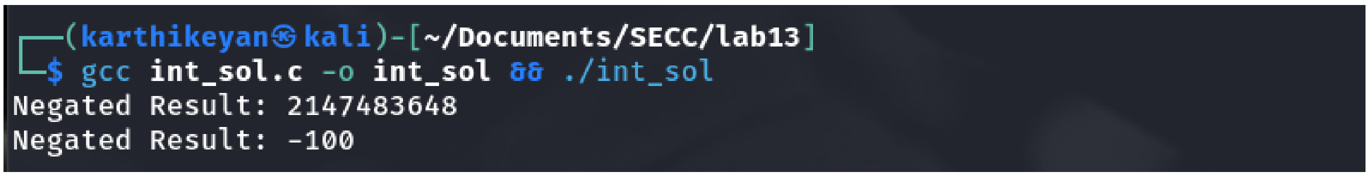


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Negated Result: 2147483648
```

Compliant Solution

This compliant solution tests the negation operation to guarantee there is no possibility of signed overflow:

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

void func(long s_a) {
    long result;
    if (s_a == LONG_MIN) {
        printf("Error: Unary negation overflow\n");
    } else {
        result = -s_a;
        printf("Negated Result: %ld\n", result);
    }
}

int main() {
    func(-2147483648); // Example that may cause overflow
    func(100);         // Safe negation
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Negated Result: 2147483648
Negated Result: -100
```

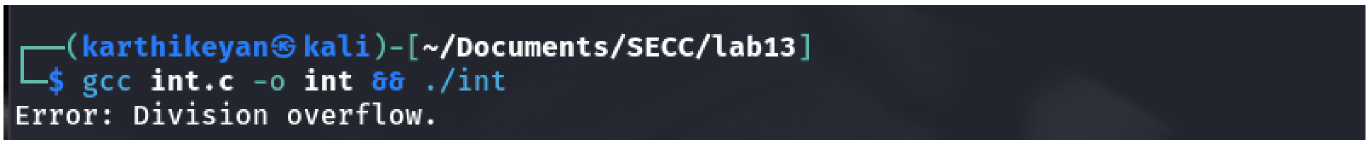
INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors

Division

Noncompliant Code Example:

This noncompliant code example prevents signed integer overflow in compliance with INT32-C. Ensure that operations on signed integers do not result in overflow but fails to prevent a divide-by-zero error during the division of the signed operands s_a and s_b:

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

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_a == LONG_MIN) && (s_b == -1)) {
        /* Handle error */
        printf("Error: Division overflow.\n");
    } else {
        result = s_a / s_b;
        printf("Result of division: %ld\n", result);
    }
}

int main() {
    signed long a = LONG_MIN;
    signed long b = -1;

    // Test noncompliant division
    func(a, b); // This will show an overflow error

    return 0;
}
```

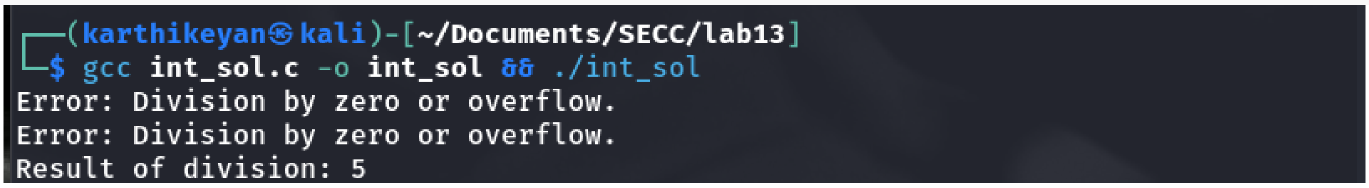


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Error: Division overflow.
```

Compliant Solution:

This compliant solution tests the division operation to guarantee there is no possibility of divide-by-zero errors or signed overflow:

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

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_b == 0) || ((s_a == LONG_MIN) && (s_b == -1))) {
        /* Handle error */
        printf("Error: Division by zero or overflow.\n");
    } else {
        result = s_a / s_b;
        printf("Result of division: %ld\n", result);
    }
}

int main() {
    signed long a = LONG_MIN;
    signed long b = -1;

