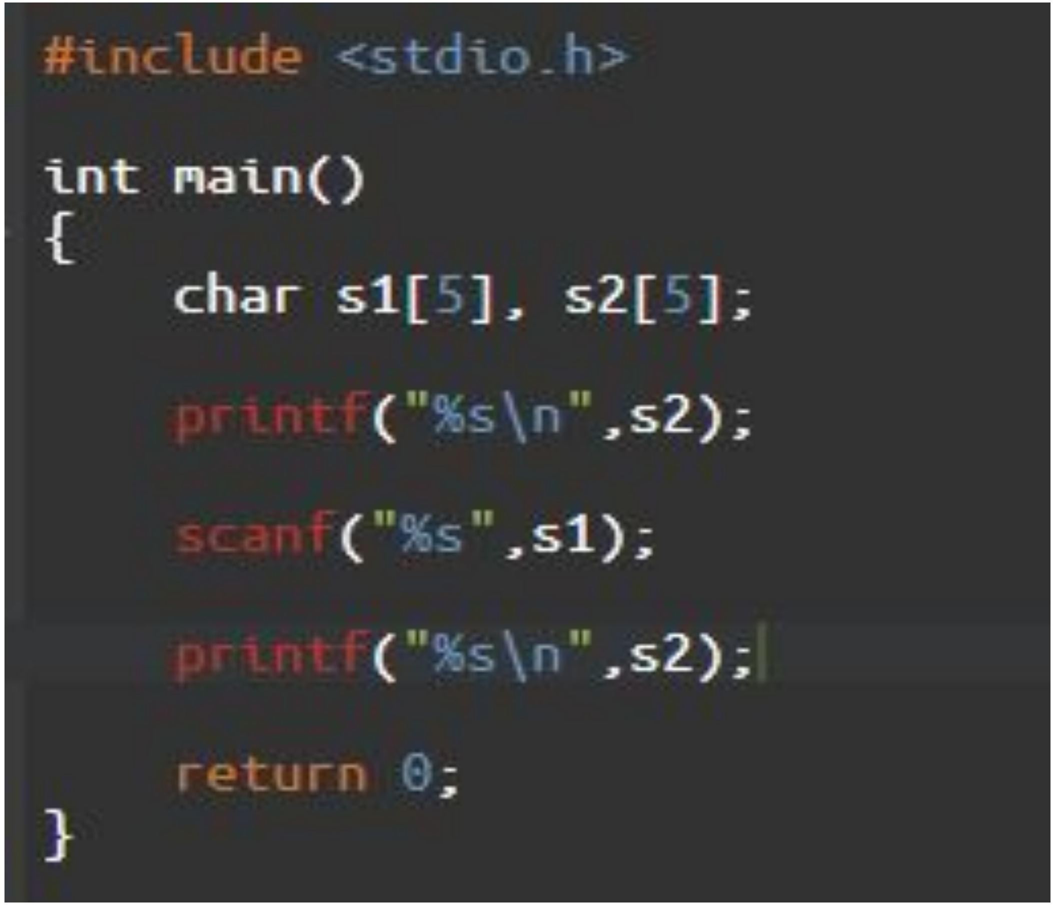
System Security

Dr. Amit Praseed

The Microsoft Dialer Exploit

- Microsoft Dialer was a program for dialing a telephone
- In 1999, security analyst David Litchfield was interested in this software
    - Dialer had to accept phone numbers of different lengths, given country variations, outgoing access codes, and remote signals
    - He tried dialer.exe with a 20, 25 and 50 digit phone numbers, and the program still worked fine.
    - When he tried a 100-digit phone number, the program crashed.
    - The programmer had probably made an undocumented and untested decision that nobody would ever try to dial a 100-digit phone number
    - The dialer.exe program is treated as a program call by the operating system, so by controlling what dialer.exe overwrote, we can redirect execution to continue anywhere with any instructions we want

