Security principles

- Variations of lists of security principles appear in literature & on-line (see course website)

- Security vulnerabilities often exploit violations of these principles
- Good security solutions or countermeasures follow these principles

- Some overlap & some tension between principles

- More generally, *checklists* are useful for security

Security principles

- secure the weakest link
- defence in depth
- principle of least privilige
- minimise attack surface
- compartmentalize
- secure defaults
- keep it simple
- fail securely
- promote privacy
- hiding secrets is hard
- use community resources
- be reluctant to trust
- ....

Security principles

These principles can be applied at *many levels*, eg.

- in source code of a application
- between applications on a machine
- at OS level
- at network level
- within an organisation
- between organisations
- ...

Secure the weakest link

- Spend your efforts on improving the security of the weakest part of a system, as this is where attackers will attack

- NB this requires a good *risk* analysis

Secure the weakest link

- educating users may be best investment to improve security
  - eg think of phishing attacks, weak passwords
- web application visible through firewall may be easier to break than the firewall
  - improve web application security, not the firewall

Example: using credit cards on internet

- Internet folklore: don't use your credit card online, or only over SSL-encrypted connection; better to fax it;...

- *NB threats vs risk:*
  - *threats* to credit card numbers include:
    - me loosing my wallet
    - eavesdropping on the network
    - stealing from customer database
  - The last poses a bigger *risk:*
    - end-points of SSL connection are the weak points

(Also, the risk to card holder is small.)

"Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench."

-- Gene Spafford

Practice defence in depth

- have several layers of security
  - two controls are better than one.
- no single point of failure

- A typical violation:
  having a firewall, and *only* having firewall
  - a user bringing in a laptop circumvents firewall
    - this is an example of enviromental creep

Defence in depth example

- have a firewall

and

- secure web application software

and

- run web application with minimal priviliges

Defence in depth example

- use OS access control to restrict access to sensitive files

and

- encrypt them

  - Esp. when files are written to removable media such as USB sticks (another example of enviromental creep), to laptops, or PCs which might be disposed off

Defence in depth example

"The only system which is truly secure is one which is switched off and unplugged, locked in a titanium-lined safe, buried in a concrete bunker, and surrounded by nerve gas and very highly paid armed guards.

Even then, I wouldn't stake my life on it"

-- Gene Spafford

Defence in depth: counterexample

- on UNIX systems, the password file, /etc/passwd, which contains hashed passwords, was world readable.

- better
  - hash passwords

  and
  - enforce tight access control to the file

Sun tarball problem (1993)

- *Every* tarball (zip-file) produced on Solaris 2.0 contained fragments of the password file **`/etc/password`**
- How did this happen?
  - **`tar`** looked up some user info directly prior to producing tarball:
    - password file was loaded in heap memory for this
    - this heap memory was then released
  - then **`tar`** allocated memory for constructing the tarball
    - allocated memory was always the memory just released
    - memory not zeroed out on allocation by program or OS…
- Solution: replacing **`char *buf (char*)malloc(BUFSIZE)`**
  by **`char *buf (char*)calloc(BUFSIZE)`**

Principle of least privilige

- be stingy with priviliges
  - only grant permissions that are really needed
  - resource permissions (eg memory limits, CPU priorities) , network permissions, file permissions, ...

- typical violations
  - logging in as root/administrator
  - device drivers having to run in kernel mode

- important cause of violations: laziness

Principle of least privilige

- in organisation
  - don't give everyone access to root passwords
  - don't give everyone administrator rights

- on computer
  - run process with minimal set of priviliges

  - Eg, don't run web application as root or administrator

Principle of least privilige

- for Java application

not the default policy

```
grant codeBase "file:${{java.ext.dirs}}/*" {
permission java.security.AllPermission;
};
```

but minimum required

```
grant codeBase "file:./forum/*" {
 permission java.security.FilePermission;
"/home/forumcontent/*","read/write";
};
```

Principle of least privilige

- in code
  - not `public int x;`
    but `private int x;`
  - not `public void m()`
    but `package void m()`
- Expose minimal functionality in interfaces of objects, classes, packages, applications

Principle of least privilige

- NB applying the principle of least privilige in code is tricky & hard and requires work & discipline.
  - this is true just for plp in general, not just in code
- Why?
  - compiler complains about private field that should be public, not the other way around
  - compiler complains about missing import, not about superfluous import.
- Can this be improved ?
  - tool support in compilers & IDEs, eg or separate source code analyzers, eg FindBugs and JAMIT tool for tightening visibility modifiers

(http://grothoff.org/christian/xtc/jamit)

Compartmentalize

- Principle of least privilige works best if access control is all or nothing for large chunks – compartments - of a system

