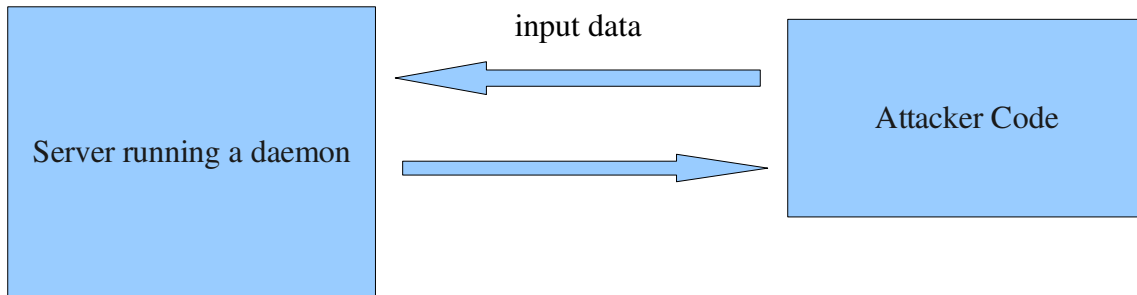


CSE 509: Computer Security

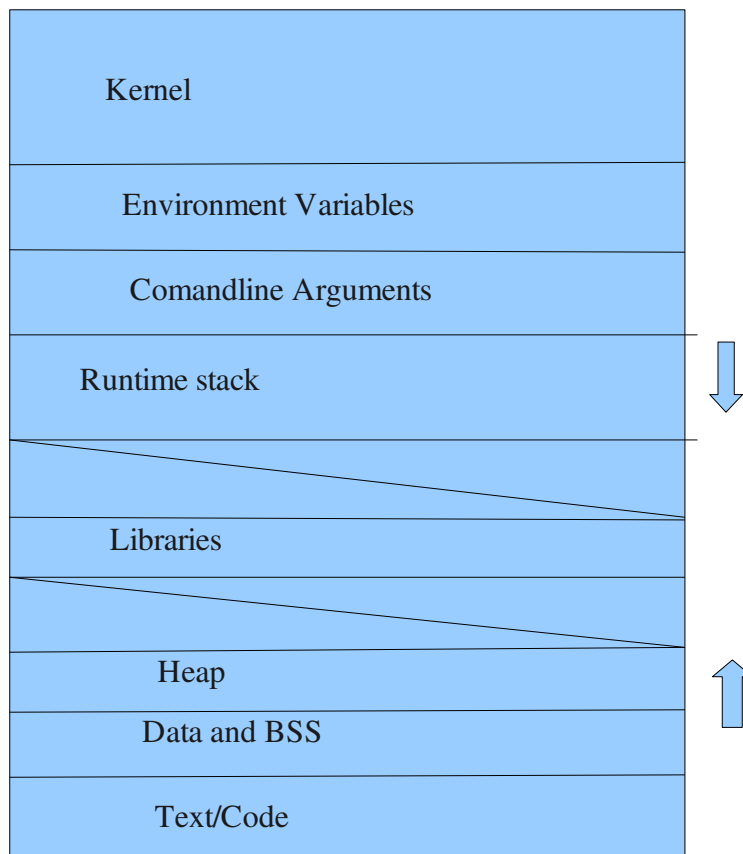
Date: 2.16.2009

BUFFER OVERFLOWS:



The attacker sends data to the daemon process running at the server side and could thus trigger a buffer overflow which could cause the server to execute malicious code.

