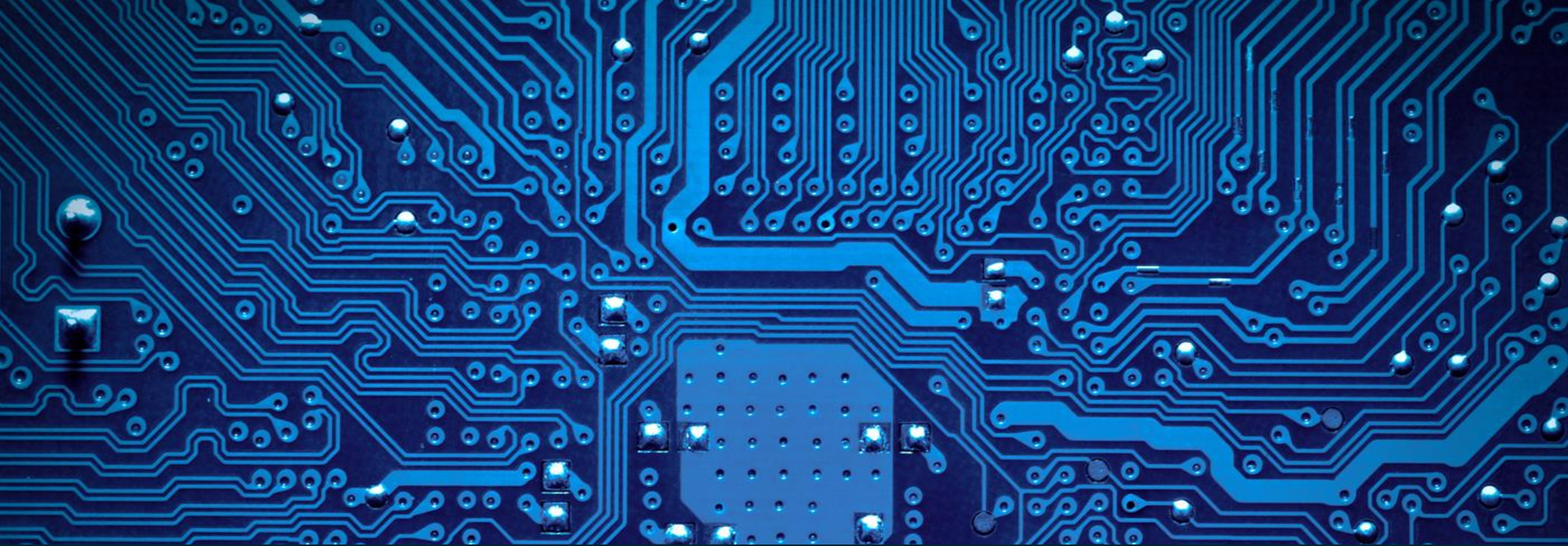


# **ECE 568:** Computer Security

## **Part 2B:** Format-String and Double-Free

Courtney Gibson, P.Eng.

UNIVERSITY OF TORONTO

# Part 2B

## Format-String and Double-Free Vulnerabilities

- Format String Vulnerabilities
- Format String Exploits
- Double-Free Exploits

# Format-String Vulnerabilities

UNIVERSITY OF TORONTO

# Format String Vulnerabilities

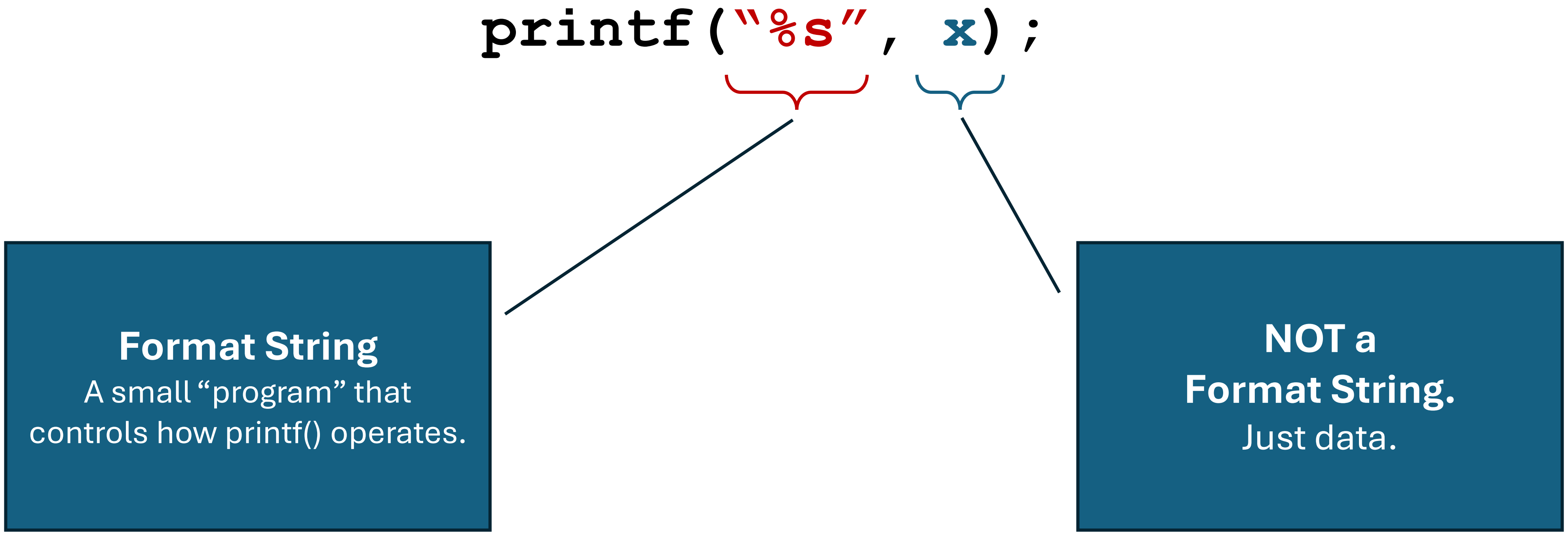
A simple format string vulnerability, similar to ***strcpy***, that can result in a buffer overflow:

```
sprintf(buf, “WARNING: %s”, attacker_string);
```

