

Secure Systems Design

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



Agenda

- Understanding Threats
- “Designing-In” Security (*)
- Convenience and Security
- Security By Obscurity
- Open vs. Closed Source
- A Game of Economics

2.1. Understanding Threats

- ID & Mitigate Threats

- ☐ Defacement
- ☐ Infiltration
- ☐ Phishing
- ☐ Pharming
- ☐ Insider Threats
- ☐ Click Fraud
- ☐ Denial of Service
- ☐ Data Theft/Loss

2.1.1. Defacement

- Online Vandalism, attackers replace legitimate pages with illegitimate ones
- Targeted towards political web sites
- Ex: White House website defaced by anti-NATO activists, Chinese hackers

2.1.2. Infiltration

- Unauthorized parties gain access to resources of computer system (e.g. CPUs, disk, network bandwidth)
- Could gain read/write access to back-end DB
- Ensure that attacker's writes can be detected
- Different goals for different organizations
 - Political site only needs integrity of data
 - Financial site needs integrity & confidentiality

2.1.3. Phishing

- Attacker sets up spoofed site that looks real
 - Lures users to enter login credentials and stores them
 - Usually sent through an e-mail with link to spoofed site asking users to "verify" their account info
 - The links might be disguised through the click texts
 - Wary users can see actual URL if they hover over link

```
<a href="http://www.evil-site.com">  
  Legitimate Site  
</a>
```

Legitimate Site

<http://www.evil-site.com/>

2.1.4. Pharming

- Like phishing, attacker's goal is to get user to enter sensitive data into spoofed website
- *DNS Cache Poisoning* – attacker is able to redirect legitimate URL to their spoofed site
- DNS translates URL to appropriate IP address
- Attacker makes DNS translate legitimate URL to their IP address instead and the result gets cached, poisoning future replies as well


2.1.5. Insider Threats

- Attacks carried out with cooperation of insiders
 - Insiders could have access to data and leak it
 - Ex: DB and Sys Admins usually get complete access
- *Separation of Privilege / Least Privilege Principle*
 - Provide individuals with only enough privileges needed to complete their tasks
 - Don't give unrestricted access to all data and resources



2.1.6. Click Fraud

- Targeted against pay-per-click ads
- Attacker could click on competitor's ads
 - Depletes other's ad budgets, gains exclusive attention of legitimate users
- Site publishers could click on ads to get revenue
- Automated through malware such as botnets



2.1.7. Denial of Service (DoS)

- Attacker inundates server with packets causing it to drop legitimate packets
 - Makes service unavailable, downtime = lost revenue
- Particularly a threat for financial and e-commerce vendors
- Can be automated through botnets

2.1.8. Data Theft and Data Loss

- Several Examples: BofA, ChoicePoint, VA
 - BofA: backup data tapes lost in transit
 - ChoicePoint: fraudsters queried DB for sensitive info
 - VA: employee took computer with personal info home & his home was burglarized
- CA laws require companies to disclose theft/loss
- Even for encrypted data, should store key in separate media

Threat Modeling

Application Type	Most Significant Threat?
Civil Liberties web site White House web site	Defacement
Financial Institution Electronic Commerce	Compromise one or more accounts; Denial-of-Service
Military Institution Electronic Commerce	Infiltration; access to classified data

Threat Modeling Frameworks

- STRIDE:

Spoofing Identity

Tampering (unauthorized modification)

Repudiation

Information Disclosure (unauthorized read)

Denial-of-Service

Escalation of privilege

(Source: MSFT)

2.2. Designing-In Security

- Design features with security in mind

- ☐ Not as an afterthought
- ☐ Hard to “add-on” security later

- Define concrete, measurable security goals. Ex:

- ☐ Only certain users should be able to do X. Log action.
- ☐ Output of feature Y should be encrypted.
- ☐ Feature Z should be available 99.9% of the time

- Bad Examples: Windows 98, Internet

2.2.1. Windows 98

- Diagnostic Mode:
 - Accessed through 'F8' key when booting
 - Can bypass password protections, giving attacker complete access to hard disks & data
- Username/Password Security was added as an afterthought
- Should have been included at the start, then required it for entering diagnostic mode

2.2.2. The Internet

- All nodes originally university or military (i.e. trusted) since it grew out of DARPA
- With commercialization, lots of new hosts, all allowed to connect to existing hosts regardless of whether they were trusted
- Deployed Firewalls: allows host to only let in trusted traffic
- Loopholes: lying about IPs, using cleared ports

