User-Input Vulnerabilities

Buffer Overflows

Buffer overflow vulnerabilities are one of the most widespread and dangerous types of software vulnerabilities. The implications of them range from being able to view the memory of a program all the way to complete takeover of a machine.

What exactly is a buffer overflow? In computer science, a *buffer* is an area of memory, usually an array or a string. An *overflow* is where you try to store more stuff in the buffer than it has room for. That extra stuff can spill over into adjacent memory locations, potentially overwriting other variables, or worse.

The C and C++ languages are particularly vulnerable to buffer overflows. Other languages, like Python and Java are not so vulnerable. In Python, if you declare a list `L = [0,1,2]` and try to write to `L[10]`, you will get an index out of bounds exception, and nothing will be written. But C and C++ will gladly try to write to that index with no warnings or exceptions at all. Here is an example of some code that suffers from a buffer overflow.

```
#include <stdio.h>
#include <string.h>

int main() {
    char buffer1[10];
    char buffer2[10];
    strcpy(buffer1, "123456789");
    strcpy(buffer2, "abcdefghijklmnopqrst");
    printf("buffer1: %s\n", buffer1);
    printf("buffer2: %s\n", buffer2);
    return 0;
}
```

We have allocated space for 10 characters in `buffer2`, but we try to store 20 characters in it. Some of those characters will overflow into the memory location for `buffer1`, and instead of being `"123456789"`, it ends up being `"qrst"` The buffers are stored on a stack, which why `buffer2` overflows into `buffer1` and not the other way around.

Programs use a stack to store information needed for function calls. Each function gets a chunk of it called a *stack frame*. That stack frame is where it stores all its local variables and some other things, the most important of which is the *return address*. This indicates where in the program's code to return to when the function is done. Buffer overflows usually try to overflow a buffer far enough to overwrite the return address with the address of some code the attacker wants to run. That code could be another function in the program, it could be a library function, or it could be code the attacker has placed in the buffer. Here is an example a program being exploited by a buffer overflow.

```
./vulnerable_program $(python -c "print('\x90'*40
 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
 \x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'
  + 'A'*40
  + '\x20\xd3\xff\xff')")
```

The person doing this is using Python to send some data to the program. Everything in the print statement is what is being put into the buffer. Most of it, all the `\x##` stuff, is raw machine language instructions. The part starting with `\x31` on the second line is *shellcode*, whose purpose is to open up a command shell. This will give the attacker access to the system to run commands. These commands will run with the same level of permission as the vulnerable program itself. So if it's running as root, then the attacker will have root privileges on the system and can basically do anything. Some descriptions of vulnerabilities refer to this "arbitrary code execution". After the shellcode, the program puts in a bunch of A's. The purpose of this is to overflow the buffer to right where the return address is. The last part, on the last line, is the address of the buffer itself in memory. This points the return address to the buffer so that the shellcode the attacker put into memory will be run.

Here is another example. This is what the Code Red worm would send to random IP addresses. It's an HTTP request with the name of the resource being requested designed to cause a buffer overflow. We see a certain number of N's being used to overflow the buffer right to where the return address is. What follows that is the raw machine code to be run.

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801
%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3
%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a  HTTP/1.0
```

Buffer overflows are one the most common and most serious vulnerabilities in modern software. A lot of system software is written in C and C++ because it is fast and gives you access to the system itself in a way that high-level languages like Java and Python can't. But in C and C++, it's very easy to accidentally create an opportunity for buffer overflows.

The eval function

Python has a useful function called `eval`. It takes a string containing a Python expression, and evaluates it. For instance, `eval(2+2)` will return 4. As another example, if we have a variable `x` in a program, and we import the math library, then `eval(math.sqrt(x))` will print the square root of `x`.

It's sometimes handy to use `eval` in conjunction with an input statement, since the `input` function will return a string, and we often want to convert that string into a number that we can work with. Here is an example of how we should convert the input if the user is entering an integer.

```python
x = int(input('Enter a number:'))
print('The square of that number is', x**2)
```

We could use the `float` function if the user is entering a floating point number. However, it's tempting to use the `eval` function instead, since it will work just as well with integers and floats, and it even allows to use to put in math expressions. However, we have to be careful. Say we have the following:

```python
password = 'Home25!'
x = eval(input('Enter a number: '))
print('The square of that number is', x**2)
```

The user could enter `print(password)`. This will call the print function, which will print the password. It will also crash the program since the print function doesn't return a value, and when we try to square `x`, there will be an error. We could fix it with by entering `3 if print(password) else 3`. This will still call the print statement, and the whole expression will end up evaluating to 3, which will work with the `x**2` in the line below.

We can even take things so far as to change the value of the password. The trick is that Python has a function called `globals()` that returns a dictionary of all the global variables in the program. (There is also a function called `locals()` for a function's local variables.) Python dictionaries have a method called `update` that allows us to change the dictionary. Putting this together gives us the following input:

```
3 if globals().update({'password':'12345'}) else 3
```

One solution to this problem is not to use the `eval` function in any scenarios where you wouldn't want people reading or changing things in the program. If you must use `eval`, then you would need to do some careful sanitization of the input. In the example above, one approach would be to only allow digits, the decimal point, and maybe a few math symbols like -, +, *, and /.

A remote code execution vulnerability

Sometimes websites will take use input and use it in data fed to something called from the command line on the server. For instance, there are some sites online that will allow you to enter a host name, and then the site will ping that host and give you back the results. Here is some PHP code that will do just that, assuming the input comes from a POST request.

```
if (isset($_POST['host'])) {
    $output=null;
    $retval=null;
    exec('ping -n 1  ' . $_POST['host'], $output, $retval);
    foreach ($output as $x)
        echo $x . '<br>';
```

The serious weakness here is that the user's input is pasted directly into the `exec` statement without any sanitization. If it's a Linux server, an attacker could enter something like `google.com; ls`. The `;` character at the Linux command line is used to string together multiple commands on the same line. The end result is that `ping -n 1 google.com; ls` will be run at the command line. The ping command will be followed by a command to list out the files in the directory. Of course, `ls` could be replaced by something much more interesting that could delete files or set up a reverse shell. In other words, the attacker can run commands or even code of their own creation remotely on the server. This is a serious problem.

The attack would work similarly for a Windows server. The equivalent input there would be `google.com & dir`. Windows uses a different character to join together commands. The solution to both attacks is to carefully sanitize the input data to make sure that multiple commands could not be run.