**sprintf** is similar to **printf**, except that the output is copied into a buffer (instead of printed on the screen).

# Format String Vulnerabilities

Reminder:



```
printf(“%s”, x);
```

**Format String**
A small “program” that controls how printf() operates.

**NOT a Format String.**
Just data.

# Format String Vulnerabilities

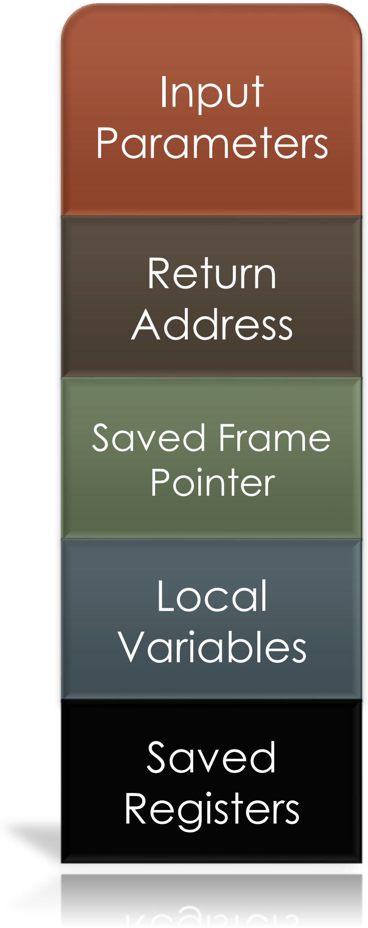
Confusing data and format-strings can lead to a more complex vulnerability:

```
snprintf(buf, len, attacker_string);
```

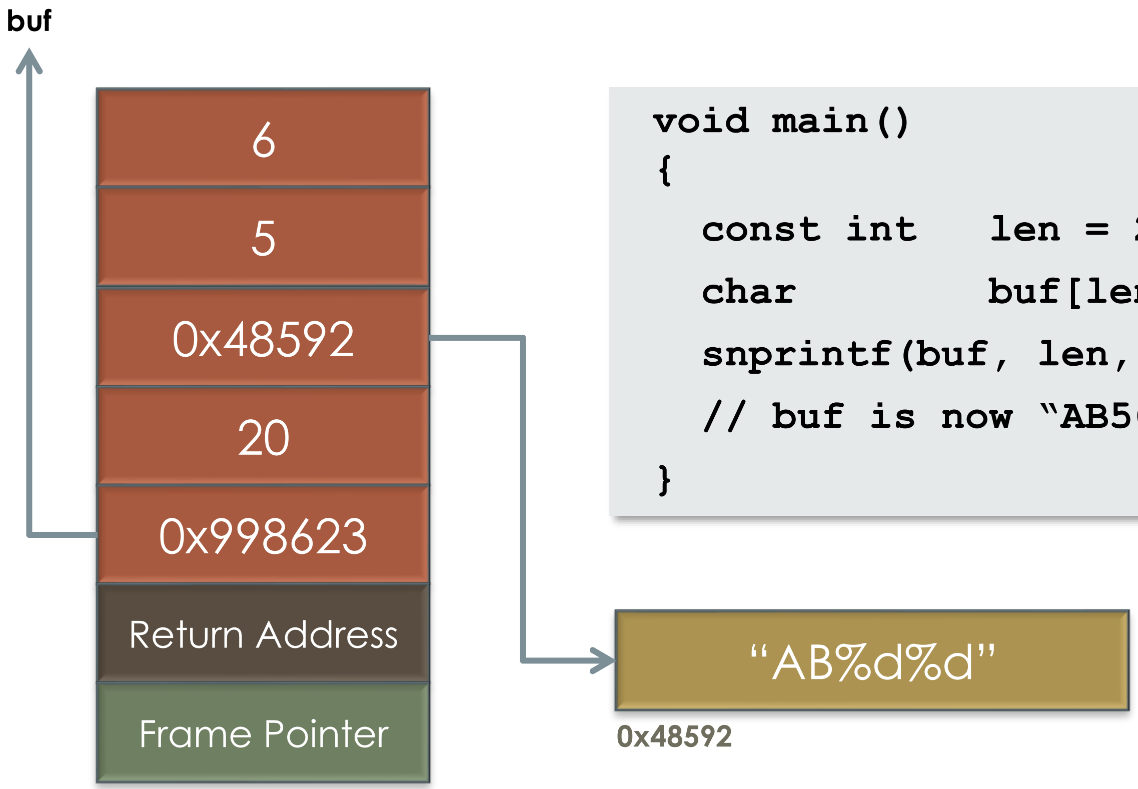
There is no buffer overflow risk, as **len** limits the number of characters written into **buf**...but, the attacker gets to specify the **format string**.