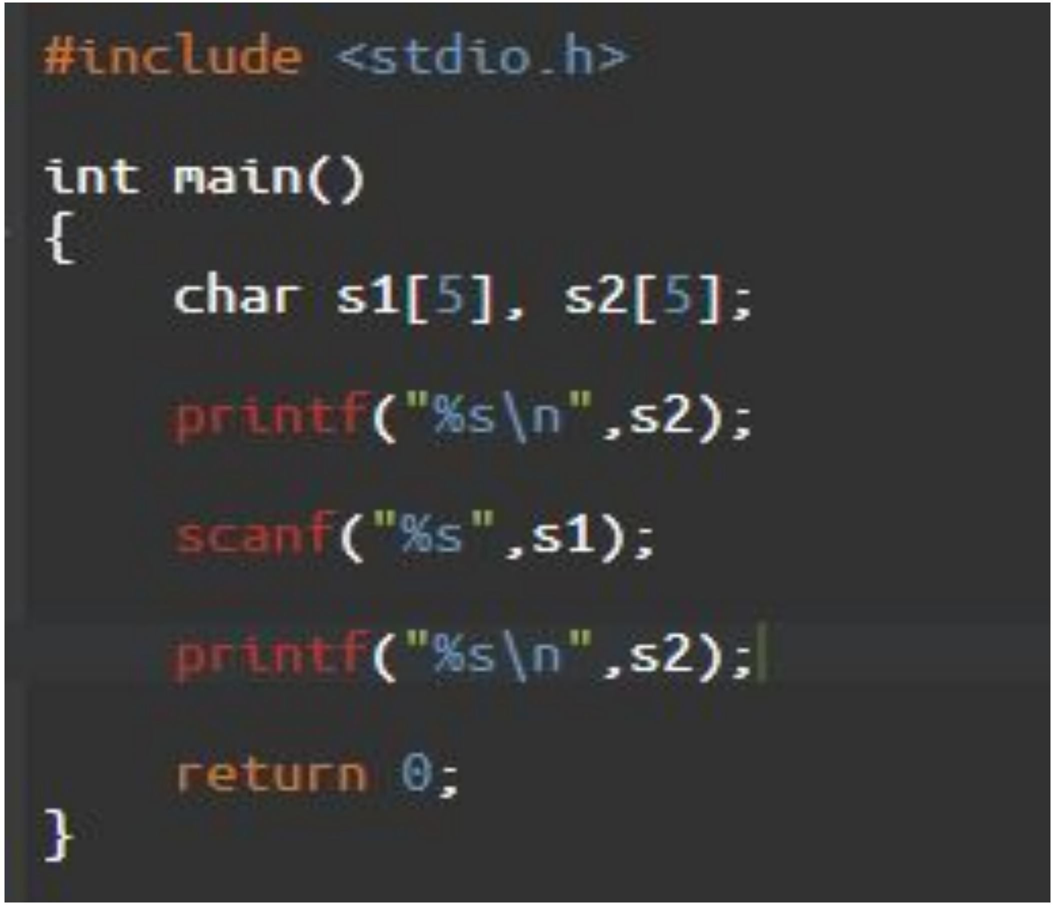
What will happen here...



```c
#include <stdio.h>

int main()
{
    char s1[5], s2[5];

    printf("%s\n",s2);

    scanf("%s",s1);

    printf("%s\n",s2);

    return 0;
}
```

Input: abcdefgh

What will happen here...



```c
#include <stdio.h>

int main()
{
    char s1[5], s2[5];

    printf("%s\n",s2);

    scanf("%s",s1);

    printf("%s\n",s2);

    return 0;
}
```

Input: abcdefgh

Output:

```
�
abcdefgh
fgh
```

Memory Allocation

- Memory is a limited but flexible resource; any memory location can hold any piece of code or data.
- To make managing computer memory efficient, operating systems jam one data element next to another, without regard for data type, size, content, or purpose
- Program counter indicates the next instruction - as long as program flow is sequential, hardware bumps up the value in the program counter
- Instructions such as IF, WHILE, FOR, GOTO or CALL divert the flow of execution, causing the hardware to put a new destination address into the program counter.
- Hardware simply fetches the byte (or bytes) at the address pointed to by the program counter and executes it as an instruction.
- **Instructions and data are all binary strings; only the context of use says a byte, for example, 0x41 represents the letter A, the number 65, or the instruction to move the contents of register 1 to the stack pointer**