- Motivations:
  - simplicity
  - containing attacker in case of failure

- Analagy: compartments on a ship
- Counterexample: OS that crashes if an application crashes

NB the fundamental conflict between

- principle of least privilege

and

- kiss principle – keep it simple
  - plp requires very fine-grained control with expressive policies
  - … which leads to more complexity
  - … which people then get wrong

- Compartmentalization can provide a solution

  using defence in depth

Compartmentalize examples

- Use different machines for different tasks
  - eg run web application on a different machine from employee salary database
- Use different user accounts on one machine for different tasks
  - unfortunately, security breach under one account may compromise both...
  - compartmentalization provided by typical OSs is poor!
- Partition hard disk and install OS twice

Improved compartmentalization

- **`chroot` jail**

  restricts access of a process to subset of file system, ie. changes the root of file system for that process

  Eg run an application you just downloaded with

  ```
  chroot /home/sos/paperino/trial;/tmp
  ```

  Nice idea, but hard to get working, and hard to get working correctly.

Improved compartmentalization

Examples in operating system world:

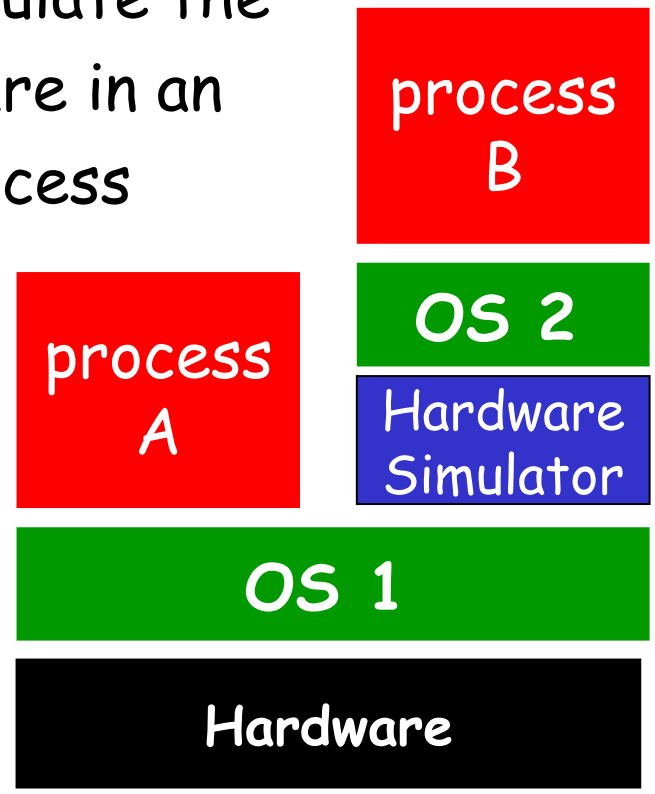
- virtual machines
  - VMWare
  - very popular these days, but mainly for reasons of convenience & costs, not security

- operating system hypervisors (true microkernels)

  small, lightweight kernel, which partions hard disk & memory, to concurrently run several copies of the OS, in different compartments

Virtualisation by virtual machine

- We simulate the hardware in an OS process

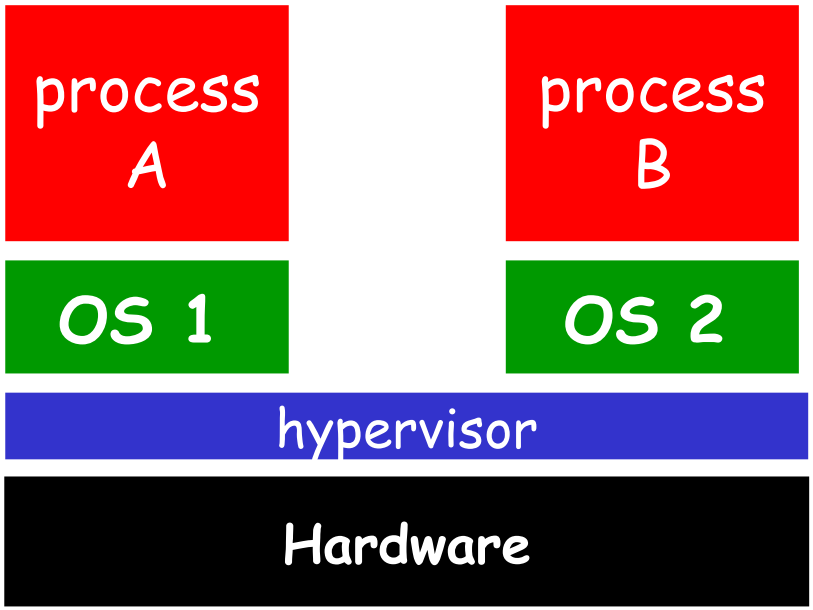


Similar to Java VM, except that we simulate the real hardware, and don't provide some abstract VM

This is solution proposed by VMware.com

Virtualisation by hypervisor

- We simulate the hardware below the operation system, in a so-called hypervisor aka micro-kernel



Compartmentalize

- in code, aka modularisation,
  - using objects, classes, packages, etc.