VIRTUAL MEMORY ORGANIZATION: (Linux, 32 bit)

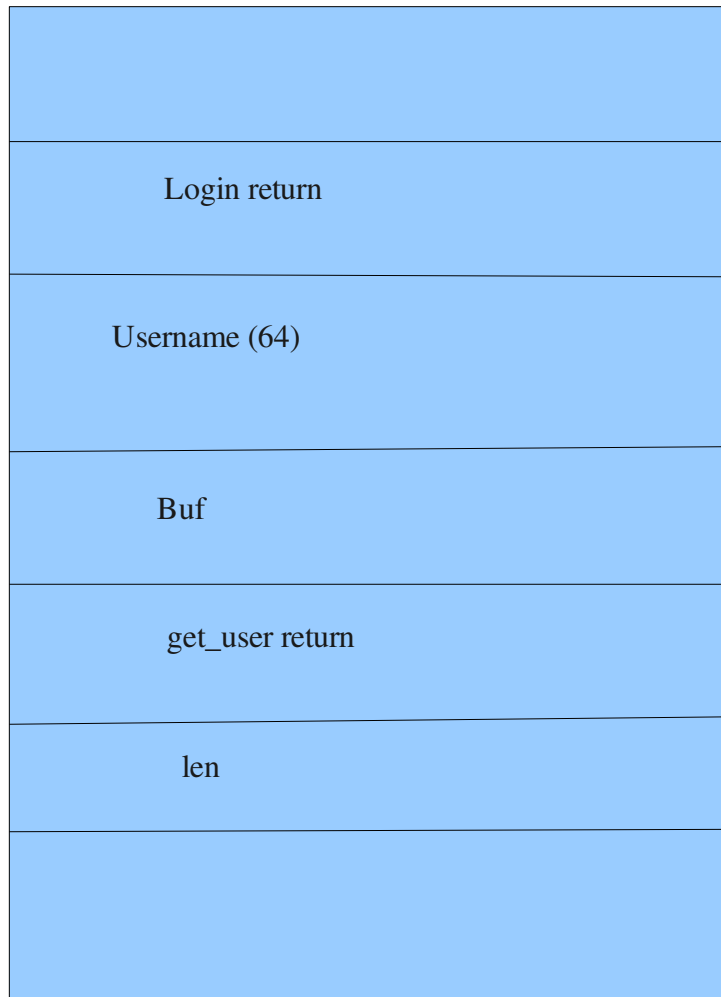


Sample Code:

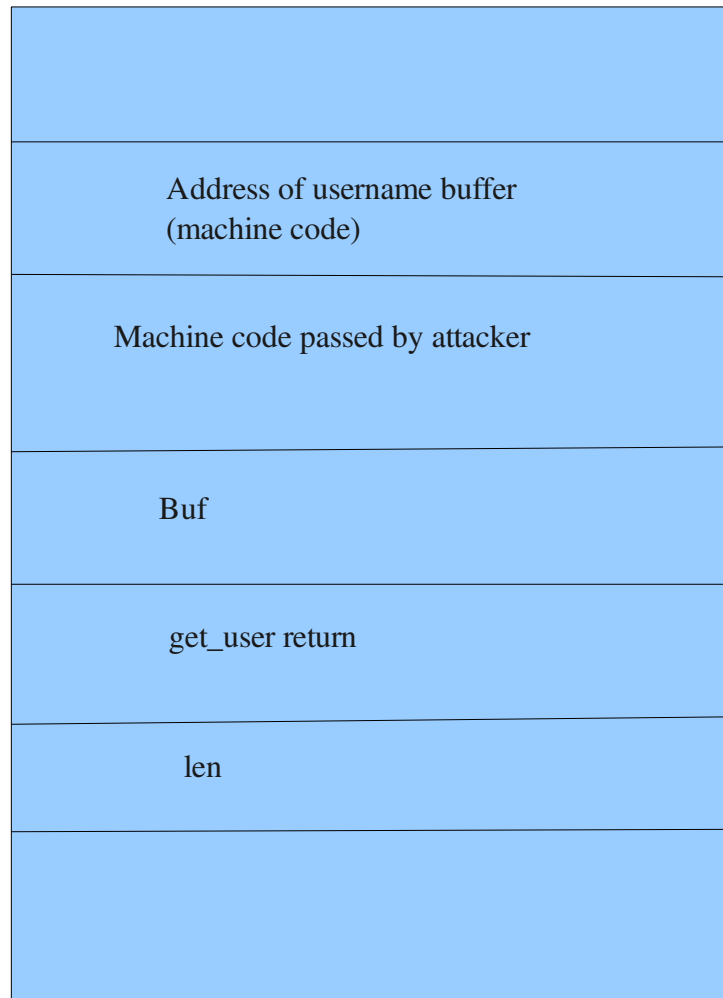
```
void get_user (char *buf) {  
    int len;  
    printf ("Username: ");  
    scanf ("%s", buf);  
    *  
    *  
}  
  
void login (void) {  
    char username[64];  
    get_user (username);  
    *  
    *  
}
```