Memory Allocation - The Security Aspect

- Hardware recognizes more than one mode of instruction - privileged instructions that can be executed only when the processor is running in a protected mode.
    - Trying to execute something that does not correspond to a valid instruction or trying to execute a privileged instruction when not in the proper mode will cause a program fault.
    - When hardware generates a program fault, it stops the current thread of execution and transfers control to code that will take recovery action
- In memory, code is indistinguishable from data. The origin of code (respected source or attacker) is also not visible.
- The attacker’s trick is to cause data to spill over into executable code and then to select the data values such that they are interpreted as valid instructions to perform the attacker’s goal.
- For some attackers this is a two-step goal: First cause the overflow and then experiment with the ensuing action to cause a desired, predictable result

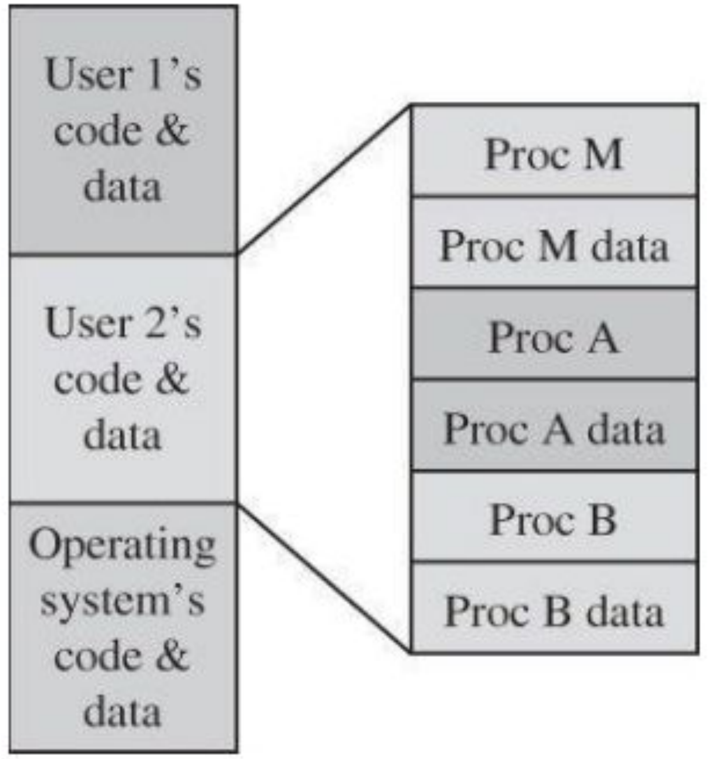
Harm from Overflows

- The attacker may replace code in the system space
  - Every program is invoked by an operating system that may run with higher privileges than those of a regular program
  - By replacing a few instructions right after returning from his or her own procedure, the attacker regains control from the operating system, possibly with raised privileges - **privilege escalation**
- The intruder may wander into an area called the stack and heap
  - By causing an overflow into the stack, the attacker can change either the old stack pointer or the return address
  - Changing the context or return address allows the attacker to redirect execution to code written by the attacker.

Implications of Overwriting Memory

- If the extra character overflows into the user's data space, it simply overwrites an existing variable value, perhaps affecting the program's result but affecting no other program or data
- If it overlaps an already executed instruction, the user should perceive no effect.
- If it overlaps an instruction that is not yet executed, the machine will try to execute an instruction with operation code corresponding to the overwritten data - if there is no instruction with operation code 0x42, the system will halt on an illegal instruction exception
- What happens if the system owns the space???

Implications of Overwriting Memory



A data overflow either falls strictly within a data space or it spills over into an adjacent code area. The data end up on top of one of

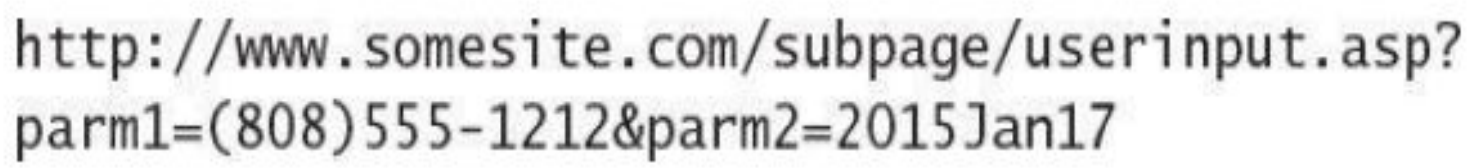
- another piece of your data
- an instruction of yours
- data or code belonging to another program
- data or code belonging to the operating system