    // Test compliant division
    func(a, b); // This will show an overflow error

    b = 0;
    func(a, b); // This will show a division by zero error
    a = 10;
    b = 2;
    func(a, b); // This will show the result

    return 0;
}
```



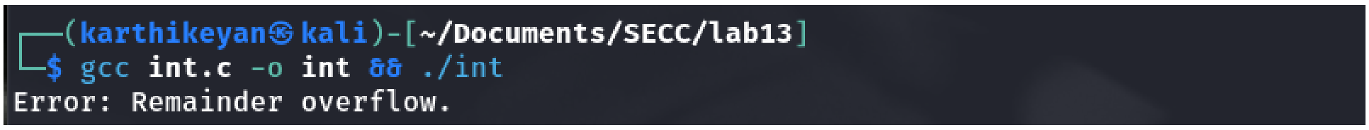
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Error: Division by zero or overflow.
Error: Division by zero or overflow.
Result of division: 5
```

Remainder

Noncompliant Code Example:

This noncompliant code example prevents signed integer overflow in compliance with INT32-C. Ensure that operations on signed integers do not result in overflow but fails to prevent a divide-by-zero error during the remainder operation on the signed operands s_a and s_b:

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

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_a == LONG_MIN) && (s_b == -1)) {
        /* Handle error */
        printf("Error: Remainder overflow.\n");
    } else {
        result = s_a % s_b;
        printf("Result of remainder: %ld\n", result);
    }
}

int main() {
    signed long a = LONG_MIN;
    signed long b = -1;

    // Test noncompliant remainder
    func(a, b); // This will show an overflow error

    return 0;
}
```

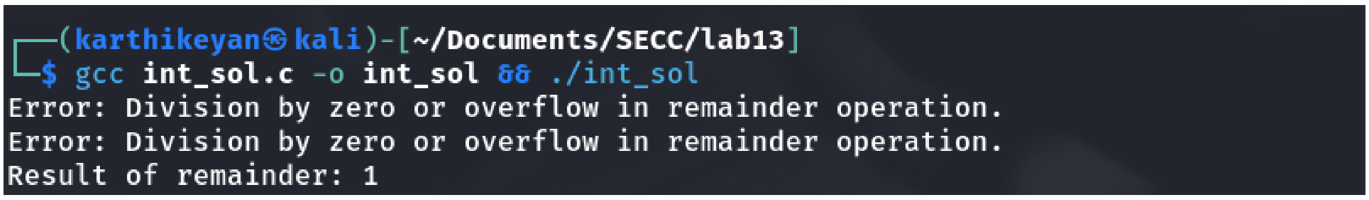


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Error: Remainder overflow.
```

Compliant Solution:

This compliant solution tests the remainder operand to guarantee there is no possibility of a divide-by-zero error or an overflow error:

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

void func(signed long s_a, signed long s_b) {
    signed long result;
    if ((s_b == 0) || ((s_a == LONG_MIN) && (s_b == -1))) {
        /* Handle error */
        printf("Error: Division by zero or overflow in remainder operation.\n");
    } else {
        result = s_a % s_b;
        printf("Result of remainder: %ld\n", result);
    }
}

int main() {
    signed long a = LONG_MIN;
    signed long b = -1;

