

Format String & Double-Free Attacks

ECE568 – Lecture 5
Courtney Gibson, P.Eng.
University of Toronto ECE

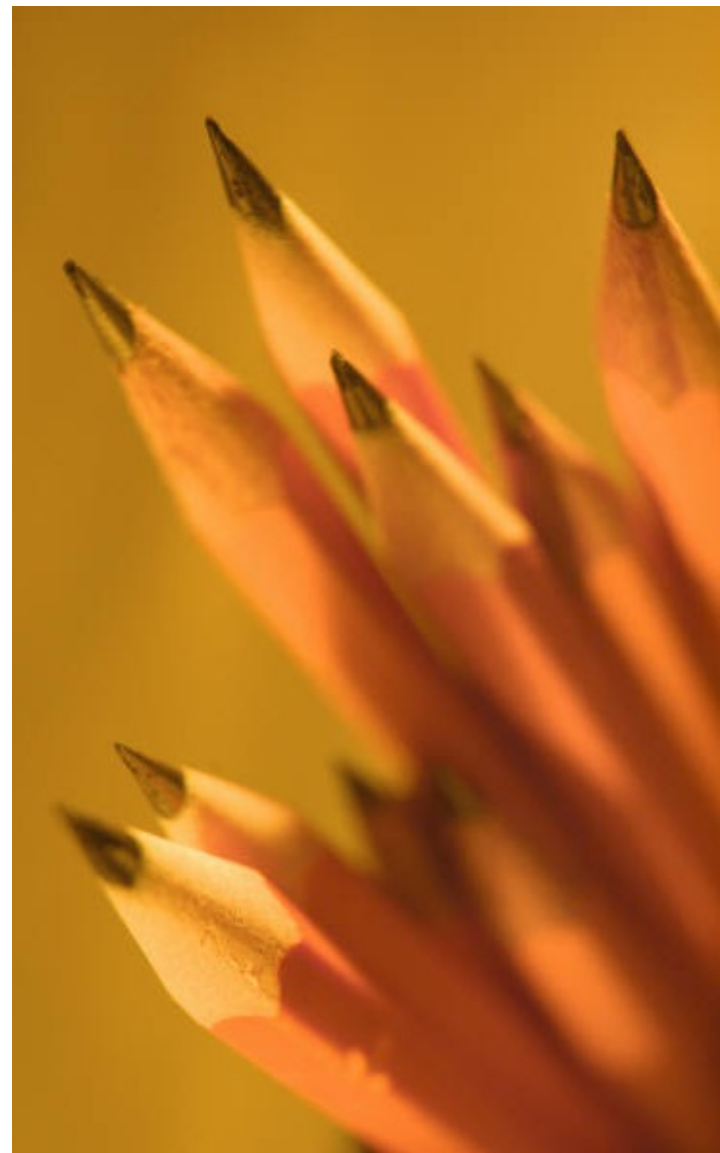
Outline

Format String Attacks

- Information leakage
- %n vulnerability
- Crafting a format-string attack

Double-Free Attacks

- malloc / free implementation
- Double-free vulnerability



Format String Attacks

snprintf, information leakage,
%n, format string vulnerability

Format String Vulnerabilities

A simple format string vulnerability:

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

- **sprintf** is similar to **printf**, except that the output is copied into **buf**
- The vulnerability above is similar to strcpy, and can result in a buffer overflow