**Example:** Application logs, language configs, locale files, etc..

# **Recall:** Stack Frame



# snprintf()



```
void main()
{
   const int   len = 20;
   char        buf[len];
   snprintf(buf, len, ”AB%d%d”, 5, 6);
   // buf is now “AB56”
}
```

- Arguments are pushed to the stack in reverse order
- **snprintf** copies data from the format string until it reaches a ‘%’. The next argument on the stack is then fetched and output in the requested format

# snprintf: Unexpected Behaviour

- What happens if there are more ‘%’ parameters than arguments?
- The argument pointer keeps moving up the stack, and points to values in the previous frame!

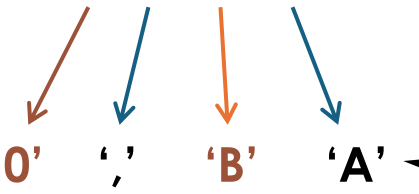
```
 void main() {
   char        buf[256];
   snprintf(buf, 256,
       ”AB,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x\n”, 5);
   printf(buf);
}
```

# snprintf: Unexpected Behaviour

```
 void main() {
   char         buf[256];
   snprintf(buf, 256,
        ”AB,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x\n”, 5);
   printf(buf);
}
```

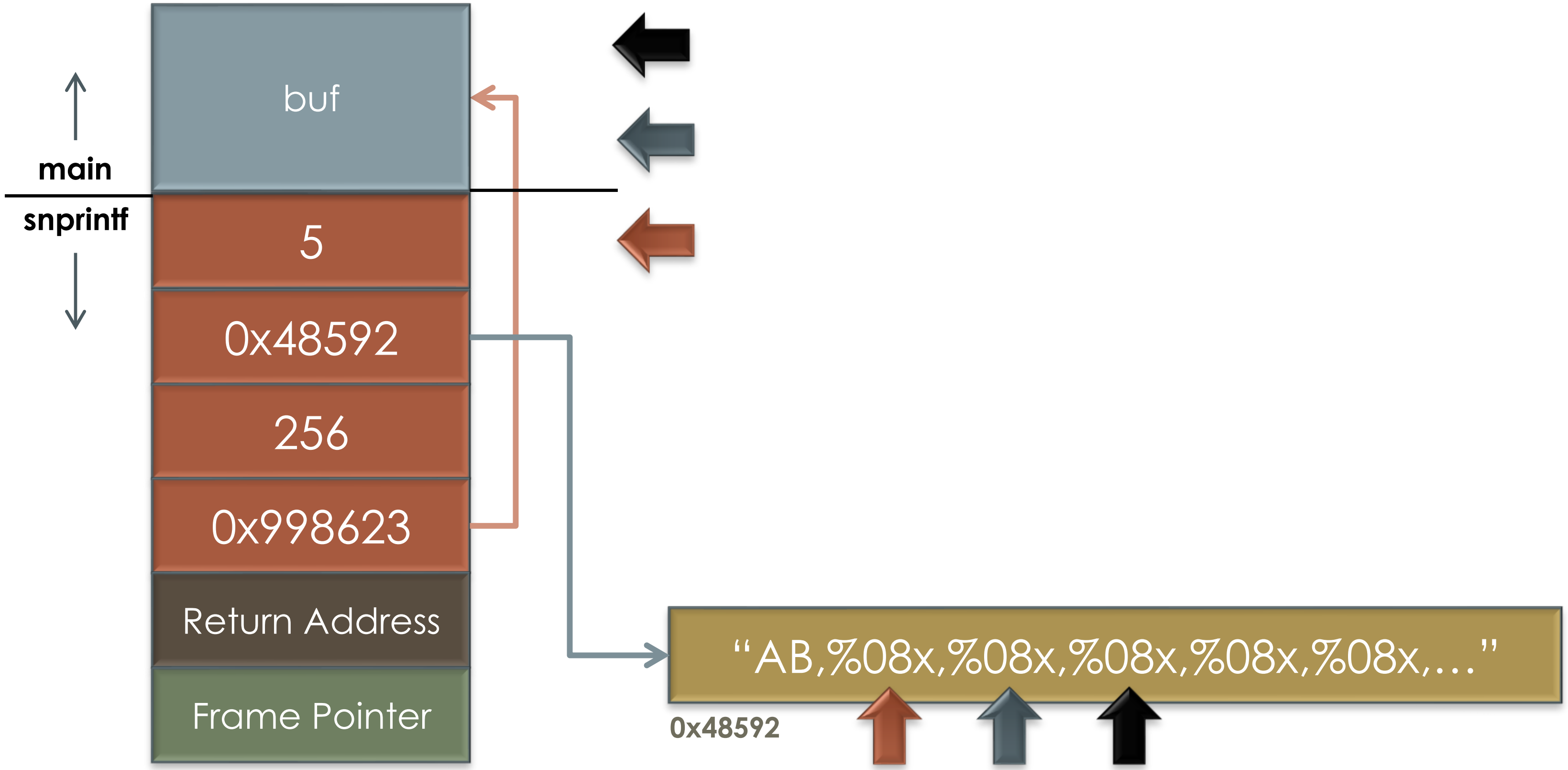
The output of the program is:

```
AB,00000005,302c4241,30303030,2c353030,...
```