    // Test compliant remainder
    func(a, b); // This will show an overflow error

    b = 0;
    func(a, b); // This will show a division by zero error

    a = 10;
    b = 3;
    func(a, b); // This will show the result

    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Error: Division by zero or overflow in remainder operation.
Error: Division by zero or overflow in remainder operation.
Result of remainder: 1
```

INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

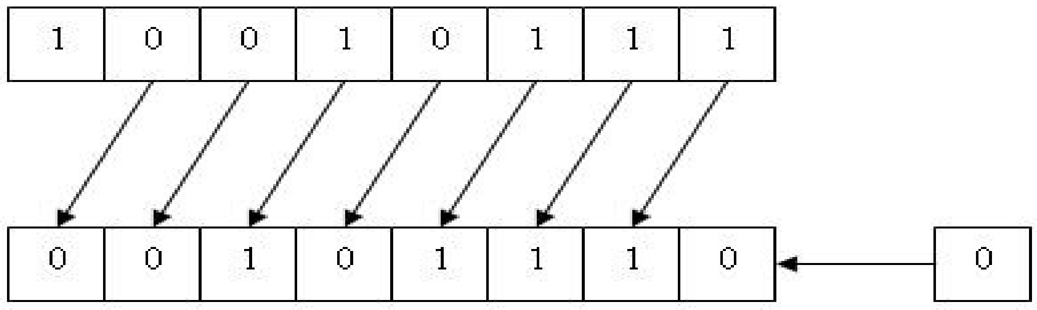
- Left (<<) and Right (>>) Shifts: Move bits left or right by a specified count.

- Integer Promotion: Operands are promoted to integer types, and the result type matches the left operand.
- Valid Shift Range: Shift count must be non-negative and less than the bit width of the left operand; otherwise, it's undefined behavior.
- Precision vs. Width: Precision counts only value bits, excluding sign bits (important for signed types).
- Unsigned Operands: Use shifts on unsigned types to avoid unexpected results with sign bits in signed integers.

Left Shift

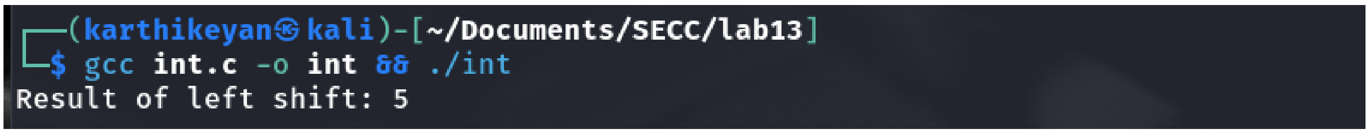
Noncompliant Code Example (Left Shift, Unsigned Type)

The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. The following diagram illustrates the left-shift operation.



According to the C Standard, if E1 has an unsigned type, the value of the result is E1 * 2^E2, reduced modulo 1 more than the maximum value representable in the result type.

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

void func(unsigned int ui_a, unsigned int ui_b) {
   unsigned int uresult = ui_a << ui_b;
    printf("Result of left shift: %u\n", uresult);
}

int main() {
   unsigned int a = 5;
   unsigned int b = 32; // Shifting beyond the precision of unsigned int

   // Test noncompliant left shift
   func(a, b); // This could result in undefined behavior

   return 0;
}
```

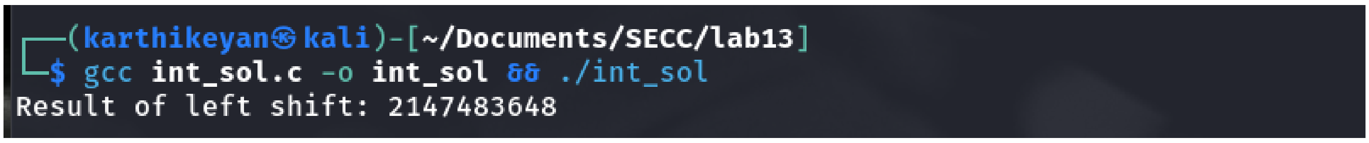


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Result of left shift: 5
```

Compliant Solution (Left Shift, Unsigned Type)

This noncompliant code example fails to ensure that the right operand is less than the precision of the promoted left operand:

```c
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>
#include <stdio.h>

extern size_t popcount(uintmax_t);
#define PRECISION(x) (sizeof(x) * CHAR_BIT) // Calculate precision based on type size

void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int uresult = 0;
    if (ui_b >= PRECISION(UINT_MAX)) {
        /* Handle error */
        printf("Error: Shift count exceeds precision of unsigned int.\n");
    } else {
        uresult = ui_a << ui_b;
        printf("Result of left shift: %u\n", uresult);
    }
}