Overflow Countermeasures

- Check lengths before writing
- Confirm that array subscripts are within limits
- Double-check boundary condition code to catch possible off-by-one errors
- Monitor input and accept only as many characters as can be handled
- Use string utilities that transfer only a bounded amount of data
- Check procedures that might overrun their space
- Limit programs' privileges, so if a piece of code is overtaken maliciously, the violator does not acquire elevated system privileges as part of the compromise.

Incomplete Mediation

- Verifying that the subject is authorized to perform the operation on an object is called mediation



```
http://www.somesite.com/subpage/userinput.asp?
parm1=(808)555-1212&parm2=2015Jan17
```

- The parameters parm1 and parm2 look like a telephone number and a date
- What would happen if parm2 were submitted as 1800Jan01? Or 1800Feb30? Or 2048Min32? Or 1Aardvark2Many?
- One possibility is that the system would fail catastrophically
- Another possibility is that the receiving program would continue to execute but would generate a very wrong result

Input Validation

- Client Side Validation
  - the program can restrict choices to valid ones only
  - search for and screen out errors
- However, attackers are free to modify the GET or POST parameters
- Solution: Complete Mediation
- Time-of-Check to Time-of-Use
  - modern processors and operating systems usually change the order in which instructions and procedures are executed
  - Instructions that appear to be adjacent may not actually be executed immediately after each other, either because of intentionally changed order or because of the effects of other processes in concurrent execution
  - It exploits the delay between the two actions: check and use, i.e. between the time the access was checked and the time the result of the check was used, a change occurred, invalidating the result of the check
  - The access-checking software must own the request data until the requested action is complete.
  - Another protection technique is to ensure serial integrity, that is, to allow no interruption (loss of control) during the validation

Integer Overflow

- An integer overflow is a peculiar type of overflow, in that its outcome is somewhat different from that of the other types of overflows.
- An integer overflow occurs because a storage location is of fixed, finite size and therefore can contain only integers up to a certain limit.
- The overflow depends on whether the data values are signed
- When a computation causes a value to exceed any limit, the extra data does not spill over to affect adjacent data items
- Either a hardware program exception or fault condition is signaled, which causes transfer to an error handling routine, or the excess digits on the most significant end of the data item are lost.

Race Conditions

- Situation in which program behavior depends on the order in which two procedures execute
- Suppose two processes or threads are using a common shared variable X=5
  - P1 is trying to do X++
  - P2 is trying to do X--
  - After both P1 and P2 execute once, what will be the output?

Race Conditions

- Situation in which program behavior depends on the order in which two procedures execute
- Suppose two processes or threads are using a common shared variable X=5
    - P1 is trying to do X++
    - P2 is trying to do X--
    - After both P1 and P2 execute once, what will be the output?
    - Logically, the output should be X=5
- However, incorrect programming practices could result in the value of X being 4, 5 or 6!!!

Race Conditions

P1

```
register1 = X
register1 = register1 + 1
X= register1
```

P2

```
register2 = X
register2 = register2 - 1
X= register2
```

Race Conditions

P1:	register1 = X	{register1 = 5}
P1:	register1 = register1 + 1	{register1 = 6}
P2:	register2 = X	{register2 = 5}
P2:	register2 = register2 − 1	{register2 = 4}
P1:	X = register1	{X = 6}
P1:	X = register2	{X = 4}

Final answer becomes X=4 (incorrect)

