ECE/CS230
Computer Systems Security

Charalambos (Harrys) Konstantinou

https://sites.google.com/view/ececs230kaust/

Program security

The picture so far

- We've looked at a large number of ways an attacker can compromise program security
  - Exploit unintentional flaws
  - Introduce malicious code, including malware
  - Exploit intentional, but nonmalicious, behaviour of the system
- The picture looks pretty bleak
- Our job is to control these threats
  - It's a tough job

Software lifecycle

- Software goes through several stages in its lifecycle:
  - Specification
  - Design
  - Implementation
  - Change management
  - Code review
  - Testing
  - Documentation
  - Maintenance
- At which stage should security controls be considered?

Security controls – Design

- How can we design programs so that they're less likely to have security flaws?

1. Modularity
2. Encapsulation
3. Information hiding
4. Mutual suspicion
5. Confinement

1. Modularity

- Break the problem into a number of small pieces (“modules”), each responsible for a single subtask
- The complexity of each piece will be smaller, so each piece will be far easier to check for flaws, test, maintain, reuse, etc.
- Modules should have low coupling
  - A coupling is any time one module interacts with another module
  - High coupling is a common cause of unexpected behaviors in a program

2. Encapsulation

- Have the modules be mostly self-contained, sharing information only as necessary
- This helps reduce coupling
- The developer of one module should not need to know how a different module is implemented
  - (S)he should only need to know about the published interfaces to the other module (the API)

3. Information hiding

- The internals of one module should not be visible to other modules
- This is a stronger statement than encapsulation: the implementation and internal state of one module should be hidden from developers of other modules
- This prevents accidental reliance on behaviors not promised in the API
- It also hinders some kinds of malicious actions by the developers themselves!

4. Mutual suspicion

- It’s a good idea for modules to check that their inputs are sensible before acting on them
- Especially if those inputs are received from untrusted sources
- But also as a defense against flaws in, or malicious behaviour on the part of, other modules
  - Corrupt data in one module should be prevented from corrupting other modules

5. Confinement

- Similarly, if Module A needs to call a potentially untrustworthy Module B, it can confine it (also known as sandboxing)
  - Module B is run in a limited environment that only has access to the resources it absolutely needs
- This is especially useful if Module B is code downloaded from the Internet

Security controls – Implementation

- When you're actually coding, what can you do to control security flaws?
- Don't use C (but this might not be an option)

1. Static code analysis
2. Formal methods
3. Genetic diversity
4. Finally: learn more!

1. Static code analysis

- There are a number of software products available that will help you find security flaws in your code
  - These work for various languages, including C, C++, Java, Perl, PHP, Python
  - Examples: Graudit, SonarQube, etc.
- They often look for things like buffer overflows, but some can also point out TOCTTOU and other flaws
- These tools are not perfect!
  - They're mostly meant to find suspicious things for you to look at more carefully
  - They also miss things, so they can't be your only line of defense

2. Formal methods

- Instead of looking for suspicious code patterns, formal methods try to prove that the code does exactly what it's supposed to do
  - And you thought the proofs in your math classes were hard?
  - Unfortunately, we can show that this is impossible to do in general
    - But that doesn't mean we can't find large classes of useful programs where we can do these proofs in particular
- Usually, the programmer will have to "mark up" his/her code with assertions or other hints to the theorem proving program
  - This is time-consuming, but if you get a proof out, you can really believe it!

3. Genetic diversity

- The reason worms and viruses are able to propagate so quickly is that there are many, many machines running the same vulnerable code
  - The malware exploits this code
- If there are lots of different HTTP servers, for example, there's unlikely to be a common flaw

- This is the same problem as in agriculture
  - If everyone grows the same crop, they can all be wiped out by a single virus

4. Learn more about software security

- We barely scratched the surface in this course
- If you are thinking about becoming a software developer, get one of these books:
- "Building Secure Software - How to Avoid Security Problems the Right Way" by John Viega and Gary McGraw
- "Writing Secure Code (Second Edition)" by Michael Howard and David LeBlanc

Security controls - Change management

- Large software projects can have dozens or hundreds of people working on the code

- Even if the code's secure today, it may not be tomorrow!