Assumption: Attacker knows everything about the server.

Stack Layout (Activation Record):



To attack the system, the attacker will pass: <machine code (64 bytes)> <address of username buf>
This will overwrite the login return address with the address of username buffer (which contains the machine code the attacker wants to execute)



- If the user is uncertain of the address of the username buffer (machine code), he can increase the address to be stored at the return address so that it is in the middle of the buffer.
- If he is even more uncertain, he can create a launching pad by creating NOP to the start of the buffer. After NOPs, we place our malicious code, then the address of this code.
- More uncertainty, use brute force.

Trampoline:

A clever solution to overcome this uncertainty is the following. When the username buffer is passed as a parameter to `get_user()`, the address of this buffer is stored in some register. Typically, when a function is to be called, all the registers are pushed on the stack, then the code pointer points the function to be called. Before returning from the function, the function does not clear the registers, instead the caller will pop the registers from the stack and restore the original values.

Assume the value of the buffer is stored in some register `R0`. On return from the `get_user()` function, the address of the `username[]` buffer is in register `R0`. If after the return to login form `get_user`, there is a call of the form `jmp %R0`, then the attacker can use this instruction to jump to the attacker provided malicious code in the username buffer.

For this, he will have to pass

<machine code> <addr of jmp> to `get_user()`

```
*
*
0x1234    pusha
0x1235    call get_user
0x123B    popa
*
*
0x2222    jmp %R0
```

pass <64 byte machine code> <0x2222> to `get_user()` function.

Return to libc attack:

- inject data
- configure returns to pre-existing code in libc

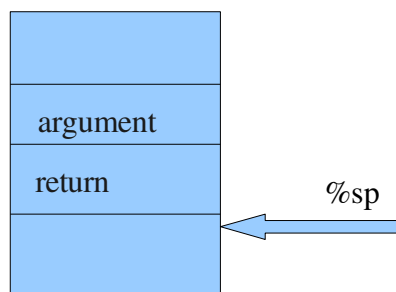
System (`const char * cmd`):

