Security

Alex S.*

1 Introduction

Security is one of the most important topics in the IT field. Without some degree of security, we wouldn't have the Internet, e-Commerce, ATM machines, emails, etc. A lot of the seemingly unrelated topics base their existence on security.

There are two primary forms of security that system designers have to care about, *authentication*, and *authorization*.

1.1 Authentication

Authentication refers to the idea of ensuring you are who you say you are. This usually takes the form of username/passwords. If you know the password, you are authenticated, and the system trusts you. If you don't know the password, the system doesn't trust you. Simple as that.

Authentication can happen in many different ways; not necessarily via passwords. It usually boils down to three basic principles:

- **Something You Know:** This would be the password. It could also be something more obscure, like mother's maiden name, or your social security number, etc.
- **Something You Have:** This is usually some sort of a token. This can take the form of a bank card, a credit card, or for more secure situations a cryptographic token (i.e.: a keychain like thing that generates 'random' keys; which you combine with a password to login).
- **Something You Are:** This takes the form of fingerprints, iris scans, DNA fingerprinting, weight, height, pictures, etc.

By itself, these methods can be easily defeated (you can guess/intercept/buy someone's password, take away their tokens/keys, or somehow cripple the person not to have fingerprints). When combined in several combinations, they present quite robust security. i.e.: A credit card with a picture prevents someone else from making in-store purchases. It is also pointless to steal cryptographic key chains, since you don't know the password, etc.

1.2 Authorization

Now that the system knows who you are, another issue in security comes into play... What are you as a user of the system allowed to do?

This concept usually boils down to what privileges you have in using the system. Can you update certain records?, etc.

This idea is usually implemented via 'user roles'. When someone logs into the system, their account contains a list of roles that the user may take. If the user's role includes being a 'user administrator', then the user is allowed to create/remove/update users, and so on and so on. Each 'role' may have various privileges associated with it. So for example, you might discover that your 'account manager' person also needs '`access_old_records`' permission, then you add that permission to that role, and anyone who is of that role will automatically get that permission. This can also work both ways; if you're a 'developer' role, you have access to certain resources. But if at the same time you're of the 'consultant' role, some privileges might be revoked (even if they were granted by the 'developer' role).

2 Web Site Security

Most websites concentrate their security handling on authentication, and less so on authorization. Most users of the website are un-trusted, so there is no point in distinguishing among several hundred thousands of equally un-trusted users.

Here's how the security works: The client (web-browser) connects to the server. The server generates a unique key, also knows as 'session id'. This session is saved on the server (somewhere—usually in memory, but sometimes in simple databases). The key is usually returned to the client as a cookie (or something equivalent of a cookie).

[Note: While most websites today maintain the session using cookies, you can also maintain session without cookies—by using URL rewriting. Basically the 'session id' becomes part of the URL, and every single page the server returns is modified to have every URL maintain that 'session id'. As you can imagine, it is quite cumbersome.][1]

The client is responsible for presenting the cookie to the server every time it reconnects. If it doesn't, the server simply issues a new cookie (starting a new session).

Login happens simply: The client sends the server the username/password pair. The server looks these up in a database, and if found, puts a little checkmark next to the 'session id' in the database. From now on, the client with that cookie is trusted as a user of the system.

Logouts happen similarly. The cookie can expire (usually when you close the browser). The session (on the server) can expire (if you don't contact the server within say 30 minutes, the session is erased—and your cookie is no longer recognized as valid; you need to re-login. Or, the user presses the logout button, at which point the session is invalidated, and the cookie is discarded.

[1] An even more cumbersome approach is to modify the session id with every request.

3 Distributed Systems Security

With distributed systems, the idea of security is very similar to the one described for web sites. There is quite a bit more complexity though.

With websites, there is a notion of a unique 'id' for the session. Most distributed systems (if they have to deal with such security) also manage a form of 'session id'.[2]

You connect to a component; you give it a username/password. The component generates the 'id', stores some session information in a database (or memory cache) and returns you the id.

The client is now responsible for presenting this 'id' to the server every time it invokes a method or does any operation.

There are several ways of passing around this token; and some are just as convenient as passing around cookies. In most distributed system architectures, there is usually a notion of a connection channel. All communication is happening via this channel. So instead of logging in and tracking the authorization of the user, you can in effect track the channel. The channel usually has some back-door mechanism, a context of sorts, which you can use to send information to the server that's not part of the actual request. In CORBA this is called piggy-back approach.

