

CSE-509: System Security

Spring'2009

Lecture Notes: Integer Overflows and Format Strings Bugs

2/20/2009

1. Integer Overflows

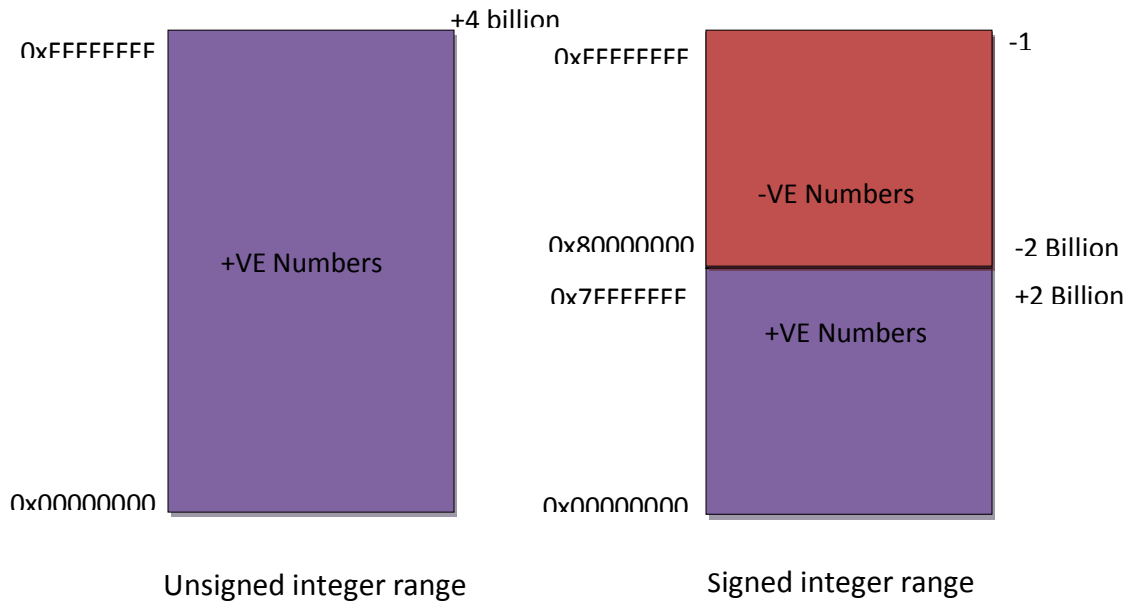
Integer overflows happens when an arithmetic operations generates a value larger than maximum representable value by a data type. Integer overflow can happen in any programming language.

```
int Main()
{
    int x, y, z;
    x = 0x7FFFFFFF; // this represent +ve 2 billion
    y = 1;
    z = x + y;      // Integer overflow

    printf("%d", z); // z prints -ve 2 billion
}
```

The above example will result into a signed addition overflow with z containing a negative value. The x has a the maximum possible positive value (2 billion) which can be represented by an signed integer and when y is added to it, causes overflow and creates a negative 2 billion number assigned to z.

In C programming language, Signed Integer overflows have undefined behavior while in unsigned integer overflow causes the number to wrap around.



Another example of integer overflow can happen when developer assigns a larger represented value by a data type to smaller one

```
int main()
{
    int x;
    short y;
    x = 100000;
    y = x;    // Integer overflow
}
```

“short” can store a 16 bit integer with maximum positive value of 32,767. In above case, a value more than that is been assigned to y which may result into negative number.

Attacks using Integer Overflows

```
Fun()
{
    unsigned int x;
    int y;
```

```

read(sockfd,&x,sizeof(x));

y = x + sizeof(PktHdr);    // causes overflow with y having a -ve value

if( y < MAX_OFFSET)
    table[y] = 0x00;
}

```

In a network packet handler Fun(), when it reads a value into a unsigned integer variable “x” , attacker can send a very large value say 0x8FFFFFFF which results a overflow causing “y” to have some negative value, which is indexed into an array table[y] causing some random memory access.

