Software Security

Information Security

Prof Hans Georg Schaathun

University of Surrey/Ålesund University College

Autumn 2011 – Week 12

Outline

Session objectives

- Be familiar with the most common implementation errors leading to security vulnerabilities
- Start developing a good methodology for secure design and implementation

- 2010 CWE/SANS Top 25 Most Dangerous Software Errors
- Robert Seacord: *Secure Coding in C and C++*
  - `https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices`

Security or Useability

- This chapter is largely about *software bugs*
  - Is this security?
  - . . . or is it useability?

Security or Useability

- This chapter is largely about *software bugs*
  - Is this security?
  - . . . or is it useability?
- Answer is *yes*
  - Bugs are user (programmer) mistakes – useability.
  - Many bugs *cause* security vulnerabilities.
- Useability is a prerequisite of security.

Security or Useability

- This chapter is largely about *software bugs*
  - Is this security?
  - . . . or is it useability?
- Answer is *yes*
  - Bugs are user (programmer) mistakes – useability.
  - Many bugs *cause* security vulnerabilities.
- Useability is a prerequisite of security.

Outline

Common Weakness Enumeration

- 2010 CWE/SANS Top 25 Most Dangerous Software Errors
  - `http://cwe.mitre.org/top25/index.html`
- A very few key vulnerabilities behind most incidents
- Massive benefit from controlling the top few

Top 9

1. Improper neutralisation of input during web page generation (Cross-Site Scripting)
2. Improper neutralisation of Special Elements in SQL Commands (SQL Injection)
3. Buffer overflow without Checking of Input Size
4. Cross-Site Request Forgery
5. Improper Access Control (Authorisation)
6. Reliance on Untrusted Inputs in a Security Decision
7. Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)
8. Unrestricted Upload of File with Dangerous Type
9. Improper neutralisation of Special Elements used in an OS Command

Top 9

1. Improper neutralisation of input during web page generation (Cross-Site Scripting)
2. Improper neutralisation of Special Elements in SQL Commands (SQL Injection)
3. Buffer overflow without Checking of Input Size
4. Cross-Site Request Forgery
5. Improper Access Control (Authorisation)
6. Reliance on Untrusted Inputs in a Security Decision
7. Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)
8. Unrestricted Upload of File with Dangerous Type
9. Improper neutralisation of Special Elements used in an OS Command

Trusting Input

Most of the top vulnerabilities relate to user input ...

- Cross-Site Scripting
- SQL Injection
- Reliance on Untrusted Input
- File upload
- Path traversal
- Special elements in OS commands

Integrity of Code and Data ...

Trusting Input

Most of the top vulnerabilities relate to user input ...

- Cross-Site Scripting
- SQL Injection
- Reliance on Untrusted Input
- File upload
- Path traversal
- Special elements in OS commands

Integrity of Code and Data ...

Outline

Top 9

1. Improper neutralisation of input during web page generation (Cross-Site Scripting)
2. Improper neutralisation of Special Elements in SQL Commands (SQL Injection)
3. Buffer overflow without Checking of Input Size
4. Cross-Site Request Forgery
5. Improper Access Control (Authorisation)
6. Reliance on Untrusted Inputs in a Security Decision
7. Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)
8. Unrestricted Upload of File with Dangerous Type
9. Improper neutralisation of Special Elements used in an OS Command

Input Checking

- 4 out of 9 vulnerabilities
  - very similar instances of input checking
- E.G. SQL injection
  - `SELECT * FROM users WHERE name='John' ;`
- Now, say the user enters a name, instead of using `'John'`
  - `SELECT * FROM users WHERE name='$n' ;`
- What if the user enters
  - `Mary' ; DROP TABLE users ; ... '`
- *What happens?*

