CSE 303: Concepts and Tools for Software Development

Dan Grossman

Winter 2006

Lecture 23— Security, Defensive Programming

Security

Computer security is a huge area; we can't cover it in 45 minutes.

Robust software (no buffer-overflows, uncaught exceptions, etc.) is necessary but *in no way* sufficient to achieve "security".

Non-coding examples:

- Guessable passwords
- File permissions
- Tricking humans (email clicks, ...)

Subtle coding examples:

- Timing and other *covert* channels (classic example: early-exit password checking)
- File-system tricks (relative paths and races on accesses)

Security is hard!

Security is Worst-Case

What you cannot say when writing a function:

- I can't imagine anyone ever passing arguments like that.
- If the arguments don't make sense, "behavior is unpredicatable".
- I'll get it working now, and close the security holes later when I have time.

Some mottos/axioms:

- Principle of least privilege: Give each entity no more rights than it needs to accomplish its task.
- Obscurity is not security.
  - Guessing URLs, phone-tree numbers, back-doors, etc.

Also: simple, well-defined security policies separate from implementations

Security-breach impact

Also be clear about what is at stake:

- Resource-consumption (denial-of-service)
- Data revealing (privacy, theft)
- Data corruption (destruction of property)
- Full machine compromise (run arbitrary programs on attacked computer)

Take preventative measures. Example: the CSE department uses a different webserver for CGI scripts and it does *not* have access to home directories.

Check your inputs

Any nontrivial program has *untrusted inputs* (command-line args, files, mouse-clicks, etc.)

- What properties do you expect them to have (integers, buffer-lengths, alphabetical characters, ...)
- **Check these properties!!!**

Bad examples:

- `printf(argv[0])`
- `char x[256]; strcpy(x,argv[1]);`
- SQL-injection attacks
  - See bash analogy.

A sad tale: The FUZZ studies

Input-checking is more general

"Defensive programming" is good software-engineering practice regardless of security:

- Check your function inputs (at least of public) methods
  - At compile-time if possible
  - At run-time if possible
  - Not just in debug-mode (if not too expensive)
  - Not everything is possible
- Assertions are so common, Java added them to the language.
- Preaching: How can assertions be more important during testing?!

Copy your inputs/outputs

In the presence of mutation (as in C and Java), checking inputs is not always enough:

- What if the untrusted source can change the inputs after you check them but before you use them.
- What if you give out pointers to internal data and untrusted recipient assigns through the pointers.

Java example (security flaw in JDK1.1) a class's permissions:

```
public class Class {
  private Identity[] signers;
  public Identity[] getSigners() { return signers; }
  ...
}
```

Another motto: "copy-in/copy-out"

A note on ethics

An analogy: Engineers learning how to make strong glass might learn about the weakenesses of glass, how one can throw rocks through glass, etc.

It is still illegal to break a window that is not yours and dangerous to throw rocks.

Giving examples of hacks can make you a more secure programmer. Unleashing hacks can lead to long "government-sponsored vacations".