int main() {
    unsigned int a = 5;
    unsigned int b = 31; // Valid shift

    // Test compliant left shift
    func(a, b); // Should show result

    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Result of left shift: 2147483648
```

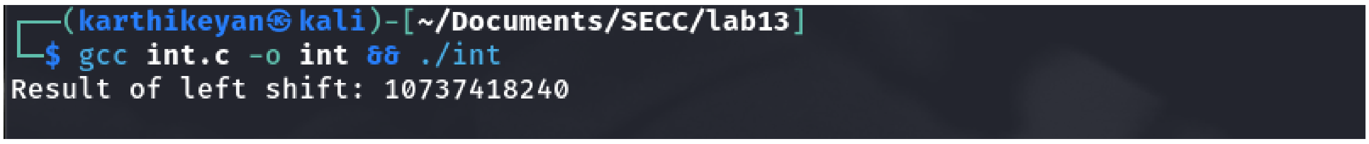
Left Shift

Noncompliant Code Example (Left Shift, Signed Type)

The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has a signed type and nonnegative value, and E1 * 2E2 is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

This noncompliant code example fails to ensure that left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand. This example does check for signed integer overflow in compliance with INT32-C. Ensure that operations on signed integers do not result in overflow.

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

void func(signed long si_a, signed long si_b) {
  signed long result;
  if (si_a > (LONG_MAX >> si_b)) {
    /* Handle error */
    printf("Error: Shift would cause overflow.\n");
  } else {
    result = si_a << si_b;
    printf("Result of left shift: %ld\n", result);
  }
}

int main() {
  signed long a = 5;
  signed long b = 31; // Valid shift

  // Test noncompliant left shift
  func(a, b); // Should show result

  return 0;
}
```



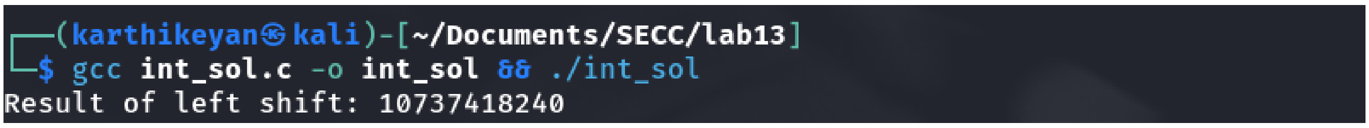
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Result of left shift: 10737418240
```

Compliant Solution (Left Shift, Signed Type)

In addition to the check for overflow, this compliant solution ensures that both the left and right operands have nonnegative values and that the right operand is less than the precision of the promoted left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>
```

```
#include <stdio.h>

extern size_t popcount(uintmax_t);
#define PRECISION(x) (sizeof(x) * CHAR_BIT) // Calculate precision based on type size

void func(signed long si_a, signed long si_b) {
    signed long result;
    if ((si_a < 0) || (si_b < 0) ||
        (si_b >= PRECISION(ULONG_MAX)) ||
        (si_a > (LONG_MAX >> si_b))) {
        /* Handle error */
        printf("Error: Invalid shift operation.\n");
    } else {
        result = si_a << si_b;
        printf("Result of left shift: %ld\n", result);
    }
}

int main() {
    signed long a = 5;
    signed long b = 31; // Valid shift

    // Test compliant left shift
    func(a, b); // Should show result

    return 0;
}
```