Input Checking

- 4 out of 9 vulnerabilities
  - very similar instances of input checking
- E.G. SQL injection
  - `SELECT * FROM users WHERE name='John' ;`
- Now, say the user enters a name, instead of using `'John'`
  - `SELECT * FROM users WHERE name='$n' ;`
- What if the user enters
  - `Mary' ; DROP TABLE users ; ... '`
- *What happens?*

Input Checking

- 4 out of 9 vulnerabilities
  - very similar instances of input checking
- E.G. SQL injection
  - `SELECT * FROM users WHERE name='John' ;`
- Now, say the user enters a name, instead of using `'John'`
  - `SELECT * FROM users WHERE name='$n' ;`
- What if the user enters
  - `Mary' ; DROP TABLE users ; ... '`
- *What happens?*

Input Checking

- 4 out of 9 vulnerabilities
  - very similar instances of input checking
- E.G. SQL injection
  - `SELECT * FROM users WHERE name='John' ;`
- Now, say the user enters a name, instead of using `'John'`
  - `SELECT * FROM users WHERE name='$n' ;`
- What if the user enters
  - `Mary' ; DROP TABLE users ; ... '`
- *What happens?*

Input Checking

- 4 out of 9 vulnerabilities
  - very similar instances of input checking
- E.G. SQL injection
  - `SELECT * FROM users WHERE name='John' ;`
- Now, say the user enters a name, instead of using `'John'`
  - `SELECT * FROM users WHERE name='$n' ;`
- What if the user enters
  - `Mary' ; DROP TABLE users ; ...  '`
- *What happens?*

UNIVERSITY OF SURREY

Input Checking

- 4 out of 9 vulnerabilities
  - very similar instances of input checking
- E.G. SQL injection
  - `SELECT * FROM users WHERE name='John' ;`
- Now, say the user enters a name, instead of using `'John'`
  - `SELECT * FROM users WHERE name='$n' ;`
- What if the user enters
  - `Mary' ; DROP TABLE users ; ... '`
- *What happens?*

What may happen

```
SELECT * FROM users WHERE name='Mary' ; DROP TABLE
users ; ... ''
```

- We select user Mary, and then drop the table
  - Successful availability attack — the table is destroyed
- The string delimiter (') in the input
  - allows the user to terminate the string (which was expected)
  - and add another command (which was not expected)

What may happen

```
SELECT * FROM users WHERE name='Mary' ; DROP TABLE
users ; ...  ''
```

- We select user Mary, and then drop the table
  - Successful availability attack — the table is destroyed
- The string delimiter (') in the input
  - allows the user to terminate the string (which was expected)
  - and add another command (which was not expected)

What should happen

```
SELECT * FROM users WHERE name='Mary'' ; DROP TABLE
users ; ...  ''
```

- The special character is escaped
  - and treated as part of the string
- The offending Command is now part of the name
  - and not harmful

What should happen

```
SELECT * FROM users WHERE name='Mary'' ; DROP TABLE
users ; ... ''
```

- The special character is escaped
  - and treated as part of the string
- The offending Command is now part of the name
  - and not harmful

What should happen

```
SELECT * FROM users WHERE name='Mary'' ; DROP TABLE
users ; ... ''
```

- The special character is escaped
  - and treated as part of the string
- The offending Command is now part of the name
  - and not harmful

Cross-Site Scripting

```
http://www.phpnuke.org/user.php?op=userinfo&uname=
<script>alert(document.cookie);</script>
```

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: `http://www.cgisecurity.com/xss-faq.html`

Cross-Site Scripting

```
http://www.phpnuke.org/user.php?op=userinfo&uname=
<script>alert(document.cookie);</script>
```

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: `http://www.cgisecurity.com/xss-faq.html`

UNIVERSITY OF SURREY

Cross-Site Scripting

`http://www.phpnuke.org/user.php?op=userinfo&uname=<script>alert(document.cookie);</script>`

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: `http://www.cgisecurity.com/xss-faq.html`

Cross-Site Scripting

```
http://www.phpnuke.org/user.php?op=userinfo&uname=
<script>alert(document.cookie);</script>
```