2. Format String Bugs

A format function is a special kind of C function that takes a variable Number of arguments, along with format string . While the function evaluates the format string, it accesses the extra parameters given to the function. These format function are used for printing the contents of the variables on the screen.

- fprintf — prints to a FILE stream
- printf — prints to the ‘stdout’ stream
- sprintf — prints into a string
- snprintf — prints into a string with length checking
- vfprintf — print to a FILE stream from a va_arg structure
- vprintf — prints to ‘stdout’ from a va_arg structure
- vsprintf — prints to a string from a va_arg structure

e.g : printf(“Hello World\n”);
 snprintf(dst,sizeof(dst),”%s:%d minutes to %x\n”,username,minutes,7);

```

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

```

How does format functions works?

1. format strings controls the behavior of the format function.
2. Parameters are saved on to the stack
3. Saved directly (by value) or indirectly (by reference)

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

When `snprintf` function is called then the stack frame looks like as shown in Fig-2

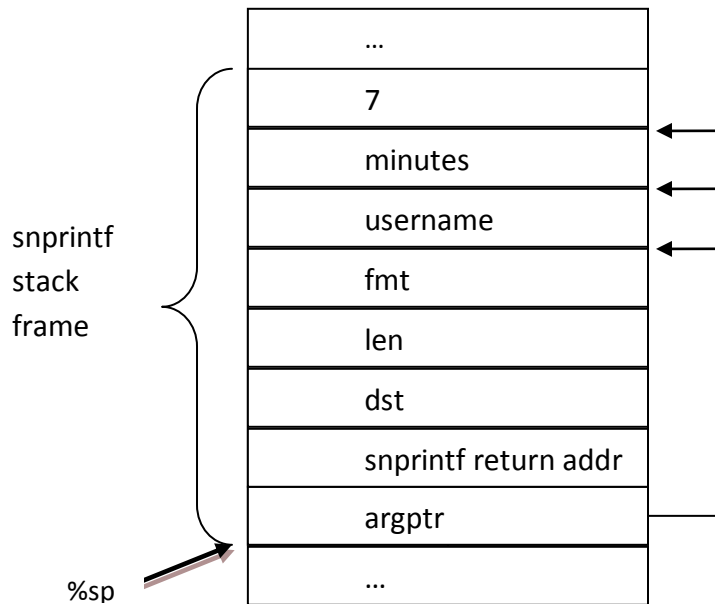


FIG - 2

Initially the `argptr` points to `username` parameter. `snprintf` will read the format string and when it encounters an format parameter "`%s`" it will print the `username` buffer contents and increment the `argptr` to point to next parameter `minutes` on stack so when it reads the next format parameter "`%d`" it will print the minutes and move the `argptr` to next parameter on stack and so on.

Attacks using Format Strings

1. Viewing the contents of the stack

```

void Caller()
{
    int secret;
    printf("%d");
}

```

When printf() function is executed, then *argptr* will point to *secret* on the stack, and since a format string "%d" has been specified, printf will print contents of the variable *secret* on to the console as shown in Fig-3.

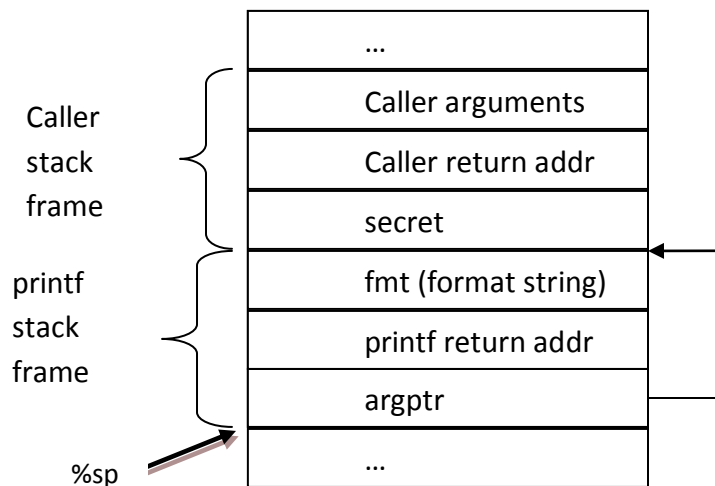


FIG - 3

Attacker can use this technique to display the contents of the stack which can be helpful to know the return pointers helpful for performing buffer overflow attacks.