Format String Vulnerabilities

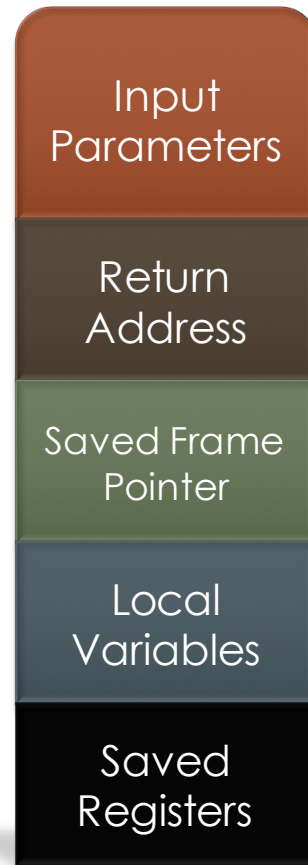
A more complex vulnerability:

```
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.

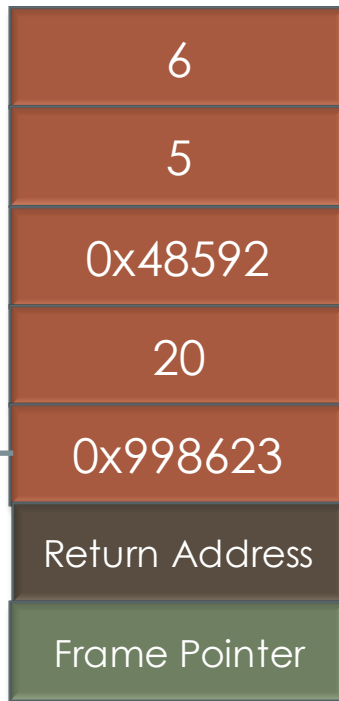
Example: Application logs, language configs, locale files, etc..

Recall: Stack Frame



snprintf Operation

buf



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

"AB%d%d"

0x48592

- 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

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\n", 5);  
    printf(buf);  
}
```

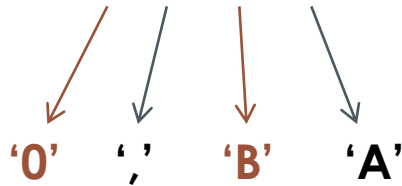

Unexpected Behaviour

```
void main() {  
    char        buf[256];  
    snprintf(buf, 256,  
             "AB,%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.

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"/>
```

Overwriting the Return Address

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

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

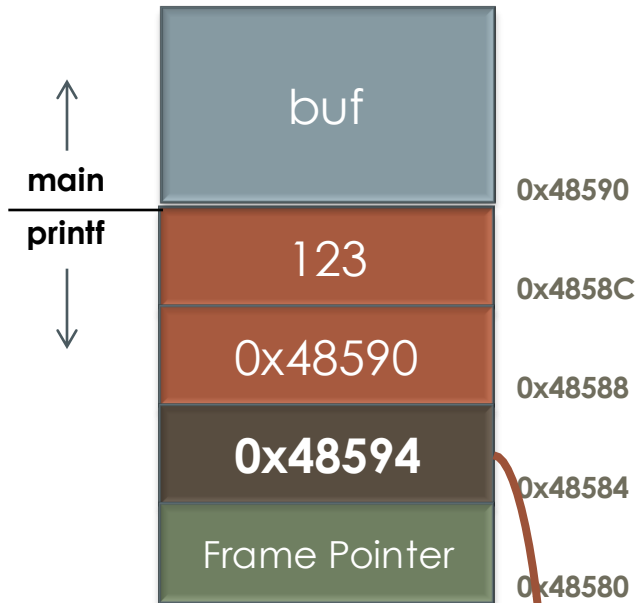
- Normally, “%” arguments **read** values, but %n modifies the memory pointed to by the argument
- We can take control of the program if a %n argument points to the saved return address on the stack

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

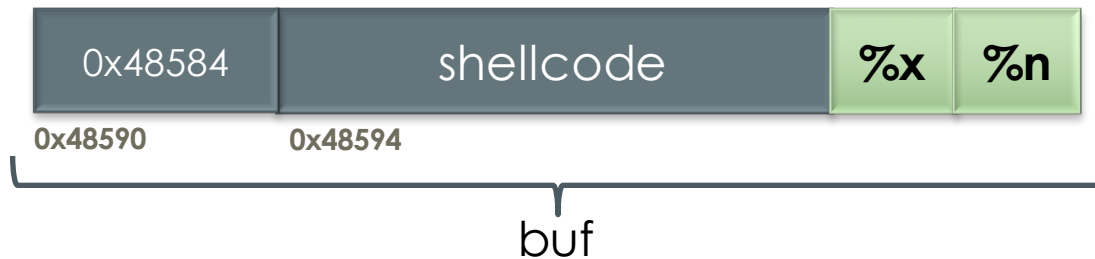


%n Vulnerability



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: would require printing many, many bytes: **buf** won't be large enough.