‘0’ ‘,’ ‘B’ ‘A’

This is **buf**: the argument pointer has worked back into **main**’s stack frame.



```
void main() {
  char        buf[256];
  snprintf(buf, 256,
         "AB,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x\n", 5);
  printf(buf);
}
```

# Information Leakage

- If there is valuable information further up the stack (*e.g.*, passwords, encryption keys, etc.), then there is a significant risk of information leakage.
- Programmers may not pay attention to sanitizing input like language config:

```
<param name=”lastLogin" value=”Votre dernière connecté il ya %d jours"/>
```

# Format-String Exploits

UNIVERSITY OF TORONTO

# Overwriting the Return Address

Rather than just leak information, can we inject an exploit?

**Yes!**

In most C “print” functions, “**%n**” assumes the current argument is a **pointer**; the number of characters written so far are copied to that address.

# Overwriting the Return Address

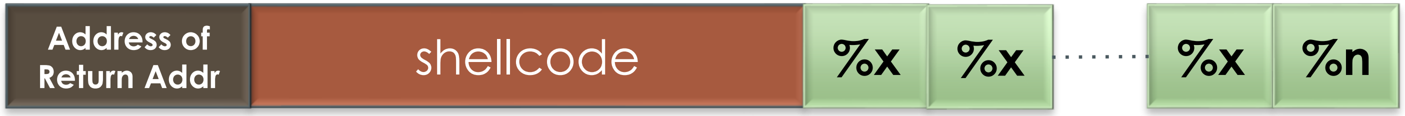
```
…
int numBytes;

printf (“Hello world%n\n”, &numBytes);
…
```

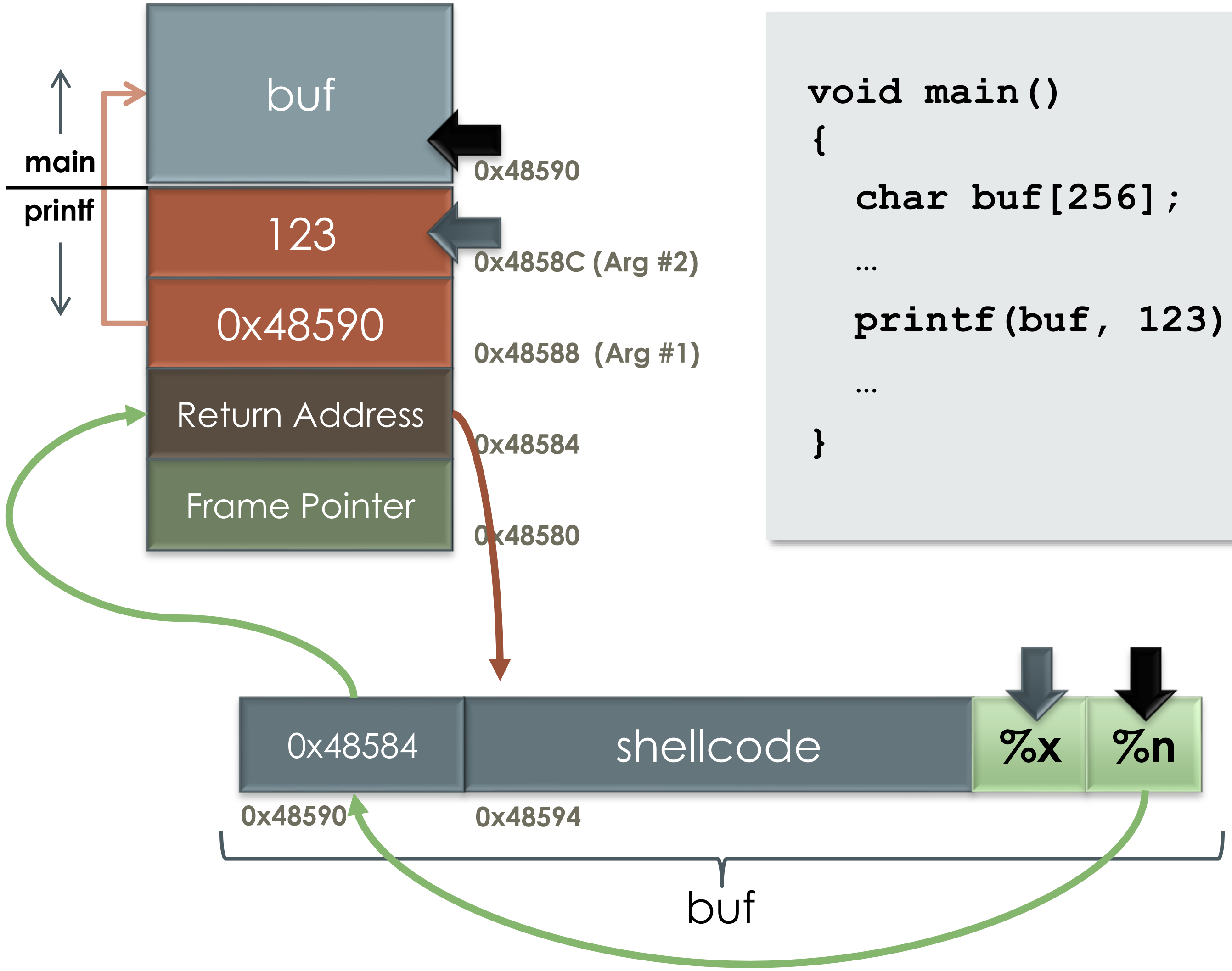
numBytes = 11

- Normally, “%” arguments tell printf() to **read** values… but **%n** modifies the memory pointed to by the argument!
- We can take control of the program if a %n argument points to the saved return address on the stack