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: `http://www.cgisecurity.com/xss-faq.html`

Cross-Site Scripting

`http://www.phpnuke.org/user.php?op=userinfo&uname=<script>alert(document.cookie);</script>`

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: `http://www.cgisecurity.com/xss-faq.html`

Cross-Site Scripting

```
http://www.phpnuke.org/user.php?op=userinfo&uname=
<script>alert(document.cookie);</script>
```

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: `http://www.cgisecurity.com/xss-faq.html`

Cross-Site Scripting

```
http://www.phpnuke.org/user.php?op=userinfo&uname=
<script>alert(document.cookie);</script>
```

- Malicious code passed as an HTTP GET argument
- Principle as before
- No input checking in the web page
  - causes execution of code from the user
- No limit to what this can achieve
- Other web pages (other sites)
  - can hide code actually loading the URL
  - no user interaction at all

Source: *http://www.cgisecurity.com/xss-faq.html*

Path traversal

- Say you allow uploading and downloading of files.
  - the user specifies the filename
  - a directory is hardcoded and prepended
- so the user enters `foobar.jpeg`
  - it becomes `/opt/archive/foobar.jpeg`
  - safe enough
- What if the user enters `../../etc/passwd`?

Path traversal

- Say you allow uploading and downloading of files.
  - the user specifies the filename
  - a directory is hardcoded and prepended
- so the user enters `foobar.jpeg`
  - it becomes `/opt/archive/foobar.jpeg`
  - safe enough
- What if the user enters `../../etc/passwd`?

Path traversal

- Say you allow uploading and downloading of files.
  - the user specifies the filename
  - a directory is hardcoded and prepended
- so the user enters `foobar.jpeg`
  - it becomes `/opt/archive/foobar.jpeg`
  - safe enough
- What if the user enters `../../etc/passwd`?

Path traversal

- Say you allow uploading and downloading of files.
  - the user specifies the filename
  - a directory is hardcoded and prepended
- so the user enters `foobar.jpeg`
  - it becomes `/opt/archive/foobar.jpeg`
  - safe enough
- What if the user enters `../../etc/passwd`?

Dangerous Filenames

- Some systems deduce file type from filename
  - e.g. `*.jpg` for JFIF/EXIF images
  - e.g. `*.php` for PHP scripts
- Dangerous: the filename is under user control
- Allows user to mark data as code and vice versa
- E.g. uploaded files on a web server
  - do you allow the users to upload PHP scripts?
  - might your server execute them?

Dangerous Filenames

- Some systems deduce file type from filename
  - e.g. `*.jpg` for JFIF/EXIF images
  - e.g. `*.php` for PHP scripts
- Dangerous: the filename is under user control
- Allows user to mark data as code and vice versa
- E.g. uploaded files on a web server
  - do you allow the users to upload PHP scripts?
  - might your server execute them?

Dangerous Filenames

- Some systems deduce file type from filename
  - e.g. `*.jpg` for JFIF/EXIF images
  - e.g. `*.php` for PHP scripts
- Dangerous: the filename is under user control
- Allows user to mark data as code and vice versa
- E.g. uploaded files on a web server
  - do you allow the users to upload PHP scripts?
  - might your server execute them?

Dangerous Filenames

- Some systems deduce file type from filename
  - e.g. `*.jpg` for JFIF/EXIF images
  - e.g. `*.php` for PHP scripts
- Dangerous: the filename is under user control
- Allows user to mark data as code and vice versa
- E.g. uploaded files on a web server
  - do you allow the users to upload PHP scripts?
  - might your server execute them?