Overwriting the Correct 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.



Overwriting the Correct Return Address

In practice, though, the stack addresses are **really, really** large values; we need `%n` to overwrite the return address with a large 32-bit number:

- Would require **printf** to produce multiple GB of output: likely will not fit in memory
- Often, large width values will crash the program



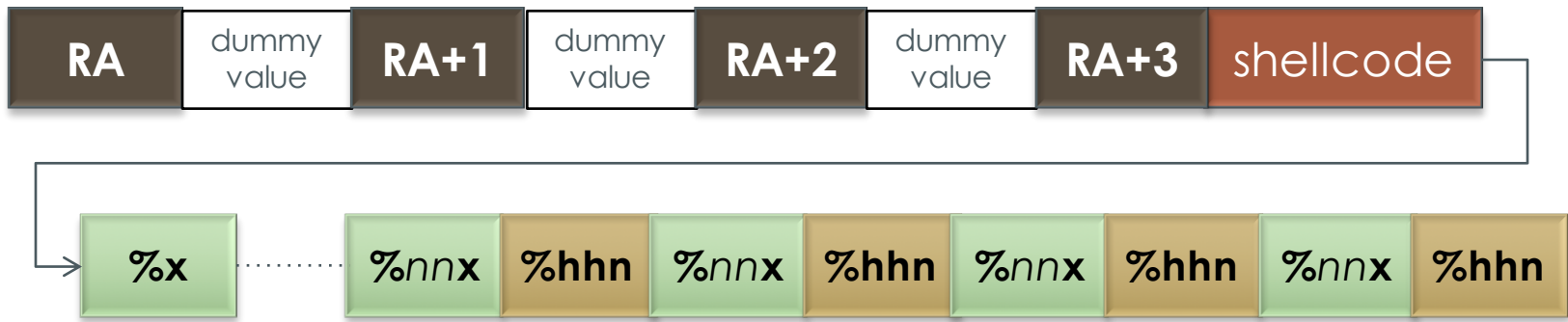
Overwriting the Correct Return Address

Fortunately, the 32-bit number return address can be written one byte at a time:

