Security engineering

Paul Jackson

School of Informatics
University of Edinburgh

What is “security”?

A large topic – see UG3 course Computer Security.

Here we will not discuss topics such as multi-level security, encryption, operating system exploits, . . .

We’ll look at high-level definitions, and briefly at programming practices in normal application software.

Security requirements

Security is an important class of non-functional requirements including:

- confidentiality – information does not leak
- integrity – information cannot be tampered with
- authentication – parties are who they say they are
- authorisation – parties can only do what they should be able to do
- non-repudiability – agreements made cannot be denied later, c.f., digital signatures.

Availability, performance etc. can also be compromised by (among other things) denial of service attacks as a result of security failures.

Why be paranoid?

Because they *are* out to get you.

There is a large number of people out on the Net who will attack your system. Some are "script kiddies"; some are spammers; some are extortionists ...

Informatics perimeter firewall blocks several hundred thousand probes per day.

NB some aspects of security mandated by law, e.g. Data Protection Act imposes responsibility to keep certain data appropriately private. Also there is an ethical dimension to security.

Typical attack

Machine runs a publicly accessible service, such as `sshd`.

Attackers send a carefully crafted very long string to `sshd`. `sshd` didn't check length of input from network – input buffer overflows into program code or function stack.

Result: attacker gets their code executed by `sshd` – which runs privileged. Machine is now totally compromised, as is everything that trusts it.

Once discovered, such a penetration costs many person-weeks of effort to clear up after.

Attacker may, of course, be authorized user of machine, trying to get unauthorized privilege. Or an authorized user who has been cracked ...

“Monster Mitigations” from CWE

Common Weakness Enumeration: a list of types of security weaknesses that arise in software, aiming to help in discussing and improving security. Annual report.

The “monster mitigations” are high-level principles that are effective against many weaknesses.

1. Establish and maintain control over all of your inputs.
2. Establish and maintain control over all of your outputs.
3. Lock down your environment.
4. Assume that external components can be subverted, and your code can be read by anyone.
5. Use industry-accepted security features instead of inventing your own.

M1: Establish and maintain control over all of your inputs

Many attacks work by handing the program illegal input that causes unintended behaviour. So validate input:

- deal with input of any length
- deal with arbitrary binary bytes (beware character encoding systems)
- check for specific escape characters (e.g. HTTP, HTML)
- ensure numerical data is within intended range
- check validity of URLs, filenames, etc.
- check cookies (ideally, only ever send out cryptographically signed data in cookies)

M1 continued

- Beware that doing validity checking is not easy. Use a trusted library if possible.
- Sometimes you handle input which is effectively a program that will be executed by an application. E.g. HTML with Javascript. It is particularly hard to check this for malicious usages. Why?
- complete mediation: validating data is no good if there's a back door (e.g. input taken from a user's file) to get bad data in: make sure validation happens at a bottleneck.
- avoid sharing: data in shared places opens possibilities for information flow you didn't expect

M1 example: Buffer overflow

The classic attack. The solution: always check the length! Or use routines that truncate the input to fit the buffer. (But is it OK silently to change user's input?)

In pure Java, buffer overflow can't happen (why not?). But Java implementations may use native C libraries in some classes.

M2: Establish and maintain control over all of your outputs.

Partner to M1. The danger is: you take untrusted input from somewhere, and use it to build some kind of message which you output to some component or application that will trust you. If your untrusted input is malicious, and you do not adequately control the structure and content of the output you construct, it may not be even structurally what you intended.

E.g. (from `http://cwe.mitre.org/data/definitions/116.html`)

```
<% String email = request.getParameter("email"); %>
...
Email Address: <%= email %>
```

M3: Lock down your environment

(more relevant to sysadmins than programmers) E.g.

- Least privilege: allow people/programs/classes to do only what they need to do. Run your code with the lowest privilege possible.
- Beware releasing information valuable to attackers in error messages
- Use whatever facilities are in your language/OS/compiler to check for vulnerabilities such as buffer overflow
- Keep up to date with security patches!

M4: Assume that external components can be subverted, and your code can be read by anyone

So:

- do not rely on "security by obscurity"
- more controversial: even go the other way, keep your design and code simple, and use open design, i.e. keep the security *mechanism* public
- if you're writing a server, don't let your security case depend on what the clients do, even if you're also writing the clients.

M5: Use industry-accepted security features instead of inventing your own.

Writing security algorithms – for authentication over an untrusted network, for encryption, etc. etc. – is *very very hard*. The literature is full of algorithms that turned out to have security flaws. As a non-expert you will almost certainly make a mistake – even as an expert you probably will!

So use standard, well-examined, trusted algorithms, and where possible, standard, well-examined, trusted implementations of them.

Beware race conditions

A race is when two (or more) events happen independently, and depending on the order, different things happen.

In particular, because of multi-tasking, arbitrary things may be done by other processes between any two lines of your program.

For example: "create file, protect it" is not safe – attacker may open file between creation and protection. It must be created in a safe state.

Beware social engineering

Your carefully crafted encryption, authentication and authorization scheme is no use if it all rests on a user's password being secret and the user falls for a phishing scam...

"Ninety per cent of office workers at London's Waterloo Station gave away their computer password for a cheap pen, compared with 65 per cent last year." (Informal study for InfoSecurity Europe 2003. NB no attempt made to verify the "passwords".)

`http://www.theregister.co.uk/2003/04/18/office_workers_give_away_passwords/`

Reading

Suggested: Browse CWE/SANS Top 25 Most Dangerous Software Errors

Suggested: Ross Anderson's paper *Why Information Security is Hard: an Economic Perspective*

Suggested: Secure Coding Guidelines for the Java Programming Language