IP Whitelisting & Spoofing

- *IP Whitelisting*: accepting communications only from hosts with certain IP addresses
- *IP Spoofing attack*: attacker mislabels (i.e. lies) source address on packets, slips past firewall
- Response to spoofing sent to host, not attacker
 - Multiple communication rounds makes attack harder
 - May DoS against legitimate host to prevent response

IP Spoofing & Nonces

- *Nonce*: one-time pseudo-random number
- Attaching a nonce to a reply and requesting it to be echoed back can guard against IP spoofing
- Attacker won't know what reply to fake
- Spoofing easier for non-connection-oriented protocols (e.g. UDP) than connection-oriented (e.g. TCP)
- TCP sequence #s should be random, o/w attacker can predict and inject packets into conversation

2.2.3. Turtle Shell Architectures

- Inherently insecure system protected by another system mediating access to it
 - Ex: Firewalls guard vulnerable systems within
 - Ex: Death Star “strong outer defense” but vulnerable
- Hard outer shell should not be sole defense

2.3. Convenience and Security

- Sometimes inversely proportional
 - More secure → Less convenient
 - Too Convenient → Less secure
- If too inconvenient → unusable → users will workaround → insecure
- Ex: users may write down passwords
- Good technologies increase both: relative security benefit at only slight inconvenience



2.5. Security in Software Requirements

- Robust, consistent error handling
- Share reqs w/ QA team
- Handle internal errors securely – don't provide error messages to potential attackers!
- Use “defensive programming”
- Validation and Fraud Checks
- “Security or Bust” Policy



2.4. Simple Web Server (SWS)

- To illustrate what can go wrong if we do not design for security in our web applications from the start, consider a simple web server implemented in Java.
- Only serves documents using HTTP
- Walkthrough the code in the following slides

2.4.1. Hypertext Transfer Protocol (1)

- HTTP is the communications protocol used to connect to servers on the Web
- Primary function is to establish a connection with a server & transmit HTML pages to client browsers or any other files required by an HTTP application.
- Website addresses begin with an **http://** prefix.

2.4.1. HTTP (2)

- A typical HTTP request that a browser makes to a web server:

`GET / HTTP/1.0`

- When the server receives this request for filename / (the *root* document on the web server), it attempts to load *index.html*. returns

`HTTP/1.0 200 OK`

followed by the document contents.

2.4.2. SWS: main

```
/* This method is called when the program is run from
the command line. */

public static void main (String argv[]) throws Exception
{
    /* Create a SimpleWebServer object, and run it */
    SimpleWebServer sws = new SimpleWebServer();
    sws.run();
}
```

2.4.2. SimpleWebServer Object

```
public class SimpleWebServer {

    /* Run the HTTP server on this TCP port. */
    private static final int PORT = 8080;

    /* The socket used to process incoming connections
    from web clients */
    private static ServerSocket dServerSocket;

    public SimpleWebServer () throws Exception {
        dServerSocket = new ServerSocket (PORT);
    }

    public void run() throws Exception {
        while (true) {
            /* wait for a connection from a client */
            Socket s = dServerSocket.accept();

            /* then process the client's request */
            processRequest(s);
        }
    }
}
```

2.4.2. SWS: processRequest (1)

```
/* Reads the HTTP request from the client, and
   responds with the file the user requested or
   a HTTP error code. */

public void processRequest(Socket s) throws Exception {

    /* used to read data from the client */
    BufferedReader br =
        new BufferedReader (new InputStreamReader (s.getInputStream()));

    /* used to write data to the client */
    OutputStreamWriter osw =
        new OutputStreamWriter (s.getOutputStream());

    /* read the HTTP request from the client */
    String request = br.readLine();

    String command = null;
    String pathname = null;
```

2.4.2. SWS: processRequest (2)

```
/* parse the HTTP request */
StringTokenizer st =
    new StringTokenizer (request, " ");

command = st.nextToken();
pathname = st.nextToken();

if (command.equals("GET")) {
    /* if the request is a GET
       try to respond with the file
       the user is requesting */
    serveFile (osw,pathname);
}
else {
    /* if the request is a NOT a GET,
       return an error saying this server
       does not implement the requested command */
    osw.write ("HTTP/1.0 501 Not Implemented\n\n");
}

/* close the connection to the client */
osw.close();
```

