CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 3rd, 2011

More examples of vulnerable code

Format string attack

We are going to demo a format string attack that does a return-to-libc attack. We will use the classic `printf` format string vulnerability to specify values to write and the locations to write them to.

Remember: Values are specified by increasing `snprintf`'s count, while addresses are specified as bytes in the format string.

Our **goal** is to overwrite three words:

- The return address of `log_user`
- The return address of the `system` function
- The pointer to the argument of the `system` function

This is what our **stack** will look like after the attack:

X + high	**New stack**
	Argument for `system` (pointer to `system`'s command string)
	Return address for `system`
	Return address for `log_user`
	Saved base pointer `ebp`
	End of buffer `buf`
	.... (remaining `buf`)
	Third address to overwrite
	Second address to overwrite
	First address to overwrite
	Some junk (to increase the `snprintf` counter)
	ARGP points somewhere above and will be increased
X - low	

Format string: We specify a series of (junk, address) pairs in the beginning of the `snprintf` format string.

- We use %90x to skip some extra junk and point to buf
- We use %150x to skip the first junk in the buff and get ARGP to point to the first address.
- The counter was just incremented to the desired value and will be written to the first address using %n.

Integer overflows & runtime integer checking

Integer overflows are pretty common and they are usually **a build up to a memory error**.

Two's complement refresher

Take 4-bit integers as an example.

If you're dealing with unsigned numbers then 0000 is 0 and 1111 is 15. You can add numbers together like 1101 and 0101, however sometimes overflows occur. 4-bit numbers can only store values up to 15, but someone might be adding 10 to 13 and get 26, which is not representable as 4-bit number. You would need 5 bits.

Example: 1101 + 0101 = [1]0010. An overflow just occurred, since we only had 4 bits but the result needs 5 bits to be represented.

How can we represent negative numbers? Using two's complement:

Binary representation	Decimal value
Negative numbers (-2^n minimum value)	
1111	-1
…	
1000	-8
Positive numbers ($2^n - 1$ maximum value)	
0111	7
…	
0000	0

Problems:

- if you add two large positive numbers, you'll get an overflow and the resulting number will be negative
- if you add two large negative numbers, you'll get an underflow and the resulting number will be positive

Exploit: Integer overflows can be used to `malloc` 0 bytes and then copy a huge amount of data into memory.

Conside the following code:

```
void getComm(unsigned int len, char * src)
{
    unsigned int size;
    size = len - 2;
    char * comm = (char *) malloc(size + 1);
    memcypy(comm, src, size);
    return;
}
```

If you let $size = 2^{32} - 1$ then 0 bytes will be allocated (size + 1 will overflow and equal 0) and then you can overwrite 4GB worth of data.

Fix: Modify the compiler to check for underflow and overflow exceptions:

- Truncation check (make sure the higher bytes are all 0 when a variable is truncated).
- Sign check when casting signed to unsigned (ensure that the values have the same sign)

Double frees

Sometimes programmers have to handle error conditions and they screw up. The most common mistake looks like this:

```
p  = malloc(sizeof(*p));

if(something_bad) {
    free(p);
    goto fail;
```

```
}

fail:
    free(p);
```

The programmer will have freed `p` twice, if `something_bad` happened. It turns out this mistake can be exploited.

Heap memory is divided into chunks. Somewhere in memory the allocator has a linked list of the free chunks of memory. Initially our `malloc`'d `p` might point to chunk #3.

#4	Chunk
#3	Chunk reserved for `p`
#2	Chunk
#1	Chunk

When you free `p`, a node will be added to the **free list** which points to that free chunk of memory. Note that after `free(p)` is called, `p` will still point to chunk 3, so freeing it again will re-add the node to the free list. So the free list will have two free slots which point to the same chunk in memory.

What can go wrong? Let's take a `struct` example:

```
struct user {
    char * name;    //  programer is careful to put the name on the stack
                    //  so that it does not overflow into is_admin
    int is_admin;
};

void loginuser(char * name)
{
    struct user * u = malloc(sizeof(user));

    u->name = malloc(256);
    u->is_admin = is_administrator(name);

    strcpy(u->name, name);
}
```

Suppose the double free executes before `loginuser` gets to execute. So now the program is in a state where it has 2 free slots pointing to the same memory chunk.

`malloc` will allocate the user `u` to point to that chunk, and then `malloc` will allocate `u->name` to point to that same chunk.

After this, the attacker chooses a careful username to give to `loginuser` such that its 2nd word has a non-zero value. When `strcpy` copies the name into `u->name` it will overwrite the `struct`, since `u->name` points to the same location the `struct` was allocated in.