- If a security flaw does leak into the code, where did it come from?
  - Not so much to assign blame as to figure out how the problem happened, and how to prevent it from happening again

Source code and configuration control

- Track all changes to either the source code or the configuration information (what features to enable, what version to build, etc.) in some kind of management system
- There are dozens of these; you've probably used at least a simple one before
  - CVS, Subversion, git, darcs, Perforce, Mercurial, Bitkeeper, ...
- Attempted backdoor in the Linux source code:
  - Bitkeeper can notice a change to the source repository that does not match any valid check-in

Security controls - Code review

- Empirically, code review is the single most effective way to find faults once the code has been written
- The general idea is to have people other than the code author look at the code to try to find any flaws
- This is one of the benefits often touted for open-source software: anyone who wants to can look at the code
  - But this doesn't mean people actually do!
  - Even open-source security vulnerabilities can sit undiscovered for years, in some cases

Kinds of code review

- There are a number of different ways code review can be done
- The most common way is for the reviewers to just be given the code
  - They look it over, and try to spot problems that the author missed
  - This is the open-source model

Guided code reviews

- More useful is a guided walk-through
  - The author explains the code to the reviewers
  - Justifies why it was done this way instead of that way
  - This is especially useful for changes to code
    - Why each change was made
    - What effects it might have on other parts of the system
    - What testing needs to be done
- Important for safety-critical systems!

"Easter egg" code reviews

- One problem with code reviews (especially unguided ones) is that the reviewers may start to believe there's nothing there to be found
  - After pages and pages of reading without finding flaws (or after some number have been found and corrected), you really just want to say it's fine
- A clever variant: the author inserts intentional flaws into the code
  - The reviewers now know there are flaws
  - The theory is that they'll look harder, and are more likely to find the unintentional flaws
  - It also makes it a bit of a game

Security controls – Testing

- The goal of testing is to make sure the implementation meets the specification
- But remember that in security, the specification includes "and nothing else" (?)
  - How do you test for that?!
- Two main strategies:
  - Try to make the program do unspecified things just by doing unusual (or attacker-like) things to it
  - Try to make the program do unspecified things by taking into account the design and the implementation

Black-box testing

- A test where you just have access to a completed object is a black-box test
  - This object might be a single function, a module, a program, or a complete system, depending on at what stage the testing is being done

Fuzz testing

- One easy thing you can do in a black-box test is called fuzz testing
- Supply completely random data to the object
  - As input in an API
  - As a data file
  - As data received from the network
  - As UI events
- This causes programs to crash surprisingly often!
  - These crashes are violations of Availability, but are often indications of an even more serious vulnerability

White-box testing

- If you're testing conformance to a specification by taking into account knowledge of the design and implementation, that's white-box testing
  - Also called clear-box testing
- Often tied in with code review, of course
- White-box testing is useful for regression testing
  - Make a comprehensive set of tests, and ensure the program passes them
  - When the next version of the program is being tested, run all these tests again

Security controls – Documentation

- How can we control security vulnerabilities through the use of documentation?

- Write down the choices you made
  - And why you made them
- Just as importantly, write down things you tried that didn't work!
  - Let future developers learn from your mistakes
- Make checklists of things to be careful of
  - Especially subtle and non-obvious security-related interactions of different components

Security controls – Maintenance

- By the time the program is out in the field, one hopes that there are no more security flaws
  - But there probably are
- We've talked about ways to control flaws when modifying programs
  - Change management, code review, testing, documentation

Standards, process, and audit

- Within an organization, have rules about how things are done at each stage of the software lifecycle
- These rules should incorporate the controls we've talked about earlier
- These are the organization's standards
- For example:
  - What design methodologies will you use?
  - What kind of implementation diversity?
  - Which change management system?
  - What kind of code review?
  - What kind of testing?

Standards, process, and audit

- Make formal processes specifying how each of these standards should be implemented
  - For example, if you want to do a guided code review, who explains the code to whom? In what kind of forum? How much detail?

- Have audits, where somebody (usually external to the organization) comes in and verifies that you're following your processes properly

- This doesn't guarantee flaw-free code, of course!

Recap

- Flaws, faults, and failures
- Unintentional security flaws
- Malicious code: Malware
- Other malicious code
- Nonmalicious flaws
- Controls against security flaws in programs

Recap

- Various controls applicable to each of the stages in the software development lifecycle

- To get the best chance of controlling all of the flaws:
  - Standards describing the controls to be used
  - Processes implementing the standards
  - Audits ensuring adherence to the processes