# 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.:

[*]© 2004-2008, Prof.Phreak

A credit card with a picture prevents someone else from making in-store purchases (or at least that's the theory). 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, file, or in a 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.

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

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.

## 2.1 Website Authorization

For authorization, it is highly recommended you follow the contrller pattern along with some users/roles/permissions scheme.

The 'controller' is a main page that gets hit on every request. For example, you can have: `get.php` get all requests that hit your website. The URL will also have some 'action' parameter, ie:

```
http://.../get.php?action=some/path/to/script.html
```

Only `get.php` is publically accessible (along with maybe image files). Everything else needs to go through `get.php`. The `get.php` script will know what the action is. It will also (via lookie) know what user is logged in.

How do we go about finding out whether that user has the rights to access that action? We can start by defining these five tables:

```
user(userid,username,password)
role(roleid,rolename)
permission(permissionid,permissionname)
userrole(userid,roleid)
rolepermission(roleid,permissionid)
```

Once all these tables are properly linked up (filled with data), we can find out whether username 'bill' can access the page 'some/path/to/script.html', via:

```
select *
from user natural inner join
    userrole natural inner join
    role natural inner join
    rolepermission natural inner join
    permission
where
    username='bill' and
    permissionname='some/path/to/script.html'
```

If we get a record back, we allow the user in. If we don't get a record back, we forward the user to some 'access denied' page.

Note that we would need to do this type of query for every single request the user makes—to properly ensure that the user is only allowed to do the things that they're allowed through their role.

Note that the above query can be 'optimized' by using userid, and permissionid, etc, and involving less tables. (also note that 'permission' table may contain the actual filepath to the resource, instead of the 'dummy' path that the world sees in the URL.

Note about the controller pattern: You can make URLs look 'natural' by using a few apache rewriting rules, ie, put the below into a `.htaccess` file in our website's htdocs (or `~/public_html` folder):

```
RewriteEngine on
RewriteRule   ^(.+\.html)$ get.php?action=$1&%{QUERY_STRING}
```

All of a sudden, URLs of the form:

```
http://some.server.com/some/path/to/script.html
```

Will be turned into:

```
http://some.server.com/get.php?action=some/path/to/script.html
```

ie: for most users, it's more natural to deal with URLs that seemingly point to plain html files.

## 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 most of confusion arises from how exactly this token gets sent around); 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.

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

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 some particular 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 Passwords in Database

Depending on the trustworthyness of your environment (public web hosting, etc.) it may be a good idea to never store passwords in clear text, not even in the database. The way to do that is to generate some cryptographically secure hash out of the password, and store that. For example, if someone sets their password to '12345', you would actually store:

```
insert into user(username,password,createdate)
    values('bob',password('12345'),now())
```

The `password(...)` function will create a hash code—from which it is impossible to decypher the password. Anyone who has admin access to the database will not be able to see the clear text password.

For login, you would use a prepared statement:

```
select *
from user
where username=? and password=password(?)
```

and pass whatever username/password the user provides.

Note that anyone who has admin access to the database probably already has too much access to care for user passwords. Also the passwords can be guessed. If the attacker somehow gets a list of such encrypted passwords, they can setup a brute force attack, looping through all he 'common' passwords, and seeing whether any of those match up the hash code.

This is similar to how unix `/etc/passwd` file works. Newer releases don't let the world see the password hashcode, but instead store it in a `/etc/shadow` file. There's a utility called `jack` that will brute force a unix passwords file given the shadow file—surprisingly, it manages to guess most ‘weak’ passwords (ie: if the attacker manages to find a list of password hashes for 1000 users, they’re almost certain to guess a password and gain access into the system).

## 5 Encryption

Encryption is well beyond the scope of this short doc, 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 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-mails you send out are 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 to stop using the system in the first place.

### 5.1 Symmetric Encryption: Cyphers

Encrypting data with a password involves a *cypher*. There are many different kinds; from simple ones that simply add the password characters to the data one by one, to really confusing ones that join the password and the data via so many binary operations that guessing the original data without the password is extraordinarily difficult (assuming there are no flaws in the encription algorithm or its implementation).

There are two types of cyphers: block cyphers and stream cyphers. Both use a password. Block cyphers encrypt a block of data (64bits and up) at a time, while stream cyphers encrypt one character at a time. For block cyphers, data is often padded with random bits to make it fit the block size.

If you want to transmit data to someone, the other party needs also needs to know the password. You encrypt the file using some algorithm (implemented in some reputable software) using the password. You send the file. The other party, using the password, and same algorithm (usually using same software) can decrypt the file.