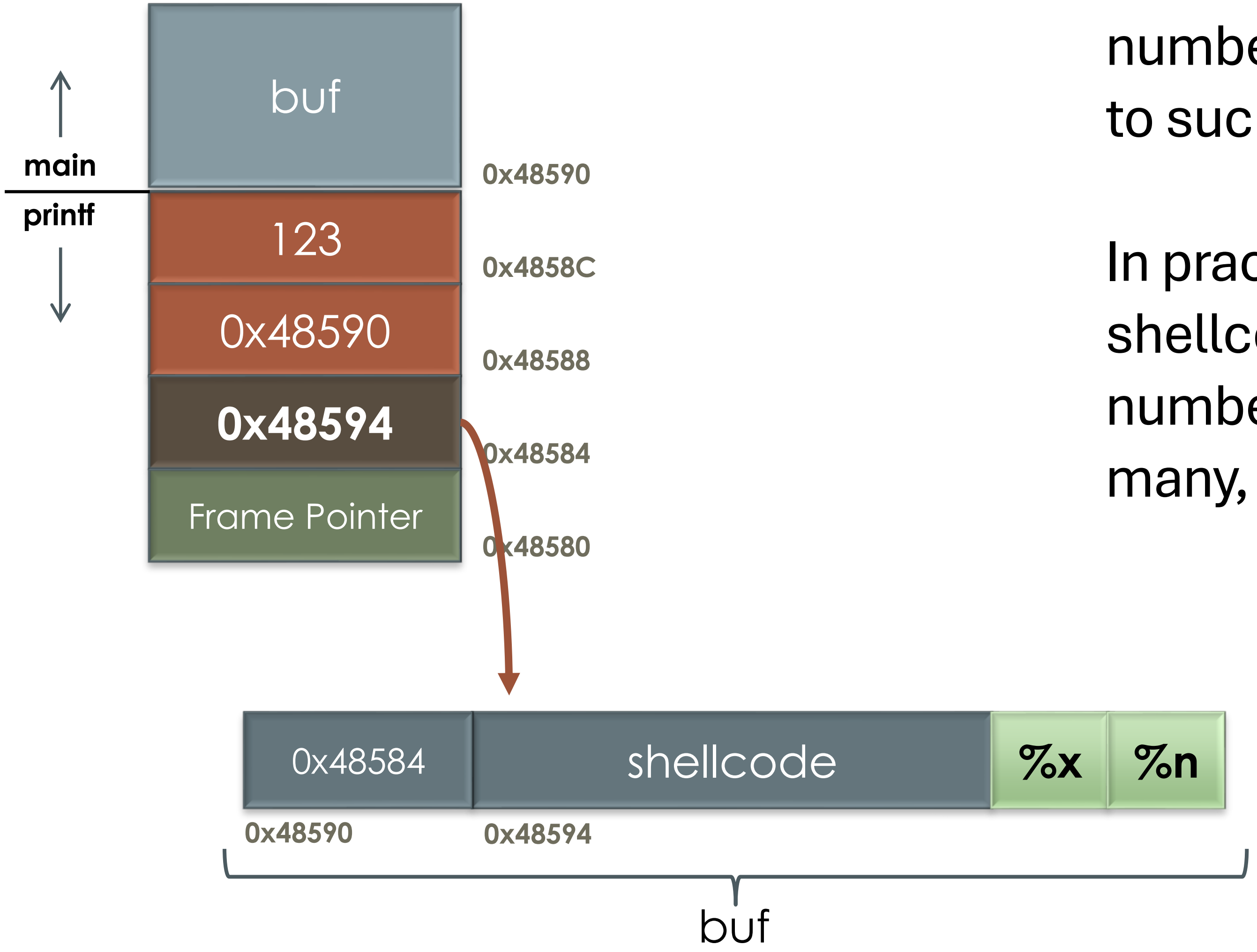
# Exploiting Format String Vulnerabilities

- At the front of your format string, put the address where you think the **return address** is stored on the stack
- Put your shellcode in the format string
- Put enough “%” arguments so that the argument pointer points to the front of your format string
- Put a **%n** at the end and overwrite the return address to point at the shellcode in the buffer





```
void main()
{
  char buf[256];
  …
  printf(buf, 123);
  …
}
```



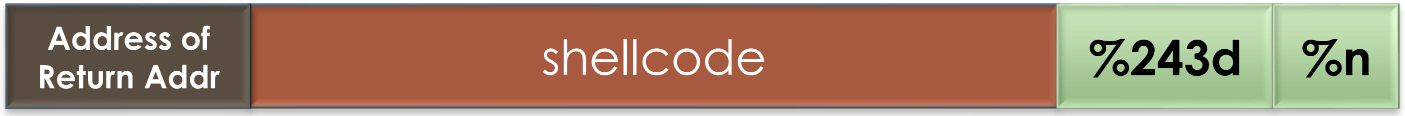
**Problem:** How do we get %n (the number of printed characters) up to such a high value?