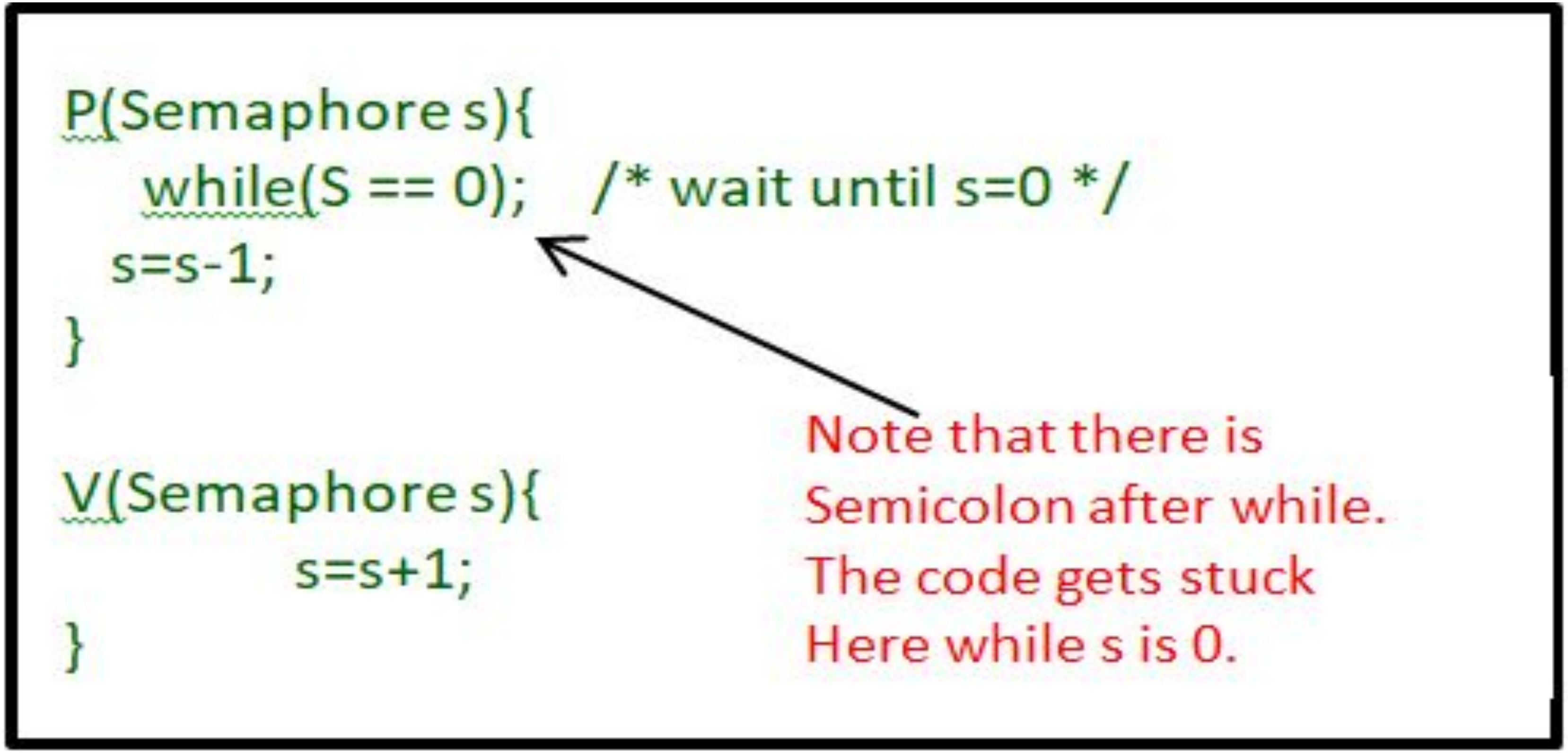
Starbucks Gift Card Hacked using Race Conditions

- Egor Homakov of the Sakurity security consultancy found a race condition in the section of the Starbucks website responsible for checking balances and transferring money to gift cards.
- To test if an exploit would work in the real world, the researcher bought three $5 cards.
- After a fair amount of experimentation, he managed to transfer the $5 balance from card A to card B twice.
- As a result, Homakov now had a total balance of $20, a net—and fraudulent—gain of $5.
- Starbucks later issued a statement claiming that the issue had been fixed

Fixing Race Conditions

- Race conditions occur when multiple processes attempt to modify the same shared data
- The solution is to make sure that only one process can access the variable at a time
- The portion of code which modifies shared variables is called critical section
- Only a single process should be able to access their critical section at a time
- This is done using the concept of semaphores or monitors
- Semaphore is an integer variable with 2 operations defined on it
    - P operation is also called wait, sleep, or down operation
    - V operation is also called signal, wake-up, or up operation.
    - Both operations are atomic and semaphore(s) is always initialized to one.