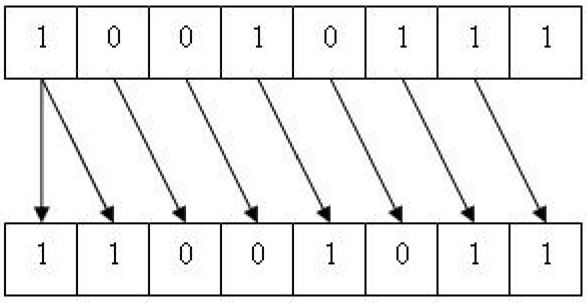


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Result of left shift: 10737418240
```

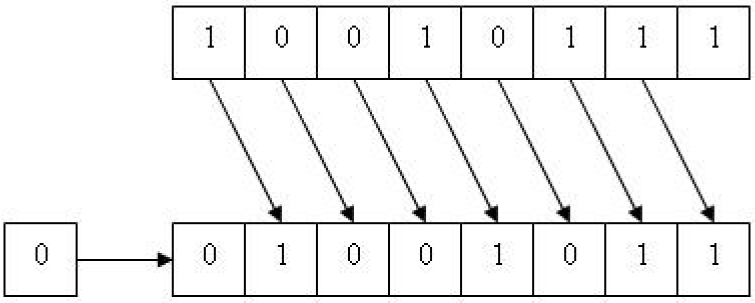
Right Shift

Noncompliant Code Example (Right Shift)

The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 / 2E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined and can be either an arithmetic (signed) shift

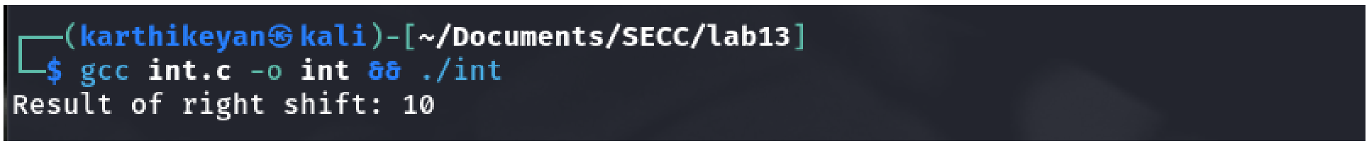


or a logical (unsigned) shift



This noncompliant code example fails to test whether the right operand is greater than or equal to the precision of the promoted left operand, allowing undefined behavior:

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

void func(unsigned int ui_a, unsigned int ui_b) {
   unsigned int uresult = ui_a >> ui_b;  // Potential undefined behavior if ui_b >= precision of unsigned
int
   printf("Result of right shift: %u\n", uresult);
}

int main() {
   unsigned int a = 10;
   unsigned int b = 32;  // Shifting by 32 bits, which is beyond the typical precision for unsigned int

   // Test noncompliant right shift
   func(a, b);  // Undefined behavior if the shift count exceeds the precision of unsigned int

   return 0;
}
```

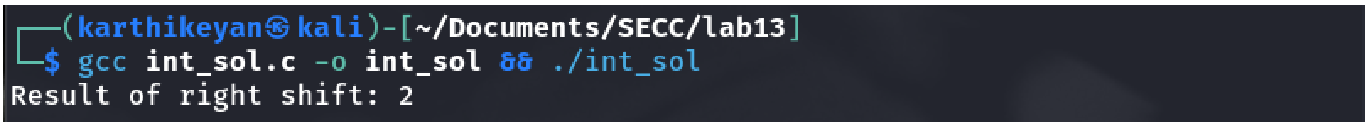


```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Result of right shift: 10
```

Compliant Solution (Right Shift)

This compliant solution eliminates the possibility of shifting by greater than or equal to the number of bits that exist in the precision of the left operand:

```
#include <limits.h>
#include <stddef.h>
#include <inttypes.h>
#include <stdio.h>

extern size_t popcount(uintmax_t);
#define PRECISION(x) (sizeof(x) * CHAR_BIT) // Calculate precision based on type size

void func(unsigned int ui_a, unsigned int ui_b) {
   unsigned int uresult = 0;
   if (ui_b >= PRECISION(UINT_MAX)) {
      /* Handle error */
      printf("Error: Shift count exceeds precision of unsigned int.\n");
   } else {
      uresult = ui_a >> ui_b;
      printf("Result of right shift: %u\n", uresult);
   }
}

int main() {
   unsigned int a = 10;
   unsigned int b = 2; // Valid shift

   // Test compliant right shift
   func(a, b); // Should show result

   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Result of right shift: 2
```

INT35-C. Use correct integer precisions

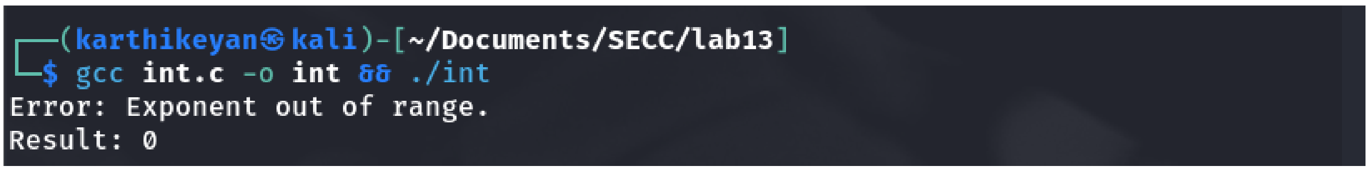
- Size refers to the number of bytes used by an integer type or object, which can be obtained using the sizeof operator.
- Precision is the number of bits used to represent values, excluding sign and padding bits, and determines the range of representable values for that type.
- Padding bits may be added for alignment purposes but do not contribute to the integer's precision, which can lead to confusion when determining the actual usable bit count.