In practice, the address of our shellcode will be a **very large** number. This would require printing many, many bytes.

# Overwriting the Return Address

The number of characters written can be controlled by adding a **width** argument between **%** and **x**, **u** or **d**.

**Example:** “%243d” writes an integer with a field width of 243; “%n” will be incremented by 243.



Address of Return Addr

shellcode

%243d

%n

# Overwriting the Return Address

In practice, though, the stack addresses are **really, really** large values. It is likely not practical to use %n to overwrite the return address with a large 32-/64-bit number:

- Would require **printf** to produce multiple GB (or hundreds of TB) of output: likely will not fit in memory
- Often, large “width” values will crash the program

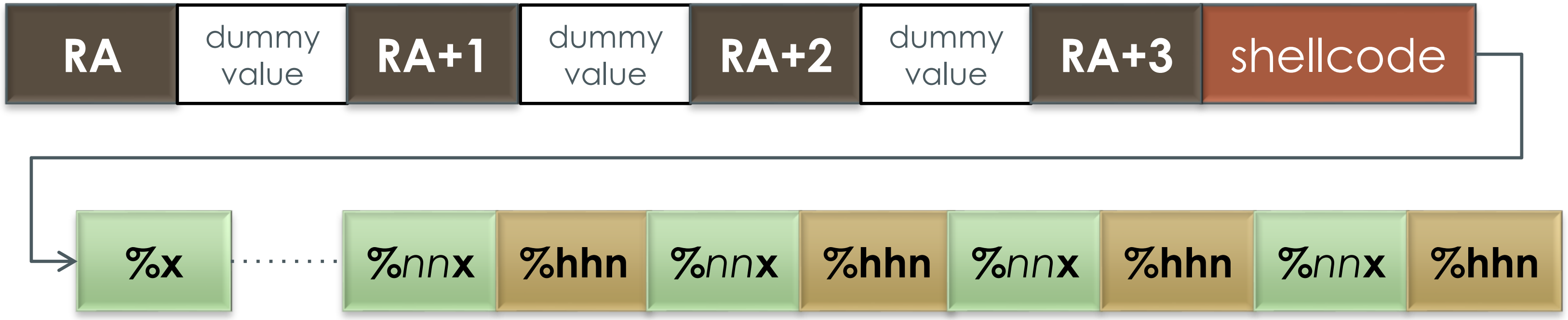
# Overwriting the Return Address

Fortunately, the 32-bit (or 64-bit!) return address can be written one byte at a time:

- Use just the lowest-order byte stored by “%hhn”
- Incremented with modulo-256 arithmetic

For more information:

- “*Exploiting Format String Vulnerabilities*” on the course website



# What Happens with a Size Limit?

Can the size limit in **snprintf** stop this attack?

```
snprintf(buf, len, formatString, …);
```

Unfortunately, no.

**snprintf** will interpret the whole format string, regardless of the size limit:

- If output is longer than **len**, it is truncated before writing to **buf**
- **%n** is always evaluated, and assumes that there is no size limit in place

# Double-Free Exploits

UNIVERSITY OF
TORONTO

# Double-Free Vulnerability

Freeing a memory location that is under the control of an attacker can result in an exploitable vulnerability:

```
p = malloc(128);
q = malloc(128);
free(p);
free(q);
p = malloc(256);
strcpy(p, attacker_string);
free(q);
```

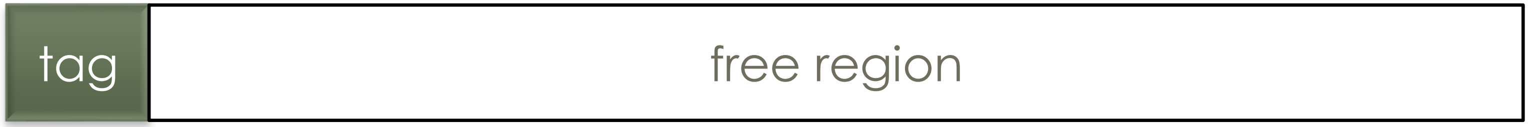
Why is this a vulnerability?

Let’s look at how **malloc** works…

# malloc() implementation

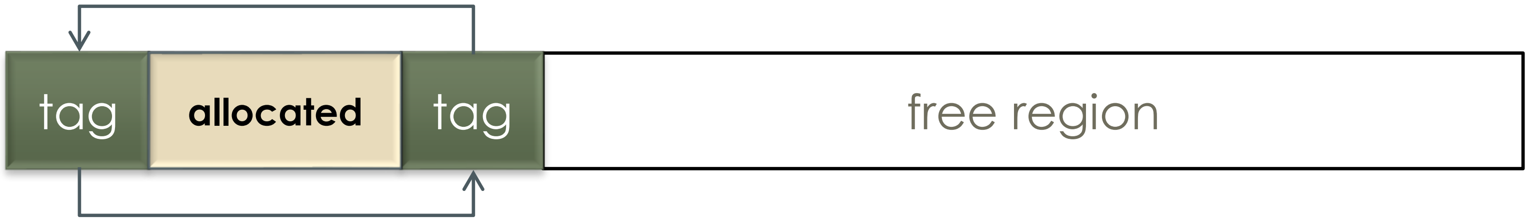
**malloc** maintains a doubly-linked list of free and allocated memory regions:

- Information about a region is maintained in a **chunk tag** that is stored just before the region
- Each chunk maintains:
  - A “free bit”, indicating whether the chunk is allocated or free
  - Links to the next and previous chunk tags
- Initially when all memory is unallocated, there is just one free memory region:

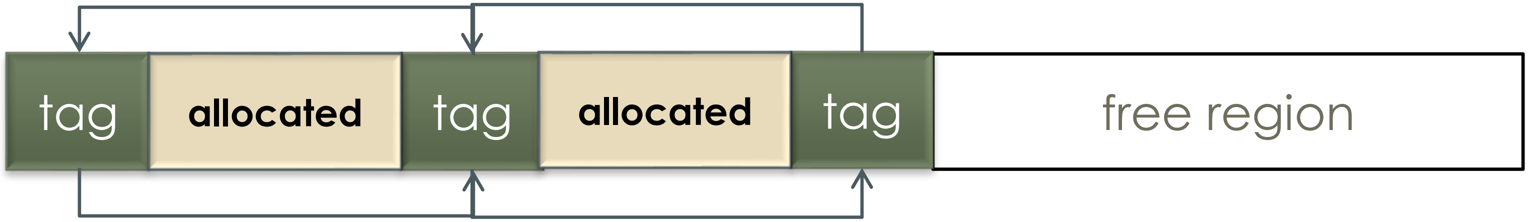


# malloc() implementation

When a region is allocated, **malloc** marks the remaining free space with a new tag:

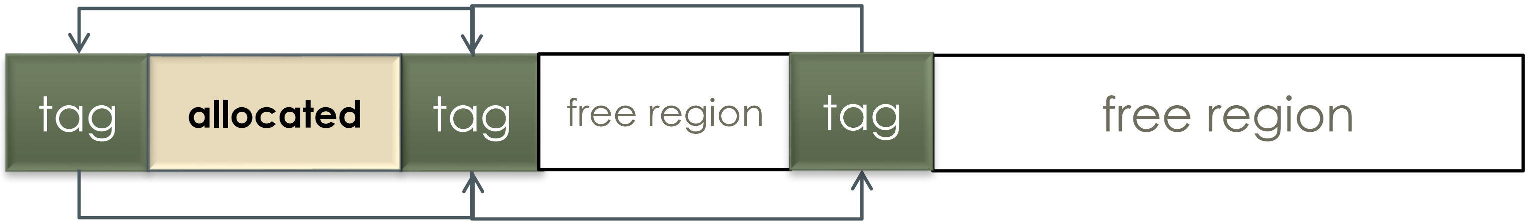


When another region is allocated, another tag is created:

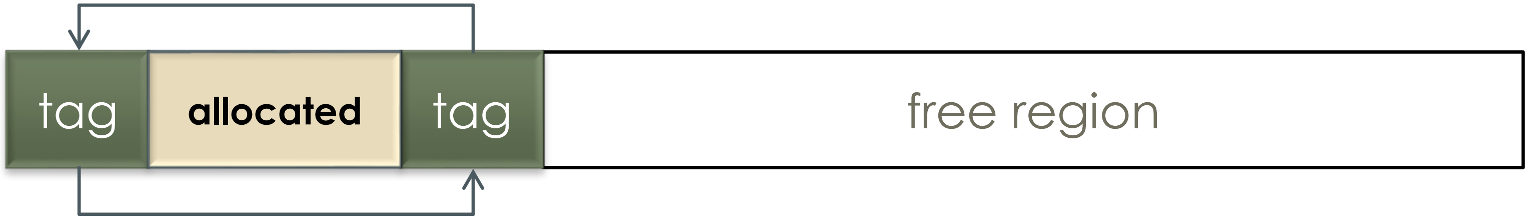


# free() implementation

When regions are de-allocated, the **free** function sets the "free bit":



**free** also tries to consolidate adjacent free regions:

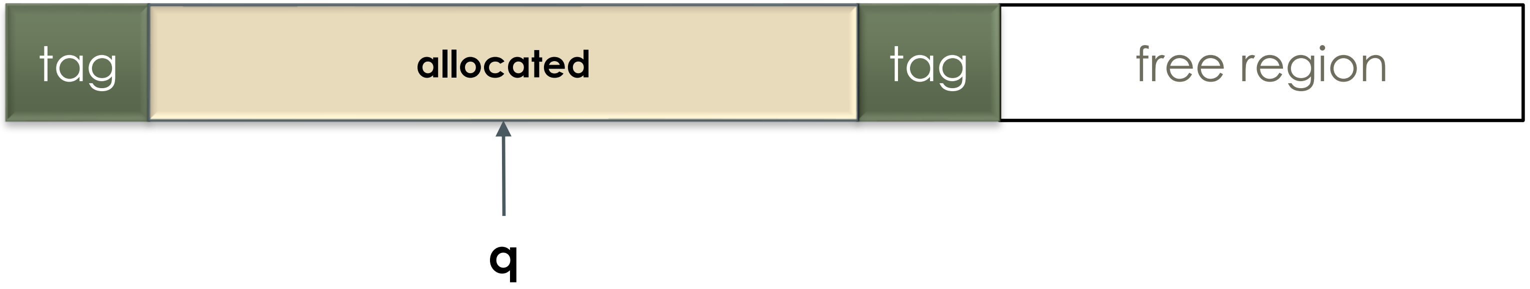


# Double-Free Vulnerability

A vulnerability occurs when the program calls **free** on a region that contains data provided by the attacker:

