Integer Overflow:

Example-1:

```c
int x, y, z;
x = 0x7fffffff;
y = 1;
z = x + y;
printf("%d", z);
```

Example-2:

```c
unsigned int x;
int y = -5;
x = y;
printf("%d", x);
```

Unsigned int and signed int Ranges:

<table>
<thead>
<tr>
<th>Unsigned int</th>
<th>Signed int</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 –&gt; +4,294,967,295</td>
<td>-2,147,483,648 –&gt; +2,147,483,647</td>
</tr>
</tbody>
</table>

short to int Conversion:

```c
int x;
short int y;
x = 100000;
y = x;
```

“short int” range: -32,768 –> +32,767  
“int” range: -2,147,483,648 –> +2,147,483,647

Security Problem Example:

```c
unsigned int x;
int y;
read(net, &x, sizeof(x));
y = x + sizeof_hdr;
if( y < MAX_OFFSET)
    Table [x] = 0;
```

Integer overflow here can lead to writing to a critical memory area.
Format String Bugs:

Various print functions:
- printf: prints to screen
- sprintf: prints to string
- snprintf: prints to string with bound

Example:
1) `printf("Hello World!\n");`
2) `snprintf(dst, sizeof(dst), "%s: %d minutes to %x\n", username, nminutes, 7);

```
int snprintf (char* dst, int len, char* fmt, …)
{
    void *argptr = &fmt + 1;
    ...
    ...
}
```

Stack:
```
snprintf(dst, sizeof(dst), "%s: %d minutes to %x\n", username, nminutes, 7);
```

As the format-specifiers are encountered, parameters are retrieved from the stack.

Format String Attack Examples:

Example-1: Too few parameters to printf function:
--Information leakage:

```
void caller(…)
{
    int secret;
    printf("%d");
}
```

Stack:
```
secret
fmt
return addr
argptr
```
Example-2:

printf("%s", username); vs. printf(username);
username: "\%x\%x\%x\%x...\%x";

stack:

Thus, attacker can peek into the stack contents. Attacker can also figure out the exact location on the stack and may use it for the buffer overflow attack.

Example-3: Reading arbitrary memory location:

void caller(…)
{
    int dummy;
    char username[512];
    printf(username);
    ...
    ...
}

stack:

username: "<0x12345678>\%x%\%s"

1. When \%x is encountered, dummy is retrieved from stack.
2. When \%s is encountered, contents of the memory location <0x12345678> are read because argptr points to username on stack.

username on stack:
Example-4: Writing to arbitrary memory location:

“%n” format-specifier: Causes the number of characters written so far to be put into the integer.

```
printf("Hello: %n%s%n", &ctr1, username, &ctr2);
```

7 7 + strlen(username)

Attack:

```
void caller(…)
{
    int dummy;
    char username[512];
    printf(username);
    ...
    ...
}
```

stack:

```
username
dummy
fmt
return addr
argptr
```

username: 

```
"<0x12345678>%x%n"
```

memory location where attacker wants to write

1. When %x is encountered, dummy is retrieved from stack.
2. When %n is encountered, the number of characters written so far is written to memory location 

<0x12345678>.

To write value “12” at this memory location:

username: 

```
"<0x12345678>%8x%n"
```

* 0x12345678 = 4 + 8 (no of bytes for dummy)

Similarly to write value “134”,

username: 

```
"<0x12345678>%130x%n"
```

* 0x12345678 = 4 + 130