- Relying on size to infer precision can result in incorrect assumptions about the numeric range of integer types, potentially causing issues in operations involving shifts or arithmetic that depend on precision.
- It is essential to use correct integer precisions explicitly in code, avoid using the sizeof operator to compute precision on architectures that use padding bits, and strive for portability in programs by adhering to standards for integer precision.

Noncompliant Code Example

This code attempts to implement a power of two function, but it improperly calculates the maximum shift based on the size of unsigned int, which could be misleading.

```
#include <limits.h>
#include <stdio.h>
unsigned int pow2(unsigned int exp)
{
        if (exp >= sizeof(unsigned int) * CHAR_BIT)
         {
                // Handle error
                printf("Error: Exponent out of range.\n");
                return 0; // Return 0 for this example
        }
        return 1 << exp; // Shift left by exp bits
}
int main()
{
        unsigned int result = pow2 (32); // Trying to compute 2^32
        printf("Result: %u\n", result);
        return 0;
}
```



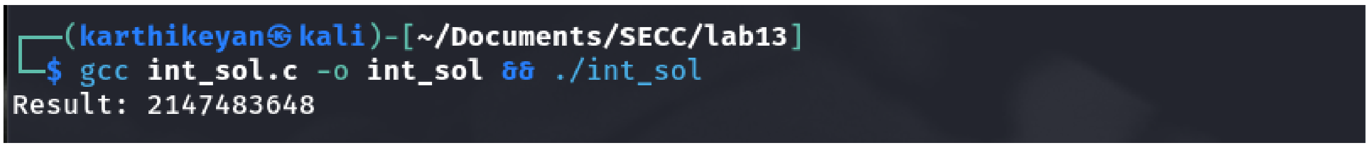
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Error: Exponent out of range.
Result: 0
```

Compliant Solution

The compliant solution employs a popcount() function to accurately determine the precision of the integer type. This approach ensures that the function only allows valid shifts.

```
#include <stddef.h>
#include <stdint.h>
#include <limits.h>
#include <stdio.h>
// Function to count set bits
```

```
size_t popcount(uintmax_t num)
{
  size_t precision = 0;
  while (num != 0) {
    if (num % 2 == 1) {
      precision ++;
    }
  num >>= 1;
  }
  return precision;
}
#define PRECISION(umax_value) popcount(umax_value)
unsigned int pow2(unsigned int exp) {
  if (exp >= PRECISION(UINT_MAX))
  {
    // Handle error
    printf("Error: Exponent out of range.\n");
    return 0; // Return 0 for this example
  }
  return 1 << exp; // Shift left by exp bits
}
int main() {
  unsigned int result = pow2 (31); // Trying to compute 2^31
  printf("Result: %u\n", result);
  return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Result: 2147483648
```

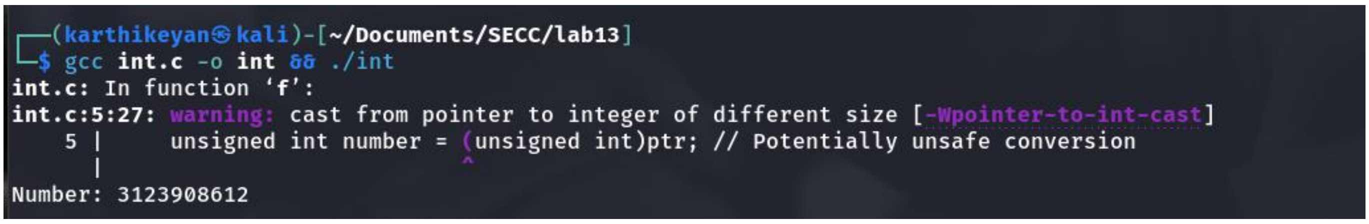
INT36-C. Converting a pointer to integer or integer to pointer

- Conversions between integers and pointers can lead to undesired consequences depending on the implementation. Care must be taken to ensure that conversions do not result in misaligned pointers or loss of information. The following guidelines outline best practices for safely performing these conversions in C.
- Do not convert an integer type to a pointer type if the resulting pointer is incorrectly aligned, does not point to an entity of the referenced type, or is a trap representation.
- Do not convert a pointer type to an integer type if the result cannot be represented in the integer type.
- The mapping between pointers and integers must align with the addressing structure of the execution environment. Issues may arise on architectures with a segmented memory model.

Noncompliant Code Example 1

This example converts a pointer of type char* to an unsigned int. If the pointer size exceeds the size of the integer, it can lead to data loss or undefined behavior.