Passing information to external programs

Improper neutralisation of Special Elements used in an OS Command

- Calling external programs is high risk
  - library calls is lower risk
  - Why is this?
- library calls provide type checking
  - external programs take arguments as strings
  - ... control codes and data are mixed

Passing information to external programs

Improper neutralisation of Special Elements used in an OS Command

- Calling external programs is high risk
  - library calls is lower risk
  - Why is this?
- library calls provide type checking
  - external programs take arguments as strings
  - ... control codes and data are mixed

The rlogin bug

- rlogin(1) used to allow remote login access to Unix systems
  - `rlogin` [`-l`*user*] *hostname*
- The rlogin client contacts a remote host which runs login(1)
  - Running `rlogin -l css1hs kyle`, would
  - ...on kyle, cause the running of `login css1hs`.
- Now, login(1) has many uses,
  - `login -froot` is a forced login (as root)
  - ... no password prompt
- `rlogin -l -froot kyle` – what happens?
  - `login -froot` – superuser login without password
  - Unused functionality is exploited.
  - ... unless `rlogin` sanitises the input

The rlogin bug

- rlogin(1) used to allow remote login access to Unix systems
  - **`rlogin`** [**`-l`***user*] *hostname*
- The rlogin client contacts a remote host which runs login(1)
  - Running **`rlogin -l css1hs kyle`**, would
  - ...on kyle, cause the running of **`login css1hs`**.
- Now, login(1) has many uses,
  - **`login -froot`** is a forced login (as root)
  - ... no password prompt
- **`rlogin -l -froot kyle`** – what happens?
  - **`login -froot`** – superuser login without password
  - Unused functionality is exploited.
  - ... unless **`rlogin`** sanitises the input

The rlogin bug

- rlogin(1) used to allow remote login access to Unix systems
  - **`rlogin`** [**`-l`**_user_] _hostname_
- The rlogin client contacts a remote host which runs login(1)
  - Running **`rlogin -l css1hs kyle`**, would
  - …on kyle, cause the running of **`login css1hs`**.
- Now, login(1) has many uses,
  - **`login -froot`** is a forced login (as root)
  - ... no password prompt
- **`rlogin -l -froot kyle`** – what happens?
  - **`login -froot`** – superuser login without password
  - Unused functionality is exploited.
  - ... unless **`rlogin`** sanitises the input

What to do?

- Two methods (principles):
  1. Input checking: reject unexpected input
  2. Sanitising: accept the input, make sure it is handled correctly (escaping)
- Which is easiest? Which is best?
- Restrictive Input Checking
  - err on the side of caution
  - relatively simple — accept a small set of safe inputs
  - availability risk (reject good input)
  - good incidence response allow quick bug fixes
- Sanitising requires comprehensive understanding
  - how to sanitise
  - what is the effect of each possible input?
  - the SQL example cannot be solved by input checking
    - O'Brian is a valid name ...

What to do?

- Two methods (principles):
  1. Input checking: reject unexpected input
  2. Sanitising: accept the input, make sure it is handled correctly (escaping)
- Which is easiest? Which is best?
- Restrictive Input Checking
  - err on the side of caution
  - relatively simple — accept a small set of safe inputs
  - availability risk (reject good input)
  - good incidence response allow quick bug fixes
- Sanitising requires comprehensive understanding
  - how to sanitise
  - what is the effect of each possible input?
  - the SQL example cannot be solved by input checking
    - O'Brian is a valid name ...

What to do?

- Two methods (principles):
  1. Input checking: reject unexpected input
  2. Sanitising: accept the input, make sure it is handled correctly (escaping)
- Which is easiest? Which is best?
- Restrictive Input Checking
  - err on the side of caution
  - relatively simple — accept a small set of safe inputs
  - availability risk (reject good input)
  - good incidence response allow quick bug fixes
- Sanitising requires comprehensive understanding
  - how to sanitise
  - what is the effect of each possible input?
  - the SQL example cannot be solved by input checking
    - O'Brian is a valid name ...

