Chapter 3

2 + 2 = 5

Program Security

Content

• Attack: Program Flaw in Spacecraft Software
• Threat: Program Flaw Leads to Failed Security
• Vulnerability: Incomplete Mediation
• Vulnerability: Race Condition
• Vulnerability: Time-of-Check to Time-of-Use
• Vulnerability: Undocumented Access Point
• Ineffective Countermeasure: Penetrate & Patch
Content

- Countermeasure: Identify + Classify Faults
- Countermeasure: Secure Software Design
- Countermeasure: Secure Software Development Process
- Countermeasure: Testing
- Countermeasure: Defensive Programming
- Conclusion

Introduction

- Early computer users had to be programmers themselves
- Modern computer users not
- Off-the-shelf software is readily available
- How do you now know if there is no flaw in the program?
- Fault can be:
  - Intentional, on integrity
  - Unintentional, however, part of an integrity attack
Attack: Program Flaw in Spacecraft Software

- One data item being overwrited incorrectly, causes a sequence of events…

FIGURE 3-1 Cascade of Errors Dooms Mars Surveyor
Threat: Program Flaw leads to Security Failing

• When performing threat analysis, you should think of each possible outcome not just as a final result but also as a possible stepping stone to exploiting other vulnerabilities.

• For example, common programming flaws:
  – exceeding the bounds of an array
  – calculating subscripts from 0 instead of 1
  – performing arithmetic on a pointer instead of data
  – storing a string value in a numeric location. In the hands of an attacker, each of these problems can become the first link in a chain of consequences.

Sidebar 3.1

• The tree swing…

  what marketing suggested

  what management approved
Sidebar 3.1

• The tree swing…

as designed by engineering

what was manufactured

Sidebar 3.1

• The tree swing…

as maintenance installed it

what the customer wanted
Threat: Program Flaw leads to Security Failing

- Many of these very fixable errors occur because programmers can be too optimistic
- However, we cannot possibly catalog all possible programming faults

Vulnerability: Incomplete Mediation

- Mediation means checking: the process of intervening to confirm an actor’s authorization before it takes an intended action
- In the last chapter, steps in the authentication process studied to confirm a subject’s identity
- Verifying that subject is authorized to perform the operation on an object is called mediation
- Mediation implements the access control triple that describes what subject can perform what operation on what object.
Vulnerability: Incomplete Mediation

• What is wrong with this URL?


Vulnerability: Incomplete Mediation

• What would happen if parm2 is submitted as
  – 1800Jan01? or
  – 1800Feb30? or
  – 2048Min32? or
  – 1Aardvark2Many?

• Something will likely fail
  – Program crash
  – Or generate a wrong result
  – Or use a default e.g. 1 January 1970

• How do we get rid of these problems?
Vulnerability: Incomplete Mediation

• We anticipate them!
• On client side we can:
  – Limit the options (drop-down box)
  – Use other legitimate options

• Would the URL then be safe?
  – Can still change the content of the URL manually!
  – Server-side cannot tell if it was the user or the browser that entered that info!
  – Data values not mediated; they are exposed!

Vulnerability: Race condition

• Example: Two processes are competing within the same time interval, and the race affects the integrity or correctness of the computing tasks
• 2 processes competing for the same memory
• Better definition: In a race condition or serialization flaw two processes execute concurrently, and the outcome of the computation depends on the order in which instructions of the processes execute
• Airline reservation system…
Vulnerability: Race condition

- Cause inconsistent, undesired and therefore wrong, outcomes—a failure of integrity.

FIGURE 3-4 File Name Race Condition (a) Normal Operation (b) Overwriting Filename Other Than Original
Vulnerability: time-of-check to time-of-use (TOCTTOU)

- Stack of 5 x 20$ bills example

FIGURE 3-5 File Access Data Structure

<table>
<thead>
<tr>
<th>File:</th>
<th>Action:</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_file</td>
<td>Change byte 4 to A</td>
</tr>
</tbody>
</table>
Vulnerability: Undocumented access point

- Easter egg
  - Open a new MS Excel worksheet
  - Press F5
  - Type X97:L97 and press Enter
  - Press Tab
  - Hold <Ctrl-Shift> and click the Chart Wizard
Program security

- Attack?
  - Program flaw in spacecraft software

- Threat?
  - Program flaw leads to security failing

- Vulnerability?
  - Incomplete mediation
    - the process of intervening to confirm an actor’s authorization before it takes an intended action

- Error $\rightarrow$ Fault $\rightarrow$ Failure = Flaw

- Ineffective countermeasure: penetrate-and-patch
  - Tiger-team approach
  - Flaws:
    - Narrow focus on the fault, not its context
    - Causes side-effects somewhere else
    - Fix this, cause another “buffer” to break
    - Fixing it this way might cause performance issues

- Inadvertent human errors vs malicious, intentionally induced flaws
• We do not have techniques to eliminate or address all program security flaws
• Security is fundamentally hard
• Security often conflicts with usefulness and performance; no “silver bullet”
• False security solutions impede real progress toward secure programming

• There are two reasons for this distressing situation:
  – Program controls apply at the level of the individual program and programmer
    • Should-do vs Shouldn’t-do approach
  – Programming and software engineering techniques change and evolve far more rapidly than do computer security techniques
Inadvertent flaws

- Validation error (incomplete or inconsistent): permission checks
- Domain error: controlled access to data serialization: program flow order
- Inadequate identification and authentication: basis for authorization
- Boundary condition violation: failure on first or last case
- Many other exploitable logic errors