Below is an example using GnuPG:

```
$ echo "Hello World." > themessage.txt
```

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

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

```
$ gpg -c themessage.txt
Enter passphrase:
```

You need to type in the password (I used "12345"). This creates a `themessage.txt.gpg` file, which you can distribute, store, etc., and anyone who knows the password will be able to open it, ie:

```
$ gpg -d themessage.txt.gpg
gpg: CAST5 encrypted data
gpg: encrypted with 1 passphrase
Hello World.
gpg: WARNING: message was not integrity protected
```

You can also encrypt to an ascii file, ie:

```
$ gpg -c -a themessage.txt
```

The resulting file would look like this:

```
$ cat themessage.txt.asc
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.8 (GNU/Linux)

jA0EAwMCZFbSErlQBbRgyTG8GbfNxinzgNMsGDh8iEd2HG1ilXl8Jm5pTLEnlY/b
UFEr2GA/npAbebJcP7RSd73c
=LIAQ
-----END PGP MESSAGE-----
```

## 5.2 Public Key

Public key algorithms work on the idea of a one way function. The idea is this: If you wish to send someone a message, you ask them to give you a box with an open padlock. The box and padlock can travel without any security (we're discussing abstract boxes and padlocks, so nobody can close the lock, etc.). The worst that anyone could do is not pass on the box (simply not give it to us), or replace the box with a seemingly identical looking box.

When we get the box, we put our message into the box, and close the padlock. At this point, not even we can open it. The only person who can open the lock is whoever gave us the lock (they supposedly have a key). Again, just as before, the worst anyone can do is not pass on the box. They cannot temper or replace the box as the destination will not be able to open it with their key.

Public Key encryption is this above scheme only mathematically implemented. The lock is a one way function—that's easy to do one way and very difficult to do in reverse. Multiplication happens to be one such function—it is very easy to take two large numbers and multiply them together. It is very difficult to take one large number and find the factors.

With public key encryption everyone publishes their 'public' key in an open directory. Whenever you want to send someone a message, you encrypt it using their public key (lock). Whenever they wish to read such encrypted message, they open it with their private key.

The example below uses GnuPG to create public and private keys for "Prof Phreak".

```
$ gpg --gen-key
gpg (GnuPG) 1.4.8; Copyright (C) 2007 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) DSA and Elgamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
Your selection? 1
DSA keypair will have 1024 bits.
ELG-E keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y

Real name: Prof Phreak
Email address: profphreak@gmail.com
Comment:
You selected this USER-ID:
    "Prof Phreak <profphreak@gmail.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
```

We now have a public and private key. They're stored in `~/.gnupg` directory, and we can view the identification for them via: all public keys:

```
$ gpg --list-keys
/home/alex/.gnupg/pubring.gpg
-----------------------------
```

```
pub   1024D/CA71743F 2008-04-03
uid                  Alex Sverdlov <alex@theparticle.com>
sub   2048g/F973936E 2008-04-03

pub   1024D/52D96DE7 2008-04-03
uid                  Prof Phreak <profphreak@gmail.com>
sub   2048g/2B01ED36 2008-04-03
```

And all secret keys:

```
$ gpg --list-secret-keys
/home/alex/.gnupg/secring.gpg
-----------------------------
sec   1024D/52D96DE7 2008-04-03
uid                  Prof Phreak <profphreak@gmail.com>
ssb   2048g/2B01ED36 2008-04-03
```

You can improt/export them using this tool, ie: to see the actual public key for Prof Phreak,

```
$ gpg -a --export
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.8 (GNU/Linux)

mQGiBEfRteoRBADF7Zp6DvVee+fYWqigB9axfIpD3sA/9T78vmfNG7V4wglH6YvK
SWQkey94aJsGavmOV8SFrzK0jdqj0dso36nUvJWDu4I0jwnlWYFE2L+JMfcGsCSe
...
gzC1aA==
=IgB2
-----END PGP PUBLIC KEY BLOCK-----
```

Notice that we have a public key for `alex@theparticle.com`, lets say we want to send a message, we would do:

```
$ gpg -r alex@theparticle.com -e -a > messageforalex.txt
This is a very secret message for Alex.
```

This creates an ascii file `messageforalex.txt`, which Alex (the recipient), can open by using his private key, ie:

```
$ cat messageforalex.txt | gpg -d

You need a passphrase to unlock the secret key for
user: "Alex Sverdlov <alex@theparticle.com>"
```

```
2048-bit ELG-E key, ID F973936E, created 2008-04-03 (main key ID CA71743F)

gpg: encrypted with 2048-bit ELG-E key, ID F973936E, created 2008-04-03
      "Alex Sverdlov <alex@theparticle.com>"
This is a very secret message for Alex.
```

## 5.3 Digital Signatures

One can also do this public/private key thing in reverse. You can encrypt a message using your private key, then anyone who has your public key can decrypt it. Now, since the public key is 'public' (everyone knows it), it's not about securing the message. It's about authenticating you as the writer (as only you could've used your private key to encrypt the original message).

