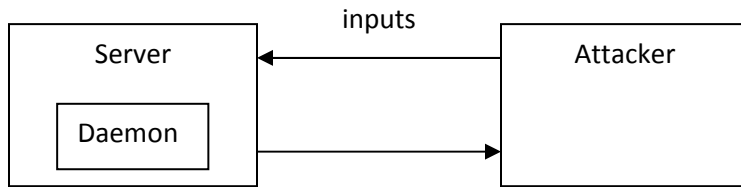


Buffer Overflow

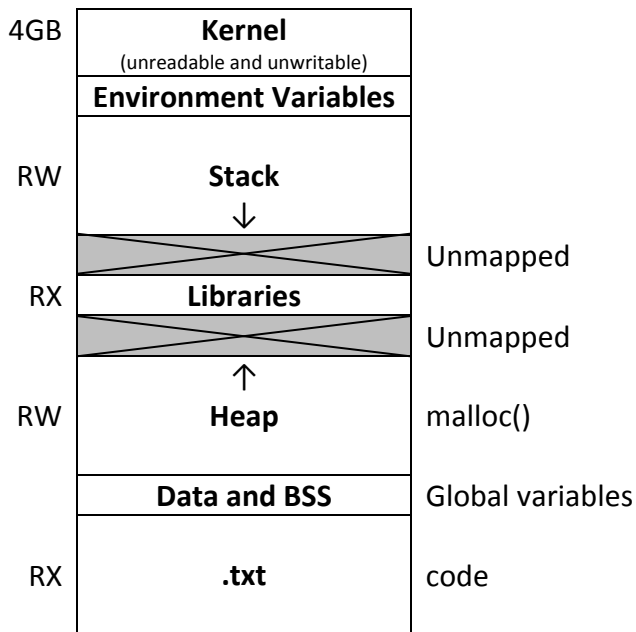
Scenario:



- A server is interacting with an attacker, example: login to the server
- Attacker want to execute some code on the server
- Assume attacker know everything about the server and even the runtime stack
- The attack: provides an crafted input to trick server in executing arbitrary code

Virtual Memory Organization

(Linux System, 32 bits)

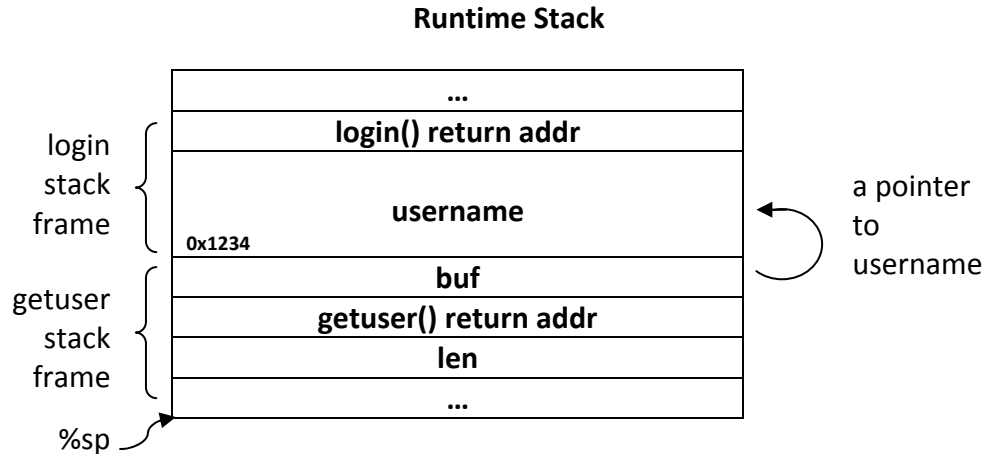


Sample Login Program

```

void getuser(char* buf)
{
    int len;
    printf("username");
    scanf("%s", buf);
    ...
}

void login(void)
{
    char username[64];
    getuser(username);
    ...
}
    
```



Attacks

1. Overwritten username and login return address
username: `<machine code (64 bytes)><0x1234>`

The username provided will fill the 64 bytes of allocated space for the username field on the stack with attacker's code. It then overflows the buffer and overwrites the `login()` return address with `<0x1234>`. If the code do not crashes, the code returns and jumps to address `<0x1234>` which is the beginning of attacker's code.

2. Landing padding (if attacker have certain degree of uncertainty)
username: `<nops><machine code (60 bytes)><0x1234>`

The attacker may have certain degree of uncertainties where the username is stored on the stack; attacker can increase the chance of successful attack by adding nops. If the code returns and land on a nops, the code will execute the nops and go to the next instruction.

3. Trampoline
 - Suppose `%r0 = &username` when `login` returns.
 - Suppose `.txt` segment contains `jmp %r0` at `(0x0122)`
 - username: `<machine code (64 bytes)><0x0122>`

It is very likely to have a degree of uncertainty on the address of the username. However, if there is a register `%r0` that holds `&username` when `login` returns and there is a code in `.txt` segment contains `jmp %r0` at `(0x0122)`, we can overwrites the `login()` return address with `(0x0122)`.

Return-to-libc Attack

- Attacker do not inject code, but configures returns to some pre-existing code
- Attacker injects data

Sample *system* function code:

```
system(char* args)
{
    exec(arg);
}
```

System:

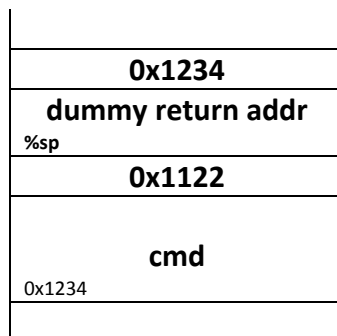
```
0x1122:    mov %r0, [%sp+4]    # move first argument
           int 0x80
```

To launch this attack, attacker wants to have a stack that looks like:



The main idea behind this attack is the mimic a function call to `system()` by supplying crafting a username that will turn the stack to the one that's above. The return address is the address of the instruction to be call and the args are the supplied arguments to the function call.

By supplying username: `<cmd><0x1122><dummy return addr><0x1234>`, we get a stack that looks like this:



Other Code Pointers

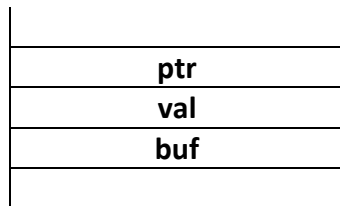
Function Pointers

- callbacks (local variables)
- structures
- vtable/vpoints
- fnlist (atexit)
- exception handlers
- combine with return-to-libc

Arbitrary Memory Write

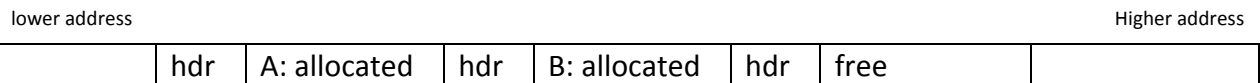
```
void function(...)
{
    char buf[64];
    int val;
    int* ptr
    <buffer overflow of buf>
    *ptr = val;
}
```

Only works if stack is organized like the below:



Heap Overflows

- allocated memory from malloc



```
hdr
{
    next;
    prev;
}
```

free: `hdr->next->prev = hdr->prev`

`*(ptr+offset(prev)) = val`

If the first hdr is vulnerable to overflow attack, attacker can modify the hdr and when a free() is called, attacker can control the resulting heap.