These attacks are more vulnerable when a developer passes a buffer which may be accessible to attackers as an format string argument to printf rather using it as an parameter to print its contents.

```
printf("%s",username);
```

correct way of displaying contents of username

Vs

```
printf(username);
```

wrong way of displaying the username because it may contain some random format strings passed by attacker which may cause invalid memory access and may cause program to crash.

For e.g,

```
Username = "%s%s%s%s%s%s%s%s%s%s%s%s%s"
```

2. Reading arbitrary memory

Using the format strings, attackers can also read the memory locations other than stack memory. To do this, attacker needs to supply an arbitrary address whose contents could be printed using "%s". For this attacker needs to somehow get the address onto the stack.

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

Now, suppose that in above code username contains

```
username = "<0x12345679> %x %s"
```

<0x12345679> is a memory address whose contents attacker wants to read, %x is used to move the argptr to point to username on the stack. Now, argptr points to stack memory containing <0x12345679> which is used by "%s" (address by reference) to dump contents of the memory address at <0x12345679>.

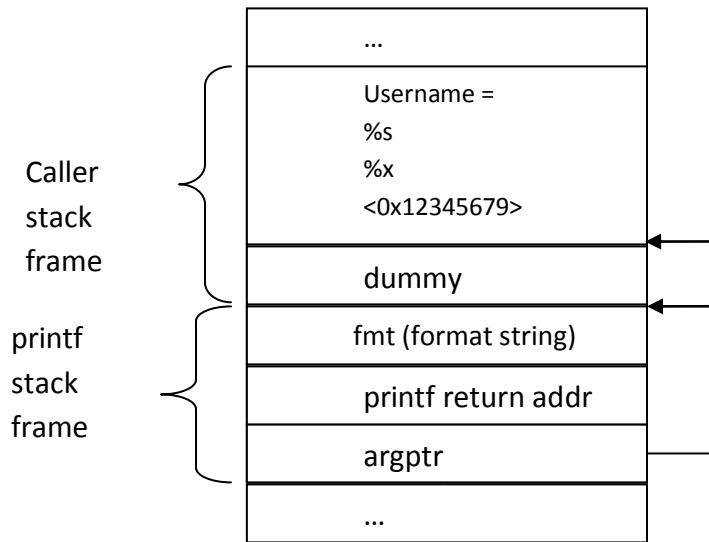


FIG - 4

3. Writing to arbitrary memory address

Attackers can also use format strings to write to any random memory using “%n” parameter . It writes the number of bytes already printed, into a variable whose address is given to the format function by placing an integer pointer as parameter onto the stack.

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

```
ctr1 = 7 and
ctr2 = 7 + strlen(username)
```

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

Now, suppose that in above code username contains
username = "<0x12345679> %x %n"

Here, <0x12345679> is a memory address to which attacker wants to write some data, %x is used to skip dummy and move the *argptr* to point to username on the stack. so, next it encounters a "%n" and the *argptr* would be pointing to memory address containing <0x12345679> which would be then referenced by "%n" to write 8 bytes.

***(0x12345679) = 8**

Here, attacker has reached a goal of writing to an arbitrary memory address but does not have the control over the number written into the location. To write a value of choice, attacker can specify the numbers of characters to be printed out using below examples

username = "<0x12345679> %8x%n"

***(0x12345679) = 12**

However, the above approach too is not sufficient to write very large values such as addresses.

On Intel based x86 architectures have a little endian format for storing data with into memory with least significant byte stored at lower memory address and most significant byte stored at higher memory address.

In order to write 4 bytes of addresses, attackers can write one byte at a time for four times in a row. Suppose, attacker wants to write 0x44332211 at memory location 0x12345679



He will start with writing least significant byte 0x11 at memory location 0x12345679 and then increment the memory location by 1 and then further writing next least significant byte and so on until all the four bytes have been written. This can be done using counter mod 256 to get least significant bytes to be passed as padding before “%n”.