Good practice

1. Take a critical view of all input
   - Don't trust anyone
2. Have a firm understanding of what the input should look like
   - don't accept odd input
3. Be aware of any special characters where the data is used
   - be wary of quotation marks ('/"), backslashes, control characters etc.
   - special scenarios like slashes in filenames
4. Don't use user input if you do not have to
   - e.g. filenames can be generated by the system
5. Spend some time on every instance of user input

Outline

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Decision Making

Reliance on Untrusted Inputs in a Security Decision.

- Decision Making depends on Information
- Where does this information come from?
- What CObIT Criteria are essential for this information?
  - Integrity
  - Reliability of Management Information
- Can your adversaries have forged information?
- Are your decissions steered by your enemy?

Untrusted Inputs in a Security Decision

- What controls can you use against this?
  - Technical
    - Unlikely – Intelligent Input needed to choose trusted sources
  - Operational
    - Yes – good operational information gathering
  - Managerial
    - Yes – choose trust policy

Untrusted Inputs in a Security Decision

- What controls can you use against this?
  - Technical
    - Unlikely – Intelligent Input needed to choose trusted sources
  - Operational
    - Yes – good operational information gathering
  - Managerial
    - Yes – choose trust policy

Untrusted Inputs in a Security Decision

- What controls can you use against this?
  - Technical
    - Unlikely – Intelligent Input needed to choose trusted sources
  - Operational
    - Yes – good operational information gathering
  - Managerial
    - Yes – choose trust policy

Untrusted Inputs in a Security Decision

- What controls can you use against this?
  - Technical
    - Unlikely – Intelligent Input needed to choose trusted sources
  - Operational
    - Yes – good operational information gathering
  - Managerial
    - Yes – choose trust policy

Quick Summary

- Decissions are based on Information
- *ensure reliability and integrity* of this information

Top 9

1. Improper neutralisation of input during web page generation (Cross-Site Scripting)
2. Improper neutralisation of Special Elements in SQL Commands (SQL Injection)
3. Buffer overflow without Checking of Input Size
4. Cross-Site Request Forgery
5. Improper Access Control (Authorisation)
6. Reliance on Untrusted Inputs in a Security Decision
7. Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)
8. Unrestricted Upload of File with Dangerous Type
9. Improper neutralisation of Special Elements used in an OS Command

Buffer Overflows

- Classic problem
- Limited memory buffer
  - writing unlimited data objects – often user input
  - system does not check the buffer limits
- Clever attackers can
  - overwrite executable code
  - inserting custom code to be executed

Buffer Overflows

- Classic problem
- Limited memory buffer
  - writing unlimited data objects – often user input
  - system does not check the buffer limits
- Clever attackers can
  - overwrite executable code
  - inserting custom code to be executed

Why is this code insecure?

```
bool IsPasswordOkay(void) {
  char Password[12] ;

  gets(Password) ;
  if ( !strcmp( Password, "goodpass" ) ) return true ;
  else return (false) ;
}

void main(void) {
  bool PwStatus ;

  puts ("Enter␣password:") ;
  PwStatus = IsPasswordOkay() ;
  if (PwStatus == false) {
    puts("Access␣denied") ;
    exit (−1) ;
  }
  else puts ("Access␣granted") ;
}
```

Cross-Site Request Forgery (CSRF)

like the stranger in the airport, asking you to take just this parcel along on the flight ...

- Web vulnerability
  - trick a user's client to make your request
  - request made with his credentials
- Integrity problem
  - Attackers can forge requests
- The attacker can gain all the priviliges of the user

Cross-Site Request Forgery (CSRF)

like the stranger in the airport, asking you to take just this parcel along on the flight ...

- Web vulnerability
  - trick a user's client to make your request
  - request made with his credentials
- Integrity problem
  - Attackers can forge requests
- The attacker can gain all the priviliges of the user

Cross-Site Request Forgery (CSRF)

like the stranger in the airport, asking you to take just this parcel along on the flight ...