```
#include <stdio.h>

void f(void) {
   char *ptr = "Hello, World!";
   unsigned int number = (unsigned int)ptr; // Potentially unsafe conversion
   printf("Number: %u\n", number);
}

int main() {
   f();
   return 0;
}
```



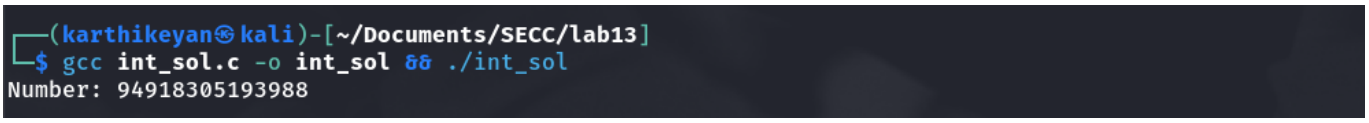
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
int.c: In function ‘f’:
int.c:5:27: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
    5 |     unsigned int number = (unsigned int)ptr; // Potentially unsafe conversion
      |                           ^
Number: 3123908612
```

The output will show an implementation-defined integer value representing the pointer. However, this can cause issues on systems where pointers are larger than unsigned int, potentially leading to data loss.

Compliant Code Example 1

In this compliant version, the pointer is converted to uintptr_t, which is designed to safely hold pointer values without data loss. This ensures that the conversion is safe and can be reversed without issues.

```
#include <stdint.h>
#include <stdio.h>

void f(void) {
   char *ptr = "Hello, World!";
   uintptr_t number = (uintptr_t)ptr; // Safe conversion
   printf("Number: %lu\n", (unsigned long)number); // Cast to unsigned long for printing
}

int main() {
   f();
   return 0;
}
```



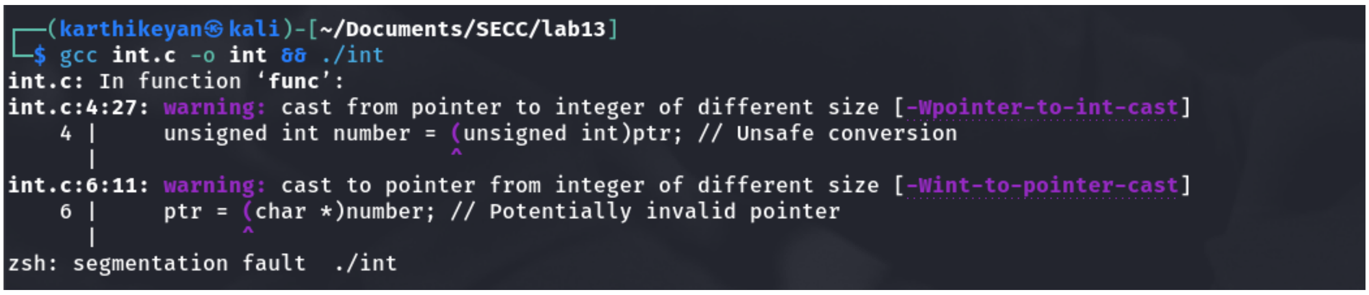
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Number: 94918305193988
```

The output will correctly represent the pointer's address without data loss.

Noncompliant Code Example 2

This example combines a pointer with an integer flag using bit manipulation. If the pointer's address does not fit in unsigned int, this can lead to undefined behavior.