2.4.2. SWS: serveFile (1)

```
public void serveFile (OutputStreamWriter osw,
                      String pathname) throws Exception {
    FileReader fr=null;
    int c=-1;
    StringBuffer sb = new StringBuffer();

    /* remove the initial slash at the beginning
       of the pathname in the request */
    if (pathname.charAt(0)=='/')
        pathname=pathname.substring(1);

    /* if there was no filename specified by the
       client, serve the "index.html" file */
    if (pathname.equals(""))
        pathname="index.html";
```

2.4.2. SWS: serveFile (2)

```
/* try to open file specified by pathname */
try {
    fr = new FileReader (pathname);
    c = fr.read();
}
catch (Exception e) {
    /* if the file is not found, return the
       appropriate HTTP response code */
    osw.write ("HTTP/1.0 404 Not Found\n\n");
    return;
}
```

2.4.2. SWS: serveFile (3)

```
/* if the requested file can be
successfully opened and read, then
return an OK response code and send
the contents of the file */
osw.write ("HTTP/1.0 200 OK\n\n");
while (c != -1) {
    sb.append((char)c);
    c = fr.read();
}
osw.write (sb.toString());
```

2.5.1. Specifying Error Handling Requirements

- Vulnerabilities often due to bad error handling
- Example: DoS on SWS – makes it unavailable
- *Just send a carriage return as the first message instead of a properly formatted GET message...*
- Causes exception when breaking into tokens

2.5.1. DoS on SWS Example

processRequest():

```
/* read the HTTP request from the client */
String request = br.readLine(); // empty string

String command = null;
String pathname = null;

/* parse the HTTP request */
StringTokenizer st =
    new StringTokenizer (request, " ");

command = st.nextToken(); // EXCEPTION: no tokens!
/* SERVER CRASHES HERE - DENIAL OF SERVICE! */
pathname = st.nextToken();
```

2.5.1. How Do We Fix This?

- *The web server should immediately disconnect from any web client that sends a malformed HTTP request to the server.*
- The programmer needs to carefully handle exceptions to deal with malformed requests.
- Solution: Surround susceptible String Tokenizing code with try/catch block.

2.5.1. Try/Catch Solution

```
/* read the HTTP request from the client */
String request = br.readLine();
String command = null;
String pathname = null;

try {
    /* parse the HTTP request */
    StringTokenizer st =
        new StringTokenizer (request, " ");
    command = st.nextToken();
    pathname = st.nextToken();
} catch (Exception e) {
    osw.write ("HTTP/1.0 400 Bad Request\n\n");
    osw.close();
    return;
}
```

2.5.2. Sharing Requirements with Quality Assurance (QA)

- Both dev & testers should get requirements
- Should have test cases for security too: Does it malfunction when provided bad input?
- *Ping-of-Death*: sending a packet of data can cause server to crash
 - Ex: DoS attack on SimpleWebServer
 - Ex: Nokia GGSN crashes on packet with TCP option field set to 0xFF

2.5.3. Handling Internal Errors Securely

- Error messages and observable behavior can tip off an attacker to vulnerabilities
- *Fault Injection*: Providing a program with input that it does not expect (as in the DoS attack against SimpleWebServer) and observing its behavior
- “Ethical” hackers often hired to find such bugs

2.5.4. Including Validation and Fraud Checks

- Requirements should specify which error cases & threats to handle
- Credit Card Example:
 - *Mod 10 Checksum*: ensures validity of number, to catch user typos
 - CVC: guards against fraudsters who have stolen # but don't know the CVC
 - Full Credit Card Check might be too costly

2.5.5. Writing Measurable Security Requirements

- Access Control Security: Only certain users can do certain functions
- Auditing: Maintain log of users' sensitive actions
- Confidentiality: encrypt certain functions' output
- Availability: Certain features should be available almost always
- Include these requirements in design docs!