- Use just the lowest-order byte stored by “%hhn”
- It is 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, ...);
```

snprintf will interpret the whole format string, regardless of the specified 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 Attacks

malloc, free, allocation tags,
double-free vulnerability

Double-Free Vulnerability

Freeing a memory location that is under the control of an attacker is 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:

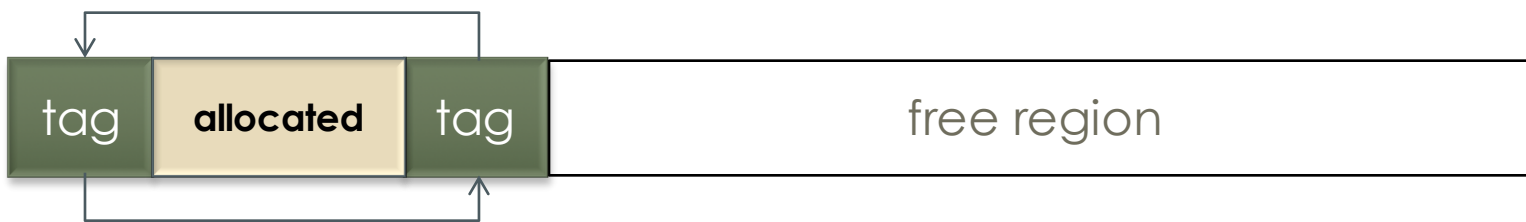
- Information about a region is maintained in a **chunk tag** that is stored just before the region
- Each chunk maintains:
 - A “free bit”, indicating whether the chunk is allocated or free
 - Links to the next and previous chunk tags
- Initially when all memory is unallocated, it is in one free memory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