  Restrict sensitive operations to small modules, with small interfaces
    - so that you can concentrate efforts on quality of these modules
    - so that only these have to be subjected to code reviews

Minimize attack surface

Mimimise

- number of open sockets
- number of services
- number of services running by default
- number of services running with high priviliges
- number of dynamic content webpages
- number of accounts with administrator rights
- number of files & directories with weak access control
- ...

Minimize the attack surface

- in code
  - not `public int x;`
    but `private int x;`
  - not `public void m()`
    but `package void m()`

This is applying principle of least privilige, and also reduces attack surface, from buggy or hostile code

Minimize attack surface in *time*

Examples

- Automatically log off users after n minutes
- Automatically lock screen after n minutes
- Unplug network connection if you don't use it
- Switch off computer if you don't use it

- On smartcards, it's good practice to zero-out arrays that contains sensitive information (usually, decrypted information) as soon as it's no longer needed

Use secure defaults

- By default,
  - security should be switched on
  - permissions turned off
- This will ensure that we apply principle of least privilige
- Counterexample: on bluetooth connection on mobile phone is by default on, but can be abused

Keep it simple (aka economy of mechanism)

- Complexity important cause of security problems
  - complexity leads to unforeseen feature interaction
  - complexity leads to incorrect use and insecure configuration by users and developers
- Note: compartmentalization is a way of keeping access control simple
- Eg: good practice: choke point – small interface through which all control flow must pass

Fail securely

- Incorrect handling of unexpected errors is a major cause of security breaches
- Counterexamples:
  - fallback to unsafe(r) modes on failure
    - sometimes for backward compatibility
    - asking user if security settings can be lowered
  - crashing on failure, leading to DoS attack
  - leaking interesting information for an attacker
- Of course, having exceptions in a programming language has a big impact

Fail securely example

```
isAdmin = true; // enter Admin mode
try {
      something that may throw SomeException

} catch (SomeException ex) {
            // should we log?
          log.write(ex.toString());
            // how should we proceed?
          isAdmin = false;
            // or should we exit?
}
```

Variants of failing insecurely

- information leakage
  - about cause of error, which can be exploited –more on that later
- ignoring errors
  - Easier in a programming language without exceptions!
    - eg forgetting to check for -1 return value in C
- misinterpreting errors
- useless errors
  - why does strncopy return an error value at all?
- handling wrong exceptions
- handling all exceptions

Failing *in*securely example

```
char dest[19]
char *p = strncpy(dest, src,19)
    // strncpy returns dest, like strcpy
if (p) { // everything went fine, use dest or p
            .....
          }
```

- Programmer is careful to check return value of strncopy
- But strncopy will not return NULL on error

- Of course, having exceptions in a programming language has a big impact

Failing *insecurely* example

Example code in **`Local System`** service in Windows

```
ImpersonateNamedClient(someUser);
                        // become someUser
DeleteFile(fileName);
RevertToSelf();    // become Local System
```

What's wrong here ?

- What happens of ImpersonateNamedClient fails?

Failing *in*securely example

```
try {... // (1) Load XML file f from disk
     ... // (2) Use some data from f to get URI
     ... // (3) get X509 certificate
     ... // (4) access URI with certificate
} catch (Exception ex) {
     ....
}
```

What's wrong here ?

- one catch block to handle SecurityException, XMLException, IOException, FileNotFoundException, SocketException, ...

Failing *in*securely example

```
try {...
      ...
    } catch (Exception ex) {
     // do nothing
    }
```

What's possibly/probably wrong here ?

- empty catch block is suspicious...
- overly broad catches is suspicious

Failing insecurely example

Security Flaw in OpenSSL and OpenSSH

(CVE CAN-2000-0535)

- PRNG (Pseudo Random Number Generator) in OpenSSH and OpenSSL seeded with **`/dev/random`**
- But failure to check for the presence of **`/dev/random`** which did not exist on FreeBSD-Alpha...