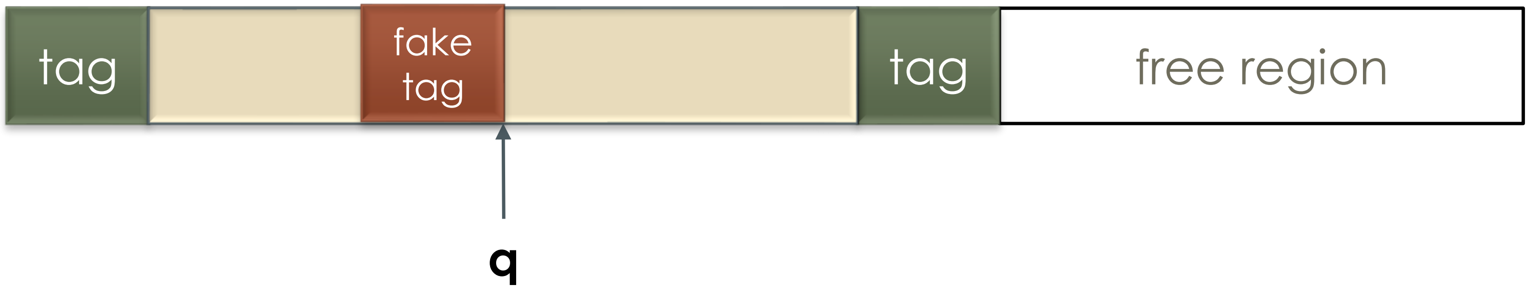
- **free(q)** assumes there is a valid chunk tag located just before the address pointed to by **q**

- If the attacker provides a fake “chunk tag” in the correct location, then it will be processed by the memory-management library:

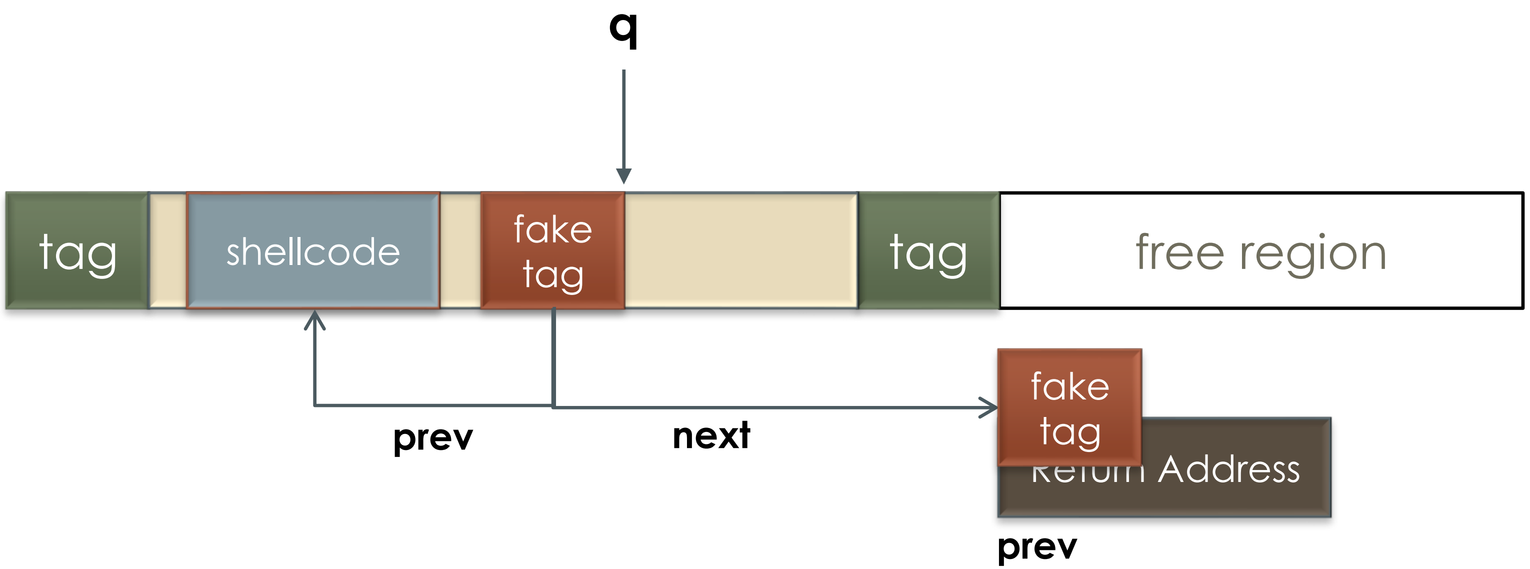


# Double-Free Vulnerability

The attacker can set the values in the fake “chunk tag” such that **free** will overwrite a memory location chosen by the attacker with a value chosen by the attacker.



# Double-Free Vulnerability



When consolidating free regions, **free** essentially does:

```
tag = q - sizeof(chunkTag);
tag->next->prev = tag->prev;
```

# Questions?

UNIVERSITY OF TORONTO