Developing Secure Things

Ways to not go as far wrong as you might...
Some Materials

OWASP
- https://www.owasp.org/index.php/Main_Page

SANS
- https://www.sans.org/security-resources/

Flawfinder and more
- https://www.dwheeler.com
Next hour

- Engineering approach
- Buffer overruns
- Defensive Programming
- Education
Security Engineering

- Networked applications need to be secure
  - not only security software needs to be secure
- Security is a dynamic property that changes with the environment
- Security cannot easily be added to an existing system
  - like safety, dependability, reliability, ...
  - not just triggered by faults, but by active opponents
- Security is a behavioral property of a complete system in a particular environment
Penetrate and Patch is Bad

- Most developers worry about security once their program is compromised - issue patch

- Problems with penetrate and patch:
  - Only known problems can be patched (the bad guy may never report the problem)
  - Patches often go unapplied
  - Patches are rushed out and may introduce new bugs
  - Patches only fix the symptom
Proactive Security Approach

- Security must be designed into the application from the beginning
- Five-step process:
  1) Design software with security in mind
  2) Analyze the system with respect to anticipated risks
  3) Rank risks according to severity
  4) Test for risks
  5) Cycle a defect system through the design process
- Defects should be found and fixed before the software is released
Design for Security

• Design with the intended environment in mind
  – don’t rework centralized applications to be used on the Internet - too many assumptions may fail

• Define a threat model - explicitly state what security problems are addressed

• Document security decisions
  – don’t state that “the program encrypts necessary data”
  – state why encryption is used
  – state where encryption is used
Analyze the System

• Highly specialized task (special “tiger teams”)

• Never analyze your own code (external analysis)
  – biased view on code - likely to miss mistakes
  – use security conscious programmers (colleagues)

• Start with the written specification
  – identify risks
Rank Risks

• Consider environment and threat model
  – is this attack probable or stopped by firewall?
  – what are the consequences?
    • DoS is not serious for clients but fatal for web-servers
• Templates or standard lists of known risks are often useful to consider
Test for Risks

• Use the ranked list of risks to direct testing
  – create test cases to reveal problems or “interesting” behavior
  – inspect code to determine extent of problem

• Security testing includes abusing the program
  – don’t just use it as intended
  – feed it strange input (malformed data, long lines)

• Code coverage should be complete
  – “dead code” is a potential stable for a Trojan horse
Redesign if System Fails

- Examine sources if the system fails the tests
- Correct simple errors:
  - buffer overflows are simply solved
    (build prevention/detection into your development processes)
- Complex errors must result in a redesign
  - this will delay the release of the program, but it is necessary
Buffer Overflows

- Very common vulnerability, more than 50% of incidents reported to CERT
- A parameter contains more data than an internal buffer can handle
- The buffer overflows and overwrites other variables allocated in nearby memory

This problem is solved in type-safe languages like Java
How do Buffer Overflows Work

Application memory

<table>
<thead>
<tr>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>start of buffer</td>
</tr>
<tr>
<td>end of buffer</td>
</tr>
<tr>
<td>variables</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>parameters</td>
</tr>
<tr>
<td>rest of stack</td>
</tr>
</tbody>
</table>

1) Calculate distance from buffer to “return address”
2) Find room for attack code
3) Overflow buffer
   a) place attack code on stack
   b) overwrite return address
Principles of Robust Programming

• Code must handle bad input reasonably
• Controlled termination on internal errors
• 4 Principles of Robust Programming
  – Paranoia
  – Stupidity
  – Dangerous Implements
  – Can’t Happen

[Matt Bishop]
Paranoia

- Don’t trust anything you don’t generate
- Whenever someone calls your library, assume they will try to break it
- Always check return codes of function calls
- Assume you make mistakes, program defensively so they will be discovered quickly
Stupidity

• Assume the caller or user is an idiot, who cannot read the manual
• Write code that handles incorrect, bogus and malformed input
  – error messages must be self-explanatory
• If you detect problems, correct the error immediately or stop, to prevent error propagation
Dangerous Implements

• A dangerous implement is anything that your code assumes to remain fixed across function calls, e.g. file pointers in I/O calls

• Never give direct access to dangerous implements - use tokens instead of pointers

• Hiding data structures also makes your program more modular
Can’t Happen

• Impossible cases normally are not, just highly improbable
• Implement code that handles impossible cases
  – check impossible cases and print an error message if they occur
• “Robust Programming” is defensive
  – protect your program from users
  – protect your program from yourself

Beware of everything - even your own code!
Education

• We know we should include consideration of security at various stages
• Will only happen when relevant people have bought into that concept
• Who?
  – Management, Senior Developers/Designers, Customers
• Use a mixture of arguments:
  – Frightening, like-insurance, enabler, customer requirement
Modern Dev. Processes

• Agile processes tend to prioritise running code over design effort (no more waterfalls:-)
  – Good that there's less theoretical stuff, less good that there's less thought
• Move away from traditional Quality Assurance (QA) teams (test teams) towards DevOps
    • It's wikipedia but good enough:-)
  – Yahoo killing QA teams => fewer errors!
• What are the security consequences?
  – Less separation/isolation between developers and those with operational access (bad, private keys in repos)
  – More involvement of those with operations clue in development (good, real threats/mitigations more obvious)
Testing...

- Memory leaks are bad: valgrind is a fine tool
- Fuzzing is a fine way to break things to make them better
- Lesson: seek out the current good tools for your development environment and use those to test your code/systems
How much is enough?

- When is a heavyweight or lightweight process better?
- Say for s/w product development
  - Heavy: ITSEC/CC; Formal internal reviews; tiger-teams
  - Light: Internal reviews; occasional external consultants; learn-as-you-go
- What should *always* be done?