Lecture 6: Security trade-offs for distributed computing

Lecture topics
- Distributed computing and security
  - CGI-bin
  - Common Web application vulnerabilities
    - Not EJB-specific
  - Contrast direct access of distributed objects by clients with browser-based clients

Motivation
- Distributed computing is everywhere
- There are different ways of making distributed components communicate
  - Choice of enabling technologies
    - Protocols
    - Programming languages
    - Scripting languages
    - Platforms, Oses
  - Choice of encryption technologies/algorithms
  - Selecting the available functionality
  - Choice of what to send across network
    - Raw data
    - Formatted (partial data)
    - Views of data (e.g. HTML pages)

Web application model
- Send HTML pages
  - Views of data, not data
- Client-side scripting may be used

Distributed object model
- Send actual objects
  - Data, not views of data

Mixed model
- In some cases, objects are sent
- In other cases, views of data are sent

Brief overview of CGI (Common Gateway Interface)
- CGI is a standard for interfacing applications with HTTP servers
- CGI scripts typically generate Web pages that are served by servers
  - E.g. a script may be given a client query, access a database with this query, then generate a Web page with the results
- Typically, CGI scripts are placed in a directory called cgi-bin
  - The Web server will know to execute files from this directory, rather than displaying them
  - The directory is under control of Webmaster, preventing placement of unauthorized scripts there
  - Instead, could just give scripts the extension .cgi, but this is not recommended
  - Possibility of attackers viewing source of backups
- CGI programs can be written in any language that the system can execute
  - Yes, even Fortran
  - Perl is widely used
All input to Web programs is insecure

- Typically obtained from HTML forms
  - Clients can enter any data, ignoring expected format
- All of the following are wrong assumptions
  - If I create a selection list, the input for that field will be one of the option choices
  - If I set the maximum length of the input field then the browser will send at most that many characters for that field
  - The fields in the QUERY_STRING variable will match the ones in my page
  - The QUERY_STRING variable will correspond to something that could be validly transmitted by the HTTP specifications

Untrusted user input

- Have to assume that the user can enter anything
- Sources of user input for a script include:
  - The parameter string portion of the request URL
  - Data submitted by HTML forms via POST and GET requests
  - Cookies (data stored in the client browser)
  - Queries to databases
  - Environment variables set by other processes

Untrusted user input, again

- Let the servlet be:
  ```
  <%@ page import="java.sql.*" %>
  <!-- some code here to open a SQL connection -->
  <%
  Statement stmt = connection.createStatement();
  String query = "SELECT * FROM USER_RECORDS WHERE USER = " + request.getParameter("username");
  ResultSet result = stmt.executeQuery(query);
  %>
  ```
- Set username to username=joe;SELECT%20*%20FROM%20SYSTEM_RECORDS
- Some SQL servers will execute both queries
- Solution: input validation

Input validation

- Syntactic and semantic checks on data derived from untrusted sources
- Positive input filtering: use input only if it looks right
- Negative input filtering: throw input away if it looks wrong

Calling programs from CGI programs

- Often called programs are standard (UNIX) utilities
- E.g. in Perl (exp is a search expression that comes from a client, database is the file to be searched): system("grep $exp database");
  - If $exp has value "root /etc/passwd:rm"
    - The superuser password is returned, then the database file is deleted
- One solution is to escape all special characters (put "\" in front of them)
- Could use a different form of system:
  ```
  system("grep", "-e", $exp, "database");
  ```
  - Shell is never called in this case
Relying on PATH variable is dangerous

- Attackers may alter the PATH variable so that it points to the program they want your script to execute
  - `putenv("PATH=/home/compromised_user/bin");`
- A call
  - `system("ls -l /local/web/data");`
  - Will not execute ls from /bin or /usr/bin

All right, we get it, accessing shell is bad. So, we just don’t do it.
- Consider:
  - `open(MANPAGE, "/usr/man/man1/$filename.1");`
- An attacker can submit the following string for $filename:
  - `"../../etc/passwd"`

Other potential problems

- Writing CGIs in shell languages
  - Variable substitution is tricky
  - Different behavior with different quotation symbols
- Sending mail
  - Sending untrusted content
  - Sending to untrusted email addresses
- Redirecting HTTP requests
  - E.g. a program that accepts a URL and fetches one of the URLs listed on the page
  - May let attackers get around access control rules
- Dangers of writing CGI programs in C
  - Buffer overflows, string termination, etc.
- PERL is one big piece of rope, more than enough to hang yourself with
  - E.g. `eval $x --- x is interpreted as string of Perl code`
  - Modifying the grep example: `while (<FILE>) {print if /bash/} --- if $exp is an incorrectly formed regular expression, will get a compilation error -- server might report it as a server configuration error`