- Web vulnerability
  - trick a user's client to make your request
  - request made with his credentials
- Integrity problem
  - Attackers can forge requests
- The attacker can gain all the priviliges of the user

Improper Access Control

- Fairly obvious – restrict access to authorised users
- But, get the roles right
  - should match business roles
- The exercise for next week explores management of
  - access
  - privileges
  - identity

Improper Access Control

- Fairly obvious – restrict access to authorised users
- But, get the roles right
  - should match business roles
- The exercise for next week explores management of
  - access
  - privileges
  - identity

Improper Access Control

- Fairly obvious – restrict access to authorised users
- But, get the roles right
  - should match business roles
- The exercise for next week explores management of
  - access
  - privileges
  - identity

Outline

Seacord’s 10 Principles

- Validate input
- Heed compiler warnings
- Architect and design for security policies.
- Keep it simple.
- Default deny.
- Adhere to the principle of least privilege.
- Sanitize data sent to other systems.
- Practice defense in depth.
- Use effective quality assurance techniques.
- Adopt a secure coding standard.

Default Deny

or principle of least privilege

- Default Deny is a General Principle with many Applications
  - Access Control
  - Input Validation
  - Feature Selection
- Advantage: prevents *unnecessary* integrity/confidentiality risks
  - accepting risks only when necessary
- Disadvantage: availability risk
  - Mitigation by incident respons $\rightarrow$ bug fix

Default Deny

or principle of least privilege

- Default Deny is a General Principle with many Applications
  - Access Control
  - Input Validation
  - Feature Selection
- Advantage: prevents *unnecessary* integrity/confidentiality risks
  - accepting risks only when necessary
- Disadvantage: availability risk
  - Mitigation by incident respons → bug fix

Default Deny

or principle of least privilege

- Default Deny is a General Principle with many Applications
  - Access Control
  - Input Validation
  - Feature Selection
- Advantage: prevents *unnecessary* integrity/confidentiality risks
  - accepting risks only when necessary
- Disadvantage: availability risk
  - Mitigation by incident respons → bug fix

Default Deny

or principle of least privilege

- Default Deny is a General Principle with many Applications
  - Access Control
  - Input Validation
  - Feature Selection
- Advantage: prevents *unnecessary* integrity/confidentiality risks
  - accepting risks only when necessary
- Disadvantage: availability risk
  - Mitigation by incident respons → bug fix

Input Checking

Default deny

- Defining harmful inputs is hard
- Defining correct input is easier
- Default deny will reject the input when in doubt
- Note that the SQL example,
  - the input is both valid and harmful
  - that's why you need sanitisation as well
- You can overdo it
  - many webpages validate email addresses
  - and reject the plus sign (+)
- The plus sign is valid according to the RFC
  - and has a very important function in non-MS mail servers

Input Checking

Default deny

- Defining harmful inputs is hard
- Defining correct input is easier
- Default deny will reject the input when in doubt
- Note that the SQL example,
  - the input is both valid and harmful
  - that's why you need sanitisation as well
- You can overdo it
  - many webpages validate email addresses
  - and reject the plus sign (+)
- The plus sign is valid according to the RFC
  - and has a very important function in non-MS mail servers

Input Checking

Default deny

- Defining harmful inputs is hard
- Defining correct input is easier
- Default deny will reject the input when in doubt
- Note that the SQL example,
  - the input is both valid and harmful
  - that's why you need sanitisation as well
- You can overdo it
  - many webpages validate email addresses
  - and reject the plus sign (+)
- The plus sign is valid according to the RFC
  - and has a very important function in non-MS mail servers

Input Checking
Default deny

- Defining harmful inputs is hard
- Defining correct input is easier
- Default deny will reject the input when in doubt
- Note that the SQL example,
  - the input is both valid and harmful
  - that's why you need sanitisation as well
- You can overdo it
  - many webpages validate email addresses
  - and reject the plus sign (+)
- The plus sign is valid according to the RFC
  - and has a very important function in non-MS mail servers

Example: path names

Default deny in input validation

- Suppose you write an application, where users upload files
  - The user can specify a filename, e.g. holiday.jpg,
  - ... and you prepend a directory name, e.g. /public/images/
- How can this be exploited?
- Suppose the users use filename /../../etc/passwd.
- How do we avoid this?
- Input checking is possible;
  - ../ is an illegal substring.