Fail securely example

When scrubbing user input – ie *validating input* -

- don't reject input with illegal words
  - eg. webforum input containing `<javascript>`
- don't reject input with illegal characters
  - eg. input containing characters `< > ;`
- but *only* allow input with *legal* characters
  - eg. only input containing `0...9 a...z`

So user validation will err on the side of caution

Promote privacy

- Privacy of users, but also of systems

- Counterexamples
  -
```
> telnet somemachine
  Trying 123.1.2.3
  Connected to somemachine (123.1.2.3)
  Red Hat Linux release 7.0 (Hedwig)
  Kernel 2.2.16 on an i686
 login:
```
  - Smartcard chips still do this

Blind SQL injection

Suppose

```
http://newspaper.com/items.php?id=2
```

results in SQL injection-prone query

```
SELECT title, body FROM items WHERE ID=2
```

Will we see difference response to URLs below?

```
http://newspaper.com/items.php?id=2 AND 1=1
http://newspaper.com/items.php?id=2 AND 1=2
```

We can use this to find out things about the database structure & content

```
../items.php?id=2 AND SUBSTRING(user,1,1) = 'a'
```

Blind SQL injection: result to an SQL injection not visible, but leaks information

Blind SQL injection

Even all SQL errors report an identical, standard error page, without any further info, errors may still leak information

```
IF <some condition> SELECT 1 ELSE 1/0
```

Worse still, response time may still leak information

```
.. IF(SUBSTRING(user,1,1) ='a',
        BENCHMARK(50000, …), null)..
```

It's hard to keep secrets

- Don't rely on security by obscurity
  [Kerckhoffs principle]
- Don't assume attackers don't know the application source code, and can't reverse-engineer binaries
  - Don't hardcode secrets in code.
  - Don't rely on code obfuscation

- Example
  - DVD encryption
  - webpages with hidden URLs
  - passwords in javascript code – this happens!

Use community resources

- Never *design* your own cryptography
- Never *implement* your own cryptography

- Don't repeat known mistakes

  If you're making an application of kind X using programming language Y on platform Z and operatins system W, look for
  - known threats for application of kind X
  - known vulnerabilities in programming language Y and platform Z, ...
  - existing countermeasures

Use community resources

Use google, books, webfora, etc. to learn & reuse

- learn about vulnerabilities
  - and avoid making the same mistakes
- learn about solutions and countermeasures
  - and reuse them

Principle of psychological acceptance

- If security mechanism is too cumbersome, users will switch it off, or find clever ways around it
- User education may improve the situation, but only up to a point
- *How many security pop-ups can we expect the user to cope with, if any?*

Don't mix data & code

This is the cause of *many* problems, eg

- traditional buffer overflow attacks, which rely on mixing data and code on the stack
- VB scripts in Office documents
  - leads to attacks by hostile .doc or .xls
- javascript in webpages
  - leads to XSS (cross site scripting attacks)
- SQL injection relies on use data (user input!) as part of SQL query

Clearly assign responsibilities

At organizational level:

- eg. make one person responsible for something rather than two persons – or a whole group.

At coding level:

- make one module/class responsible for input validation, access control, ...
- for a method

```
public void process(String str)
```

is the caller or callee responsible for checking if for instance

```
str!=null & !(str.equals(“”)) ?
```

But still practice defence in depth...

Identify your assumptions

- including obvious, implicit assumptions
  - these may be sources of vulnerability, and may change in long run

Examples

- laptops invalidate implicit assumption that computers don't move past the company firewall
- assumption that user is a human may cause you to ignore possibility of brute-force password guessing
- TCP SYN flood attack exploits implicit assumptions in the spec *"If we receive a TCP packet with the SYN flag set, it means the sender wants to start a dialog with us"*
- assumption that `new LoginContext()` won't throw an `OutOfMemoryException`
- assumption that a Java applet won't use all CPU time

Be reluctant to trust

- Understand and respect the chain of trust
- NB trust is transitive
- NB trust is *not* a good thing
- NB "trusted" is not the same as "trustworthy"
- Minimize Trusted Computing Base (TCB), ie that part of the system (software and hardware) that *has* to be trusted

Ken Thompson (Reflections on trusting trust)

Backdoor in UNIX and Trojan in C-compiler revealed during Turing award lecture

1. backdoor in `login.c` of UNIX

```
if (name == "ken") {don't check password;
                    log in as root}
```

2. code in C compiler to add backdoor when recompiling login.c
3. code in C compiler to add code (2 & 3!) when (re)compiling a compiler

Be reluctant to trust

- All user input is evil !
  - Eg unchecked user input leads to
    - buffer overflows
    - SQL injection
    - XSS on websites
  - User input includes cookies, environment variables, ...

  User input should not be trusted, and subjected to strong input validation checks before being is used
- Don't trust third-party software

Security principles

- secure the weakest link
- defence in depth
- principle of least privilige
- minimise attack surface
- compartementalise
- secure defaults
- keep it simple
- fail securely
- promote privacy
- hiding secrets is hard
- use community resources
- be reluctant to trust
- ...