Sensitive data in GET requests

- Input data is appended to the request URL and represented in the form:
  - `URL[?name=value[&name=value[&...]]]`
- Should not be used for sending sensitive data
  - Sent in cleartext
  - Gets logged on the server and any intermediate routers
  - Instead, POST should be used, with encryption mechanisms (e.g. SSL connections)

Cookies

- Small piece of information the server puts on the client
  - Used to maintain session state information
- Security problems arise when sensitive information is stored in cookies
  - The whole content of the cookie is visible to the client
  - Nothing prevents the user from responding with a forged cookie
  - Browsers work hard to prevent this
- Any information submitted by the client browser may be malicious

Cross site scripting

- Malicious HTML tags embedded in client Web requests
  - Malicious user submits client-side scripts (e.g. JavaScript) or malicious HTML or XML tags
  - The JSP server includes these in dynamically generated pages
- Example: some discussion group servers allow participants to include tags in their posts
- Solution: input validation on the server side
URL probing attacks

- Complex Web applications interact with users over a number of HTTP requests
  - Problem: HTTP is a stateless protocol
  - Every browser request -> server response interaction is handled separately
- URL probing attack: jump to a page that is supposed to be accessed through a specific sequence of pages
  - The application may be in an inconsistent state
  - E.g., access to content has to be preceded by registration
- A simple solution:
  ```jsp
  <jsp:include page="verifyinclude.jsp" flush="true" />
  ```
  The source of `verifyinclude.jsp` is as follows:
  ```java
  <%
  String smName = (String) request.getAttribute("smName");
  if(smName == null)
    throw new Exception("Page not called from within application.");
  %>
  ```

Distributed object programming — a different ballgame

- With CGIs and servlets, only one type of data is sent from clients to servers --- strings
- With distributed objects, whole objects get sent from clients to servers and back, via remote method calls
- So which model is easier to deal with from the point of view of security?
  - On the one hand, the distributed objects model is very rich, with the possibility for unbounded complexity
  - On the other hand, rich data can be encoded as a string
    - Error prone because a lot of parsing has to be done
- The two models are intended to be used in different situations: thin clients vs fat clients

Distributed object model — two kinds of shared objects

- **Distributed objects** are on the server
  - Clients can call their methods
  - Clients cannot obtain copies of distributed objects (and their fields) directly
    - So, a private field remains private unless it is returned via a remote method
- **Data objects** are sent between server and clients
  - Arguments in method calls
  - Returns from method calls
  - Copies of an object can exist on the server and clients at the same time
    - Potential need for data coherence: when one copy changes, the changes should be propagated to other copies of the object

A simple example from bookauction

- Data object type `Auction` represents auctions
- An auction can have a number of bids (type `Bid`)
  - Stored in a field inside of an `Auction` object
- Clients request the server for auctions
- A client can enter a new bid for an auction
- The server and clients should be notified of that
- Illustrates that data coherence requires additional steps in distributed applications
  - In non-distributed programs, the notification step may not be necessary, if references are used

It gets worse when data objects can reference each other

- If one data object `A` references another object `B`, whenever `A` has to be sent from server to a client, chances are `B` has to be sent at the same time
  - The alternative is to match `B` to another existing object `B'` of the same type on the client
    - This is rarely feasible, so `B'` is replaced by a copy of `B`
- But what if `B'` is referenced by another object, `C`?

Open Web Application Security Project (OWASP)

- owasp.org
- Dedicated to advancing knowledge about web application security
- Maintains Web application security vulnerability top ten list
  - Not just names, but explanations, examples, links to information
- WebGoat
  - An application that illustrates common security flaws
  - You can actually experiment with things like SQL injection
OWASP top ten Web application security vulnerabilities for 2004
- Unvalidated input
- Broken access control
- Broken authentication and session management
- Cross-site scripting (XSS) flaws
- Buffer overflows
- Injection flaws
- Improper error handling
- Insecure storage
- Denial of service
- Insecure configuration management

Guiding principles for software security
1. Secure the weakest link
2. Practice defense in depth
3. Fail securely
4. Follow the principle of least privilege
5. Compartmentalize
6. Keep it simple
7. Promote privacy
8. Remember that hiding secrets is hard
9. Be reluctant to trust
10. Use your community resources

Remember that all these involve trade-offs, such as complexity and usability

Do any of these principles favor Web applications over distributed applications or vice versa?