The way digital signatures work is: You take a file, generate some hash key from it (SHA1, MD5, etc.), then encrypt that hash code with your private key. You then attach this encrypted hash code along with the file.

Anyone who gets the file, can compute the same hash code from the data. They can then decrypt (using your public key) the encrypted hash code, and compare them. If they match, they can be reasonably sure that the message came from you, and it wasn't altered since you signed it.

This sort of 'authentication' is often used in conjunction with encryption. So not only can you send a message, the destination can also verify that it was you who sent it.

We first create our 'news' message—that people can verify against our public key:

```
$ echo "This news deserves to be heard." > news.txt
```

To sign with our `profphreak@gmail.com` key:

```
$ gpg --default-key profphreak@gmail.com --clearsign -a news.txt

You need a passphrase to unlock the secret key for
user: "Prof Phreak <profphreak@gmail.com>"
1024-bit DSA key, ID 52D96DE7, created 2008-04-03
```

The above creates `news.txt.asc`, ie:

```
$ cat news.txt.asc
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

This news deserves to be heard.
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (GNU/Linux)
```

```
iEYEARECAAYFAkf1OJcACgkQIeMTMlLZbefCeQCeOD/YLVeh6NAQhjhMwgBL7S15
CMUAn1NiE8b7JLUQZ6eU3RYF1V+eUQq3
=4qCC
-----END PGP SIGNATURE-----
```

Notice that the data itself is not encrypted, but has a "signature" attached. We can verify it via:

```
$ gpg --verify news.txt.asc
gpg: Signature made Thu Apr  3 16:05:43 2008 EDT using DSA key ID 52D96DE7
gpg: Good signature from "Prof Phreak <profphreak@gmail.com>"
```

Obviously we can also encrypt and sign the message, etc.

## 5.4 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).

## 5.5 Clever Public Key Attack

A few years ago, there was a clever timing attack on RSA and a bunch of other similar algorithms, that impacted SSL and a lot of other security protocols that incorporate such algorithms. The attack revolves around noting that the encryption/decryption algorithms are:

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

and

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

Now, we know that $n$ is *huge* (product of two large primes), but so are $e$ and $d$ (well, one of them is usually small). In any case, how does one compute this power operation? If $e$ is 100 digits long, you certainly don't want to do *that* many multiplications.

The solution comes in the form of an algorithm that relies on the `mod` operator to simplify things, ie (perl implementation):

```
# a^n mod z
sub exp_mod {
    my ($a,$n,$z) = @_;
    my $exp = 1;
    my $x = $a % $z;
    while($n > 0){
        $exp = ($exp * $x) % $z if $n % 2;
        $x = ($x*$x) % $z;
        $n = int($n/2);
    }
    return $exp;
}
```

In the inner loop, notice that we loop though the bits of $n$, and if a bit is set, we do a multiplication. So the running time of this relies entirely on the number of set bits in $n$.

So, if we time how long this takes to run (or somehow measure the power requirement of the chip, device, etc., that runs it) as it is running this algorithm, we'd have a direct correlation between running time and set bits in the secret key!

Now picture that we know that our 1000 bit key, has 900 set bits. How many trials would it take for us to find the correct key? We'll leave this as homework.

## 5.6 SSH, Secure Shell

Secure shell is a way of setting up a secure stream between two end points on a network. ie: secure telnet. It can be used to tunnel information (use it as a proxy).

Upon installation, SSH sets up a public/private key pair for all host related activities. This enables a concept of 'trusted' host, etc.

The most common authentication method is the password. When you try to connect, both server and client exchange their ‘public’ keys. Each side may or may not trust this public key. If a server suddenly ‘changed’ keys on some particular day, that may be a sign of someone impersonating the host. After the initial public key exchange, the end points agree on a symmetric cypher key, and use that cypher & key for further communications—starting with the password for authentication of the user. In this setup, the password is never sent in clear text. Anyone listening to the channel will simply see two public keys going by, and then nothing comprehensible.