tag

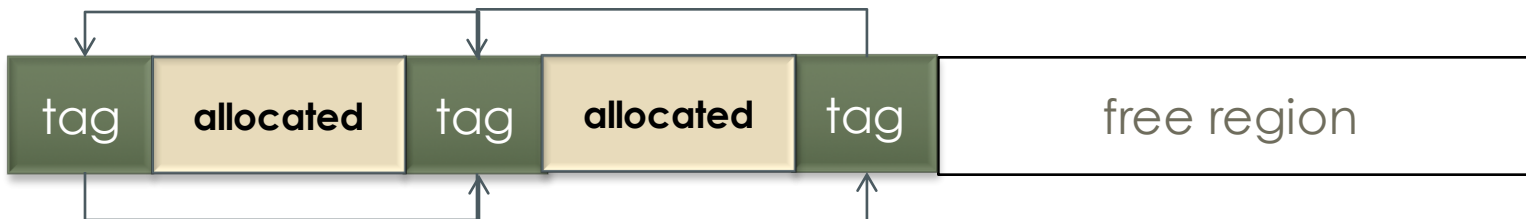
free region

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:

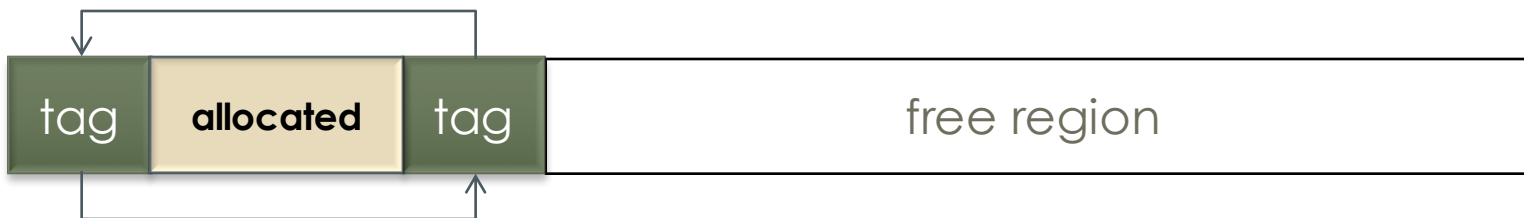


free Implementation

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



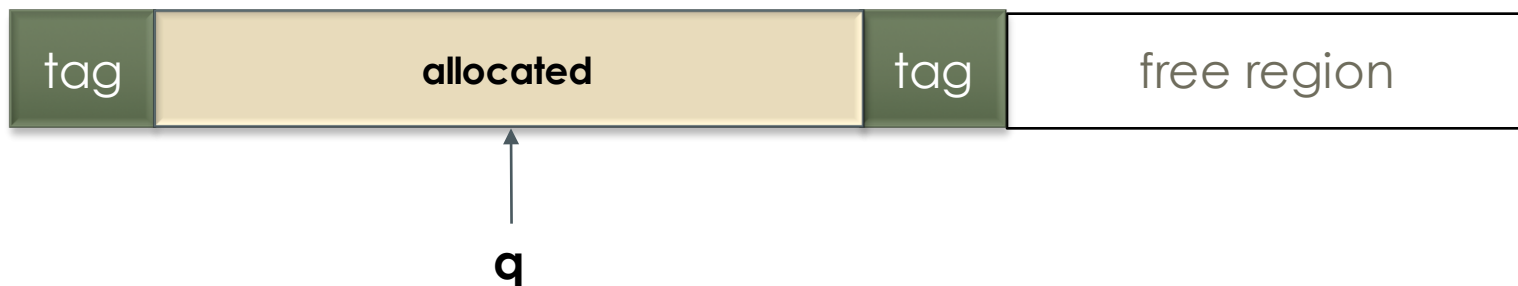
free also tries to consolidate adjacent free regions:



Double-Free Vulnerability

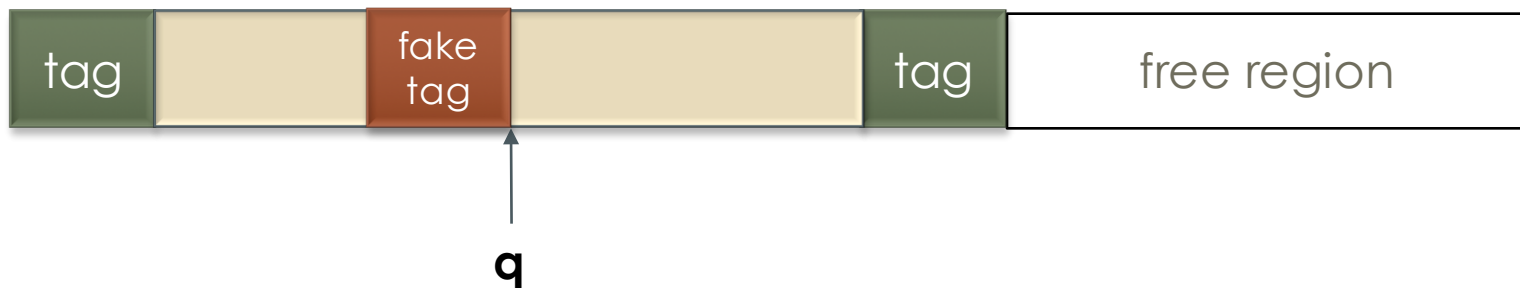
A vulnerability occurs when the program calls **free** on a region that contains data set by the attacker:

- **free(q)** will try to use the chunk tag located just before the address pointed to by **q**
- In this case, the “chunk tag” is now actually part of the attacker’s string

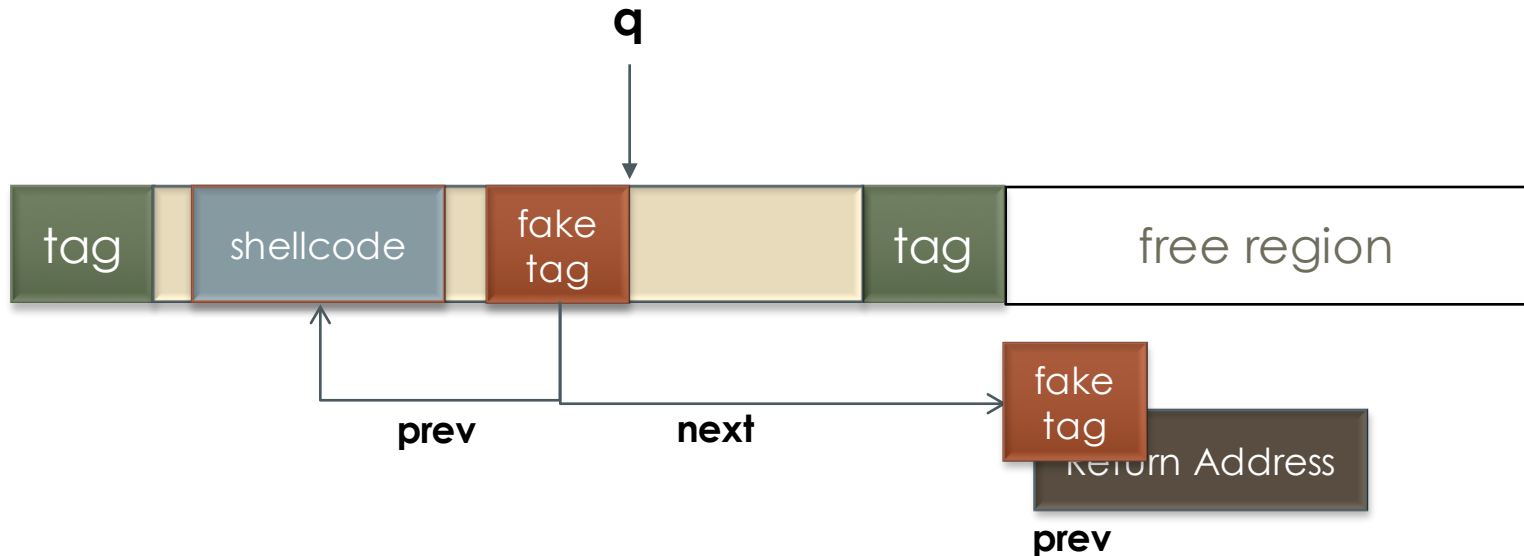


Double-Free Vulnerability

The attacker can set the values in their “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?