Anyway, from the client side, the whole thing boils down to: sending username/ password, and the received 'id' is added to the communication channel context (under some nice name like 'ticket'), and from then on, the client doesn't have to care about security—just invoke methods on the server, and the communication context will ensure that the ticket is sent with every request.

From the server's perspective, after the user logs in, you need to generate the ticket, save it, and send it to the user. On every request that the user performs, you need to extract the ticket, find the user, determine if that user has the privileges to execute that method, and then continue on with the method.

Several things should be mentioned: Often the security/privileges of methods can be setup externally. For example, with EJBs, the container manages security and access to the beans. So you never have to code up security access in actual processing code.

The user 'id' lookup usually happens in a memory database; so handling security doesn't involve an extra database hit; for some systems this may be un-scalable, but even for those, combined with a memory caching scheme, most user based security is not hitting the database every single time.

4 Encryption

Encryption is well beyond the scope of this discussion, but here are a few pointers:

If the connection ever goes through an un-trusted network (wireless, Internet, etc.) and the data shouldn't be seen by anyone, then it should be encrypted. Usually the form that

[2]Except they usually call it a 'ticket' instead of a 'cookie' or 'session id'.

takes is via SSL (Secure Socket Layer).

For websites, (for properly setup websites), that usually means just connecting via `https` instead of `http`[3]. A good rule of thumb is never to send a password to an http connection, but use https.

For distributed systems, that usually means using an SSL interface, or some other custom encryption scheme[4].

In the real world, most data is not encrypted, and nobody seems to care. Most e-mail you send out is unencrypted. In fact, very often people are checking their e-mail via an unsecured wireless connection (like in the park). So while security is a good idea, is it important not to go overboard and make it intrusive enough for people stop using the system in the first place.

4.1 RSA

RSA is the most popular (and 'classic') public key cryptosystem.

If we have two large primes, p and q, we can compute their product n. For this special value of n, we can easily compute the Euler's Totient function as:

$$\phi(n) = (p-1)(q-1)$$

We then pick another integer e, such that $\gcd(e, \phi(n)) = 1$, in other words, e must be relatively prime to $\phi(n)$. The pair (n, e) is now our *public* (or *encryption*) key. Let b, $0 \leq b \leq n-1$, be a block of a message we are encrypting. The encryption function $E(b)$ is just:

$$E(b) = b^e \bmod n$$

For decryption, we need to compute a value d, which is the inverse of e modulo $\phi(n)$, and can be computed by the extended Euclidean algorithm. Once we have d, the decryption function $D(a)$, where a is the encrypted message block, is:

$$D(a) = a^d \bmod n$$

In effect, we have $D(E(b)) = b$. The trick about this scheme is that the speed of encryption and decryption revolves around our ability to do

$$m^x \bmod n$$

relatively quickly. If x happens to be large (have a high 'binary weight'—or many of its bits set to 1), then this step is very slow. Usually in RSA, e is chosen to be relatively small, while d is relatively large; that means encryption is fast, while decryption is slow.

A sort of semi-vulnerability is that you can time the encryption or decryption step, and figure out the binary weight—from which it is 'easier' to figure out what x is. Most current implementations add a random wait period when computing such things—so it's difficult to judge by just timing the response time. Folks have also used the power consumption of a device to get a hint at the binary weight (this works for devices like smart-cards).

[3]Start the URL with `https://...` instead of `http://...`

[4]Note that custom encryption has a very high chance of not working right.

5 Input Validation

A lot of security issues come up in input validation. Sometimes even a valid input causes an undesired operation—like input `/*/*/*/*` would cause an ftp server to use up all the server's resources, and eventually crash the computer (this has been fixed now). Whenever you setup anything, you need to be aware of the importance of validating input.

5.1 Buffer Overlow

Buffer overlow should be mentioned in any discussion dealing with security. It is literally the single most notirious security hole that is causing numerious security issues with lots of software. The core of the issue is bounds checking on data input, which usually results in using methods like `gets(char*)`, or something similar.

If someone sets up a program like:

```
#include <stdio.h>
int main(){
    char name[100];
    printf("what's your name: ");
    gets(name);
    printf("Hello %s!\n",name);
    return 0;
}
```

The program will work just fine for all names less than 100. In fact, it can work for years before anyone notices anything. When someone does notice that the program crashes when you put in more than 100 bytes as input, they could examine the effect (especially the stack) in a debugger. Now, since the memory grows forward, but stack grows backwards (and 'name' variable is stored on the stack), that means that right beyond those 100 bytes, is the return address of the location that called 'main' function (usually it's not really a 'main' function—but in this example it is). In any case, by writing another address over that (by filling in more data into 'name' variable than it can hold) the attacker can send the program execution to anywhere. Usually, the return location is actually somewhere in the 'name' variable (which has been filled with an attacking program).