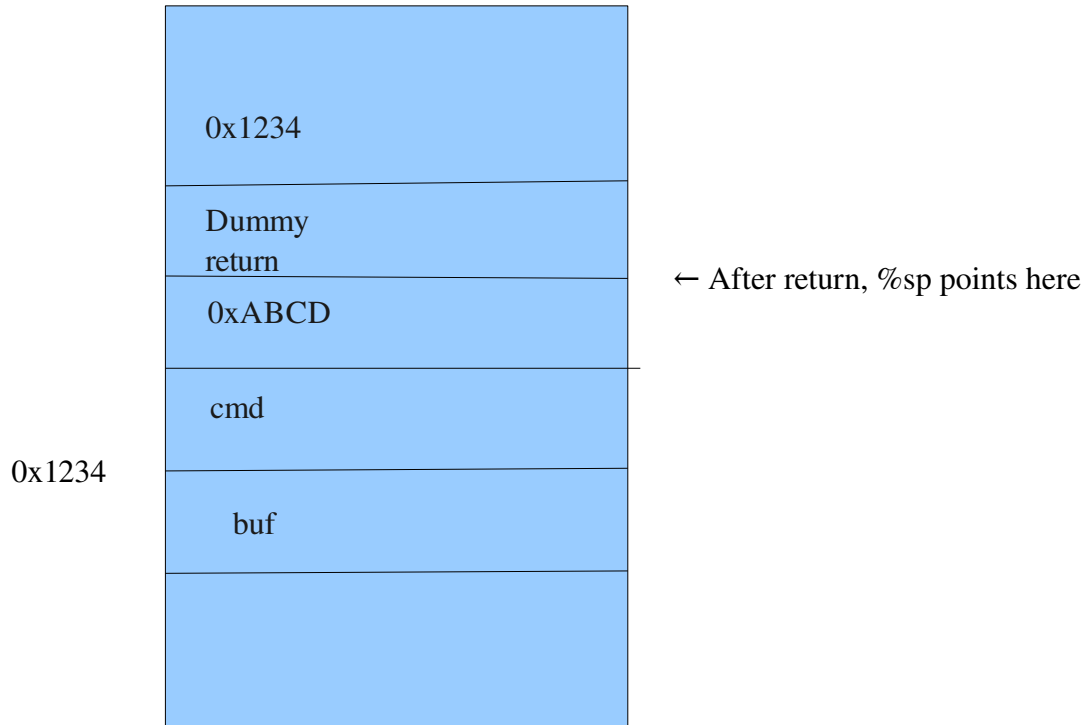
`system ()` executes a shell command specified in `command` by calling `/bin/sh -c cmd` and returns after `cmd` is executed.

Disassembly of `system()` would look like:

```
*
*
0xABCD    mov %R0,[%sp + 4]
           int 0x80
```

So, for `system()`, the stack looks like





So we have to make the stack layout at the call to `get_user()` look like that for the `system ()` call. So, the attacker stores `<cmd> <0xABCD> <dummy return><0x1234>`

Other Code pointers:

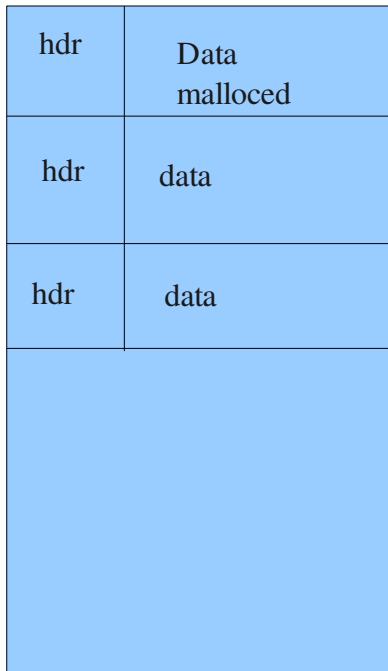
- function pointers
 - callbacks (local variables)
 - structures
 - V table/ V pointer (C++)
 - function list (atexit)
 - exception handlers
- * combine with return to libc

Arbitrary memory Write:

```
void fun () {
    char buf[64];
    int val;
    int *ptr;
    <BUFFER OVERFLOW OF BUF>
    *ptr = val;
}
```

can be user to transfer a lot of data (4 bytes at a time)

Heap Overflow:



The memory blocks allocated via malloc/new contain some metadata along with each memory chunk allocated which contains information like size, flags, etc of the allocated memory.

The memory chunks are stored in the form of a linked list, thus typical DS for a memory chunk would look like

```
struct malloc_chunk {  
    *  
    *  
    struct malloc_chunk *next;  
    struct malloc_chunk *prev;  
};
```

When freeing a chunk of memory, it is removed from the allocated list. The code for the same will look like:

```
hdr->next->prev = hdr->prev;
```

which is $*(\text{ptr} + \text{offset}(\text{prev})) = \text{val}$;