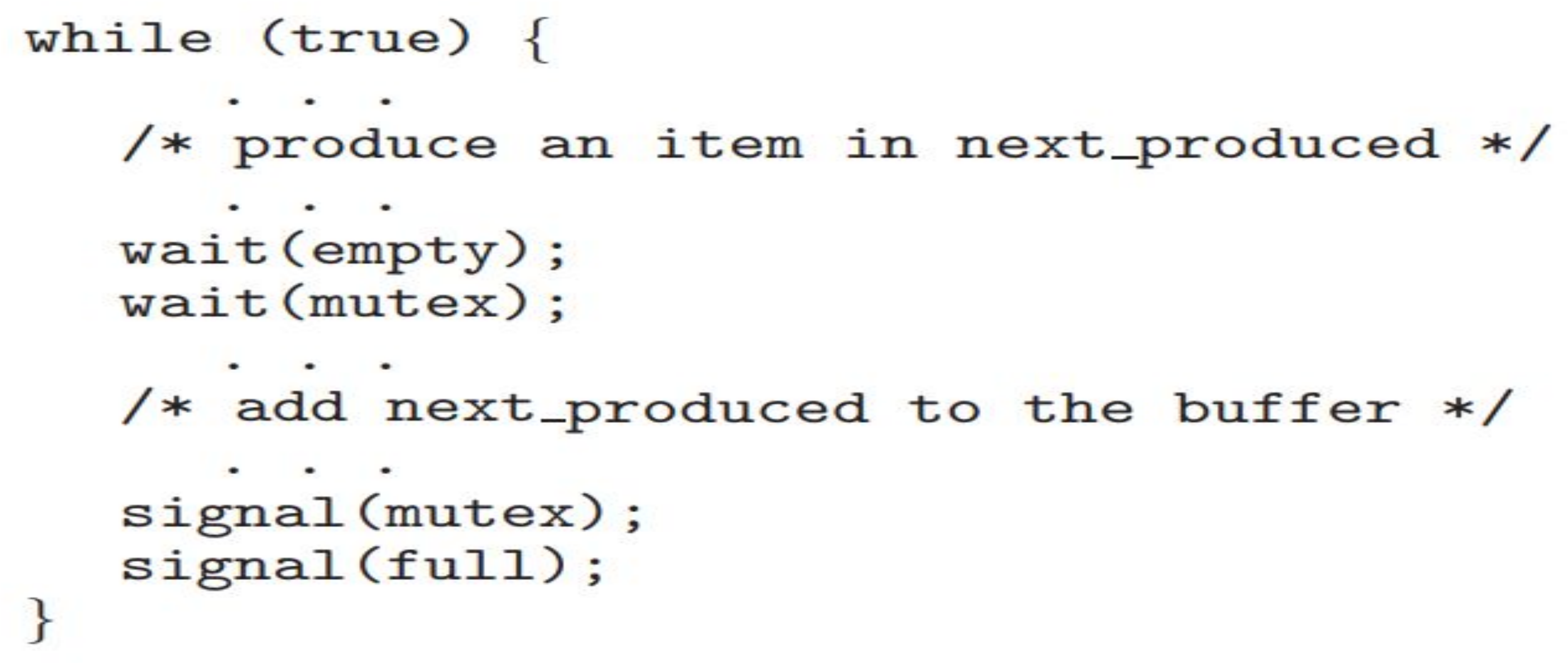
Fixing Race Conditions



```
P(Semaphore s){
    while(S == 0);   /* wait until s=0 */
   s=s-1;
}

V(Semaphore s){
            s=s+1;
}
```

Note that there is
Semicolon after while.
The code gets stuck
Here while s is 0.

Fixing Race Conditions



```
while (true) {
        . . .
    /* produce an item in next_produced */
        . . .
    wait(empty);
    wait(mutex);
        . . .
    /* add next_produced to the buffer */
        . . .
    signal(mutex);
    signal(full);
}
```

Producer Code

Fixing Race Conditions

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

Consumer Code