2.5.6. Security or Bust

- Should not ship code unless its secure
- Advantage gained by launching earlier could be lost due to vulnerabilities that tarnish brand and lead to lost revenue
- Ex: Microsoft delayed ship of .NET server in '02 because security requirements not met by "code freeze" deadline

2.6. Security by Obscurity

- Trying to be secure by hiding how systems and products work (to prevent info from being used by attacker)
- Ex: Military uses Need to Know basis
- Maybe necessary, but not sufficient to prevent determined attackers

2.6.1. Flaws in the Approach

- What assumptions to make about adversary?
 - Knows algorithms? Or not?
 - Algorithms in “binary” secret?
- Attackers can probe for weaknesses
 - reverse engineer exes
 - observe behavior in normal vs. aberrant conds. (use *fault injection*)
 - *Fuzzing*: systematically trying different input strings to find an exploit
 - blackmail insiders

Secret Keys

- *Kerckhoffs' doctrine* (1883): “The method used to encipher data is known to the opponent, and that security must lie in the choice of key.”
 - assume the worst case!
 - obscurity alone is not sufficient
- Compromised key can be changed without re-designing system.
- Key is easier to keep secret

2.6.2. SWS Obscurity

- Just distributing Java bytecode of SWS (and not source code) not enough security
- Can be disassembled or decompiled (e.g. Mocha, Jad) to produce rough source code
- Even disassembling can reveal the DoS exploit of the vulnerable tokenization process

2.6.2. Disassembling SWS

```
public void processRequest(java.net.Socket); 43: new 34; //class StringTokenizer
throws java/lang/Exception                  46: dup
Code:                                         47: aload 4
0: new 25; //class BufferedReader           49: ldc 35; //String
3: dup                                       51: invokespecial 36;
4: new 26; //class InputStreamReader        54: astore 7
7: dup                                       56: aload 7
8: aload_1                                   58: invokevirtual 37;
9: invokevirtual 27;                        61: astore 5
12: invokespecial 28;                       63: aload 7
15: invokespecial 29;                       65: invokevirtual 37;
18: astore_2                                68: astore 6
19: new 30; //class OutputStreamWriter      70: aload 5
22: dup                                       72: ldc 38; //String GET
23: aload_1                                   74: invokevirtual 39;
24: invokevirtual 31;                       77: ifeq 90
27: invokespecial 32;                       80: aload_0
30: astore_3                                81: aload_3
31: aload_2                                   82: aload 6
32: invokevirtual 33;                       84: invokevirtual 40;
35: astore 4                                87: goto
37: aconst_null                             99: astore 8
38: astore 5                                101: aload_3
40: aconst_null                             102: invokevirtual 44;
41: astore 6                                105: return
                                           93: invokevirtual 42;
                                           96: goto 101
```

2.6.3. Things to Avoid

- Don't "invent" your own encryption algorithm!
- Don't embed keys in software!
- Nor in Windows Registry which is readable by all
- Don't Forget Code Reuse: reuse well-tested software known to be reliably secure instead of doing same thing from scratch

2.7. Open vs. Closed Source

- “Is open-source software secure?”
- Open:
 - Some people might look at security of your application (if they care)
 - may or may not tell you what they find
- Closed:
 - not making code available does not hide much
 - need diverse security-aware code reviews
- A business decision: Not a security one!

2.8. A Game of Economics

- All systems insecure: how insecure?
- What is the cost to break system? Weakest link?
- For every \$ that defender spends, how many \$ does attacker have to spend?
- If (Cost to “break” system >> Reward to be gained)
 - Then system is secure
 - Otherwise system is NOT secure
- “Raise the bar” high enough
- Security is about **risk management**

2.8. Economics Example

- Two ways to break system with L -bit key
 - Brute-force search for key: costs C cents/try
 - “Payoff” employee (earning S yearly for Y years, interest α) for the key: costs $P = \sum_{i=0}^Y S\alpha^{Y-i}$ dollars
- Brute-Force Total Cost:
 - On average, try half the keys
 - $\text{Cost} = (C/2)(2^L) = 2^{L-1}C$
- Ex: Say $P = \$5$ million, $L = 64$, $C = 3.4e-11$, brute-force cost is $> \$300$ million (better to payoff)
- Break-even point: $2^{L-1}C = \sum_{i=0}^Y S\alpha^{Y-i}$

2.9. “Good Enough” Security

- Alpha Version: security should be good enough
 - Won't have much to protect yet
 - Difficult to predict types of threats
 - But still set up a basic security framework, “hooks”
- Beta Version: throw away alpha
- Design in security to deal with threats discovered during testing



Summary

- Threats (DoS, Phishing, Infiltration, Fraud, ...)
- SimpleWebServer: Security by Obscurity Fails
- Economics Game (cost >> reward for attacker)
- “Good Enough” Security: Design Incrementally From Beginning