UNIVERSITY OF SURREY

Example: path names

Default deny in input validation

- Suppose you write an application, where users upload files
  - The user can specify a filename, e.g. holiday.jpg,
  - ... and you prepend a directory name, e.g. /public/images/
- How can this be exploited?
- Suppose the users use filename /../../etc/passwd.
- How do we avoid this?
- Input checking is possible;
  - ../ is an illegal substring.

Example: path names

Default deny in input validation

- Suppose you write an application, where users upload files
  - The user can specify a filename, e.g. holiday.jpg,
  - ... and you prepend a directory name, e.g. /public/images/
- How can this be exploited?
- Suppose the users use filename /../../etc/passwd.
- How do we avoid this?
- Input checking is possible;
  - ../ is an illegal substring.

Character Encoding

Vulnerabilities in Unicode

- Unicode collects characters for (almost) every language
- UTF-8 is the most common encoding of Unicode
- Variable length characters
  - ASCII (American 7-bit character set) uses one byte
    - Ensuring compatibility.
  - Western European (non-ASCII) characters use two bytes
  - More exotic characters require 3 or 4 bytes

Unicode encoding

- Each byte has a prefix
  - 0 – one-byte character
  - 110 – first byte of two-byte character
  - 1110 – first byte of three-byte character
  - 11110 – first byte of four-byte character
  - 10 – second or later byte of multi-byte character
- Remaining bits contain a unicode character number
  - 1 byte : 7 bits
  - 2 bytes : 11 bits (5+6)
  - 3 bytes : 16 bits (4+6+6)
  - 4 bytes : 21 bits (3+6+6+6)
- Only shortest possible representation is legal
  - but illegal representations are often accepted

Exploiting it

- Your application bans filenames containing ../
- But there are many ways to write /
  - / is Unicode 0010 1111
    - 1 byte : 0010 1111
    - 2 byte : 1100 0000 1010 1111
    - 3 byte : 1110 0000 1000 0000 1010 1111
    - 4 byte : 1111 0000 1000 0000 1000 0000 1010 1111
- So if your system accepts multi-byte forms,
- ... your input checking has to ban all representations of /.
- Default deny makes it easier
  - Accept only the canonical form

Exploiting it

- Your application bans filenames containing ../
- But there are many ways to write /
  - / is Unicode 0010 1111
    - 1 byte : 0010 1111
    - 2 byte : 1100 0000 1010 1111
    - 3 byte : 1110 0000 1000 0000 1010 1111
    - 4 byte : 1111 0000 1000 0000 1000 0000 1010 1111
- So if your system accepts multi-byte forms,
- ... your input checking has to ban all representations of /.
- Default deny makes it easier
  - Accept only the canonical form

Canonical Representation

- UTF-8 is an example of the use of canonical representations
- Several equivalent forms are *defined*
- Only the shortest form is *canonical*
- Before a safe comparison can be made
- . . . data should be converted into canonical form

Example: Napster filenames

- Napster was ordered by court to block certain songs
- Solutions
  - filter downloads based on filename
- Napster users by-passed this control
  - using equivalent (variations of) the song titles
- Almost impossible to control
  - title equivalence is defined by the users...
- Blatant breakdown of *‘Default Permit’*

Outline

Conclusions

- Secure coding is an essential part of software development
  - relatively new field
- The Top 25 Vulnerabilities database is a good source
  - avoid the Top 5 and you will be better than average ...
  - the list is updated regularly — check the latest version
- Practices may vary between languages
  - try to look up a book for whatever language you use