This may seem complicated, but in reality it's not that hard—all that's needed is time (figure out where/what's causing the problem), a debugger (to examine what's happening; what memory locations are involved), and an assembler (to construct a custom program that will fit into the space just right).

Modern languages like Java and C# (.NET) have somewhat removed that problem (since you're no longer running native code), but the issue may still arise with some library not checking the bounds. There's also the no-execute flag that can now be setup in newer processors; which will prevent the execution of stack space—the only trick is that the OS must be setup to support such a thing, and properly flag stack segments as un-executable.

Also note that the problem isn't just with user input, but with file input, and all sorts of other input (like network packets). Some folks have once found an issue with how Windows verifies image files (it uses a signed instead of an unsigned comparison to ensure if image isn't of improper size), which enabled folks to embed viruses into image files (when someone tries to view the image (say on the website), it causes a buffer overflow, which loads the virus into the computer).

5.2 SQL Injection

Another very common security issue that happens in the database world is SQL Injection. Basically that boils down to not properly handling user input, which somehow then finds its way into the database query. For example, imagine if our database query was something like this:

```
SELECT * FROM USER WHERE USER='$user' AND PASS='$pass'
```

Now, imagine that we didn't properly clean `$user` or `$pass` variables. Imagine that the `$pass` variable has something like:

```
' OR 1=1;
```

Or something along these lines. The resulting query would be something like:

```
SELECT * FROM USER WHERE USER='username' AND PASS='' OR 1=1; '
```

Which might just enable the user to login without even knowing the password. On the other hand, the query could've done something a bit more malicious.

The example above used to work on a great number of sites a long while ago, when people were just getting used to the net, and Perl CGI scripts. Current attacks are usually a bit more advanced, but they still take the same basic form.

One way to protect against this is to filter out improper characters (escape characters, quotes, etc.) out of user input. This is not always as easy as it may sound, because you're dealing with the web-escape characters, your language escape characters, and database escape characters. Also, some of them may allow inputs in octal notation, and some seemingly 'ok' characters can function as escape codes for escape codes for the database query. A good strategy is to limit the input to something concervative like alphanumeric strings (and nothing else; like special characters).

Another way to deal with the issue is to pass everything through a stored procedure. Stored procedures take arguments, and the data stays as those arguments—it's not possibly to modify the SQL query that's inside a stored procedure. Care must be taken in ensuring the parameters are correct, etc., but that's usually simpler than ensuring the query is perfectly fine when it comes to all the possible escape codes.

6 BugTraq

There's an online mailing list that tracks bugs, security issues, etc., in many common programs. If you're serious about security, you'll subscribe to that list (it's free—it does tend to generate a lot of traffic though). There are many such lists, the primary one is BugTraq, but there's also NTBugTraq, etc., so subscribe to the ones you consider important.

This list is sometimes used for noble purposes (of notifying everyone of the bug) and sometimes for evil purposes, such as finding a bug in a particular software package, and some tips (or code) on how to exploit that bug. So it's worth to always be in touch with these things—if that's what you do.

7 Network Scanning

There are various network scanning programs, like `nmap`, `Ethereal`, etc. Basically you should know what these software do, and why someone would use them. Using nmap for example, someone can find out what computers are on a particular network, what ports they have open, what software they have running, what version of the software is running, etc., (at which point a malicious user can see out that program/version on BugTraq to see if there are any problems with it). Ethereal (and other similar programs—in fact, it's pretty trivial to write a shell or Perl script to do that on a Linux box) can be used to capture all network traffic (and then do a search on particular things). For example, if someone's browsing the web in Bryant Park (via wireless), someone can capture all that traffic and get any un-encrypted information that's flying through the air.

8 The Human Factor

Most security issues aren't technological—they're mostly the result of human beings. Things like using weak passwords (or never changing passwords). Or being totally gullible when it comes to IT. For example, an average secretary might get a call from someone who claims to be from "IT", who might ask the secretary to follow some steps on her computer to do something. Most people won't question the person calling them from the "IT Department". Also issues like opening e-mails attachments (it's a great thing folks are becoming better at that one though).