Countermeasure

- Secure software design elements
  - Modularity (small components)
  - Encapsulation
  - Information hiding
  - Mutual suspicion (human and computer trust)
  - Confinement (“sandbox”)
  - Simplicity
  - Generic diversity (heterogeneity)
Advantages to having small, independent components

- Maintenance
- Understandability
- Reuse
- Correctness
- Testing
- Should have high cohesion (focus) and low coupling (dependence on other modules)
FIGURE 3-8 Types of Coupling

FIGURE 3-9 Information Hiding
Design principles for Security

- Least privilege
- Economy of mechanism
- Open design
- Complete mediation
- Permission based
- Separation of privilege
- Least common mechanism
- Ease of use

Top 10 secure coding practices

1. Validate input.
2. Heed compiler warnings.
3. Architect and design for security policies.
4. Keep it simple.
5. Default to deny.
6. Adhere to the principle of least privilege.
Top 10 secure coding practices

7. Sanitize data sent to other systems.
8. Practice defense in depth.
9. Use effective quality assurance techniques.
10. Adopt a secure coding standard.

Countermeasure: Secure development process

- Specify the system
- Design the system
- Implement the system
- Test the system
- Review the system at various stages
- Document the system
- Manage the system
- Maintain the system
“Solid” software

- peer reviews
- hazard analysis
- good design
- prediction
- static analysis
- configuration management
- analysis of mistakes
Countermeasure: Testing

- Try to make the software error free
- Realistically, will not be able to test for everything
- Testing is hard!
- James Whittaker: “Developers grow trees; testers manage forests”
- The job of the tester is to explore the interplay of many factors.
Types of Testing

• Unit testing
• Integration testing
• Function testing
• Performance testing
• Acceptance testing
• Installation testing
• Regression testing
• Blackbox testing
• Clearbox testing

James Whittaker’s Testing Ingredients

• Product expertise
• Coverage
• Risk analysis
• Domain expertise
• Common vocabulary
• Validation
• Boundaries
Effectiveness of testing

• The mix of techniques appropriate for testing a given system depends on the system’s size, application domain, amount of risk, and many other factors
• But understanding the effectiveness of each technique helps us know what is right for each particular system

Effectiveness of testing

• The conclusions of testing is limited because:
  – Testing can demonstrate the existence of a problem, but not the absence of problems
  – Testing is difficult because of explosion of inputs and internal states
  – Testing only observable effects, does not ensure any degree of completeness
  – Testing is “artificial” and can itself be a source of vulnerabilities or can mask other vulnerabilities
  – Testing real-time systems makes it hard to reproduce and analyze problems reported
Testing especially for security

- Penetration testing (Tiger team analysis or ethical hacking)
- Proofs of program correctness; hard because:
  - Correctness proofs depend on programmers ability, just as programming is prone to errors, so is this.
  - The logical engine to generate proofs runs slowly; degrades as the size of the program increases.
  - Often people focus so much on formalism they ignore underlying security properties to be ensured.
  - Program verification is less developed than code production; not consistently applied to large systems.

Validation

- Formal verification is a particular instance of the more general approach to assuring correctness: verification.
- There are many ways to show that each of a system’s functions works correctly.
- Validation is the counterpart to verification, assuring that the system developers have implemented all requirements.
- Thus, validation makes sure that the developer is building the right product (to specification).
- Verification checks quality of implementation.
Validation

• A program can be validated in several different ways:
  – Requirements checking
  – Design and code reviews
  – System testing

Countermeasure: Defensive Programming

• Defenders have to counter all possible attacks, attackers have to find one weakness to exploit
• Strong defense is not only helpful, it’s essential
• Program designers and implementers need not only write correct code but must also anticipate what could go wrong
• As we pointed out earlier in this chapter, a program expecting a date as an input must also be able to handle incorrectly formed inputs such as 31-Nov-1929 and 42-Mpb-2030
Countermeasure: Defensive Programming

- Kinds of incorrect inputs include:
  - value inappropriate for data type, such as letters in a numeric field or M for a true/false item
  - value out of range for given use, such as a negative value for age or the date February 30
  - value unreasonable, e.g. 250 kg of salt in a recipe
  - value out of proportion, for example, a house description with 1 bedroom and 500 bathrooms
  - incorrect number of parameters
  - incorrect order of parameters

Countermeasure: Defensive Programming

- Options for action in event of incorrect input:
  - Stop, or signal an error condition and return.
  - Generate an error message and wait for user action.
  - Generate an error message and reinvoke the calling routine from the top
  - Try to correct it if the error is obvious
  - Continue, with a default or nominal value, or continue computation without the erroneous value
  - Do nothing, if the error is minor, superficial, and is certain not to cause further harm.
Conclusion

• The source of a flaw—benign or malicious—does not relate to its seriousness; unintentional failures can have important consequences.

• Some programmers are too optimistic, assuming that inputs are correct and properly formed, that testing will uncover all flaws, and that each correction is the last.

Conclusion

• Incomplete mediation is a situation of unchecked authorization. Security is upheld only if a system checks for authorization before allowing each access.

• A race condition occurs when one task can affect the security of another executing concurrently, depending on the order in which parts of the two tasks are performed.
Conclusion

• The programming flaw time-of-check to time-of-use can happen if the system loses control between the time an access permission is checked and the access occurs. Between the check and the access security-critical data must be protected against outside modification.

• See table 3-3