SSH also supports key based authentication. The user generates a public/private key pair. The public key is then copied to the server. From that point on, the user can login without presenting a password, simply by using their private key. Note that the user’s private key never needs to go across the network, the client uses it’s private key to encrypt data which can be decrypted on the server side by the client’s public key. As before, both end points agree on a symmetric algorithm and password for the session, and use that symmetric cypher for the rest of the connection.

## 5.7 SSL

SSL is very similar to SSH in functionality, except it’s designed for the web. There’s also a lot more emphasis on preventing “man-in-the-middle” attacks.

The attack would work like this: An attacker would setup a website that looks exactly like amazon.com, or citibank.com, and would register it under a domain that’s a simple misspeling of the legitimate domain name. Users will either be tricked to click the domain (“Please update your account information”—email from your bank), or will mistakenly get to the site via a typo.

From then on, the user types their login information (maybe even using a secure token, etc.), which the attacker immediately forwards to the real bank. They may even get the account information and display it back to the user. As far as the user is concerned, they’ll be seeing their account—perfectly as they shold.

Instead of loging out however, the rogue website would make transfers to other accounts, possibly with a few dozen such redirects, before physically withdrawing the money (or purchased merchandise) in some foreign country with less-than-friendly relationship with US.

So the main security problem is: how to stop some rogue website impersonating a ‘real’ website. For this sole reason, we have ‘certificates’.

The theory is this: your computer will have a public key of some big trustworthy company, such as VeriSign (do you trust them?, according to your computer you do!). These trustworthy companies are known as ‘Certificate Authorities’. Your web browser implicitly trusts a lot of such entities.

A website operator would make a public/private key pair, and (after presenting some other information, proving their trustworthyness) would get their public key signed by the private key of certificate autheority.

Now when anyone goes to the website, they’ll get the site’s public key—why should they trust it? Well, because they can verify that the public key is signed by the certificate authority, which your web-browser trusts.

In a rogue website case, you would go to some site, and the public key it presents wouldn't be signed by the certificate authority, which would make your web-browser display a nasty 'invalid certificate' (or something along those lines) messages—which the user can ignore.

Sounds like a great setup? Unfortunately, certificate authorities are for-profit entities, and are motivated by... profit. Certificates cost money to get. Whenever you hand over money for something that's virtually 'free' for them (signing your public key with their private key is trivial, automated, and is virtually free), chances are, they'll be *happy* to do you this service, for which 99.9% of the money is pure profit (ignoring what they pay their website administrators). When they're that eager to get your business, what are the chances they'll turn you away because they don't consider you trustworthy? None!

So if you operate a rogue website, you will very likely be able to get a legitimate and widely trusted certificate authority to issue you a certificate (ie: sign your public key with their private key). And then any uses who goes to your site won't get a nasty pop-up telling them not to trust your site.

Of course certificate authorities can and do revoke keys (and this is part of the reason you need to do those updates to Windows once in a while). So if someone complains, the certificate authority will likely 'fix' the error in their next iteration of updates (users need to get the next batch of updates, etc.)

## 6 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.

### 6.1 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).

## 6.2 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.

Username: `'; drop table users; --`.

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.

## 7 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.

## 8 Network Scanning

There are various network scanning programs, like `nmap`, `Ethereal`, `tcpdump`, 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.

# 9 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; though it causes silly behavior like renaming .exe files into .txt and zipping them, etc.).

## 9.1 Trojans

A user can often be fooled into entering their username/password into a program they believe to be legitimate. For example, a unix login prompt seems legitimate. Whenever a user wishes to login, they typein the username/password, and press enter. It takes very little skill to create a 'real looking' login program for unix (simply printf the login prompt, read username, prompt for password, setup no echo on terminal, read password, and then simply email it somewhere or append it to a file.

Take a look at `/etc/issue` file, and then do something similar:

```
Red Hat Enterprise Linux ES release 4 (Nahant Update 6)
Kernel 2.6.9-55.ELsmp on an i686

server login:
```

Most users will simply proceed to type in their username/password at such a prompt. The program just has to return invalid login. Users in a computer center (common environment for such things) will eventually give up and move to a different workstation.

Another class of these programs arrives as attachments to some programs or emails. You download some shareware program, double click on the exe file, only to have the program replace your `svchost.exe` file, or something. The motivation for this is varied, but is generally malitious in some way.