Secure programming

Perdita Stevens

School of Informatics
University of Edinburgh

What is secure programming?

Mostly, not doing things.

In particular, not doing things that lead to the possibility of information being inappropriately released or modified, or unauthorized programs being executed.

In summary, confidentiality, integrity, availability.

Typical Java projects have 1–2 security-critical defects per kLoc...

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 ...

Our perimeter firewall blocks several hundred thousand probes per day.

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 ...

Security

is a huge topic. See third year course Computer Security.

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

We just consider programming practices in normal application software.

Validate input

Many attacks work (as above) 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)

Beware that doing validity checking is not easy. Use a trusted library if possible.

Note also that this applies to the content of user files!

Validate programmatic input

Sometimes you handle input which is effectively a program that will be executed by an application. E.g. HTML with Javascript.

It is very hard to check this for malicious usages. Try!

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. But Java implementations may use native C libraries in some classes.

General principles

Follow general principles to reduce possibility of security exposures, and mitigate results of compromise:

- least privilege: allow people/programs/classes to do only what they need to do
- KISS! Keep it simple, stupid: clear simple code is easier to check
- open design: keep the security *mechanism* public
- complete mediation: validating data is no good if there's a back door 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

Reading

Suggested: Ross Anderson's paper *Why Information Security is Hard: an Economic Perspective* (see web) http://www.cl.cam.ac.uk/ftp/users/rja14/econ.pdf

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.