```c
#include <stdio.h>
void func(unsigned int flag) {
   char *ptr = "Hello, World!";
   unsigned int number = (unsigned int)ptr; // Unsafe conversion
   number = (number & 0x7fffff) | (flag << 23);
   ptr = (char *)number; // Potentially invalid pointer
    printf("Pointer: %s",ptr);
}

int main() {
   func(1);
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
int.c: In function 'func':
int.c:4:27: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
    4 |     unsigned int number = (unsigned int)ptr; // Unsafe conversion
      |                           ^
int.c:6:11: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    6 |     ptr = (char *)number; // Potentially invalid pointer
      |           ^
zsh: segmentation fault  ./int
```

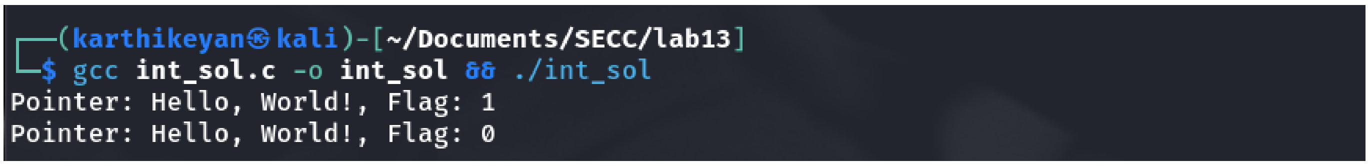
The output is undefined since it may lead to an invalid pointer or memory access violation.

Compliant Code Example 2

This version uses a struct to store both the pointer and a flag, ensuring that the pointer is stored safely without losing information.

```c
#include <stdio.h>

struct ptrflag {
   char *pointer;
   unsigned int flag : 9; // Flag using bit field
};
```

```
void func(unsigned int flag) {
    struct ptrflag pf;
    char *ptr = "Hello, World!";
    pf.pointer = ptr; // Safe storage
    pf.flag = flag; // Safe flag assignment
    printf("Pointer: %s, Flag: %u\n", pf.pointer, pf.flag);
}

int main() {
    func(1);
    return 0;
}
```



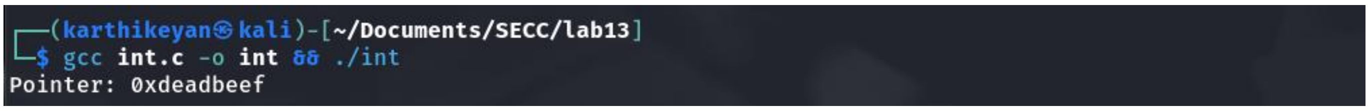
```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol && ./int_sol
Pointer: Hello, World!, Flag: 1
Pointer: Hello, World!, Flag: 0
```

The output confirms that both the pointer and flag are stored safely and accurately.

Noncompliant Code Example 3

This example directly assigns an integer constant to a pointer, which is implementation-defined and may lead to misaligned or invalid pointers.

```
#include <stdio.h>

unsigned int *g(void) {
    unsigned int *ptr = (unsigned int *)0xdeadbeef; // Unsafe assignment
    return ptr; // Potentially invalid pointer
}

int main() {
    unsigned int *result = g();
    printf("Pointer: %p\n", (void *)result);
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int.c -o int && ./int
Pointer: 0xdeadbeef
```

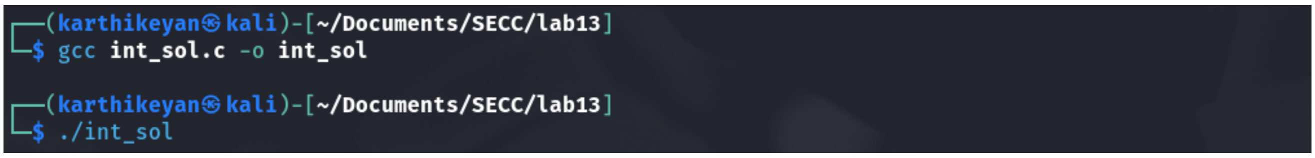
The output may show a pointer, but dereferencing this pointer can lead to undefined behaviour.

Compliant Code Example 3

Unfortunately, this code cannot be made safe while strictly conforming to ISO C.

A particular platform (that is, hardware, operating system, compiler, and Standard C library) might guarantee that a memory address is correctly aligned for the pointer type, and actually contains a value for that type. A common practice is to use addresses that are known to point to hardware that provides valid values.

```
#include <stdio.h>
void h(void) {
// No compliant solution; this remains implementation -specific
// Use with caution , knowing the environment.
}
int main() {
h(); // Placeholder for demonstration
return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ gcc int_sol.c -o int_sol

┌──(karthikeyan㉿kali)-[~/Documents/SECC/lab13]
└─$ ./int_sol
```