Secure Programming Techniques

Meelis Roos
mroos@ut.ee

Institute of Computer Science
Tartu University

spring 2019
Course outline

• Introduction
• General principles
• Code auditing
• C/C++
• Web
• SQL Injection
• PHP
• Shell scripts
• Java
Literature

- OWASP *(Open Web Application Security Project)* documents
- Secure Programming for Linux and Unix HOWTO — Creating Secure Software
- Secure coding: principles and practices, Mark Graff, Kenneth R. Van Wyk, O’Reilly 2003
- Secure Programming with Static Analysis, Brian Chess, Jacob West, Addison-Wesley Professional, 2007
Dates and times

- 4 homeworks during the semester (mostly to find the bugs and report or fix)
- Homework deadline 2-3 weeks (noted with each homework)
- Question: suitable dates for exams in June?
Optional project

- Practical project is under different course code (MTAT.07.016, https://courses.cs.ut.ee/2018/secprog-proj/)
- Goal: find and fix a new security problem in real software
- Find an open-source project where you can find one or more vulnerabilities
- Use code analysis tools and read code
- Document the bug
- Fix the bug (all bugs of the same kind if possible)
- Test and document the fixes
- Send a patch upstream, rewriting it if asked, until the patch is merged
- Give a presentation and submit a report
Introduction

- Scope of this course
- Why programs are insecure
- Security requirements
- General principles
**Scope of this course**

- Learn about secure *coding* practices in popular and widely used languages and environments
- Not about exploitation of vulnerabilities — only enough to see why the problems are relevant
- Not about designing the architecture of secure systems — there’s a different course for that
- Some lab exercises (you will need a laptop)
- Some Unix and Windows system programming background is also included
Why are programs insecure?

- Economic reasons — consumers do not select products based on real security
  - So implementing real security is unnecessary for selling
- Security measures affect usability and waste users’ time so a real security feature might waste more time and money on global scale than possible losses from bad security
- Marketing: don’t worry, be crappy!
  - Let’s go into production with it now and we can secure it later
Why are programs insecure?

- Programmers are not taught enough
- Programmers are lazy
- Many programmers are lousy programmers
- Programmers are not security experts
- Security costs more (time, money)
- C/C++ are hard to write securely
- No thought is given to multi-user systems
- Insufficient security models
- Formal verification is rarely used
- Users don’t care
- Lots of old software is in use
Security requirements

- So, what exactly does *secure* mean?
- General security goals as usual:
  - Confidentiality
  - Integrity
  - Availability
- We need more specific requirements to be usable in validating a program
- Example: as per Common Criteria (CC)
CC Functionality Requirements

- Auditability
- Non-repudiation
- Cryptography support
- User data protection
- Identification and authentication
- Security management
- Privacy
- Protection of security functions
- Resource utilization
- Access
- Trusted path
CC Assurance Measure Requirements

- Configuration management
- Delivery and operation
- Development
- Guidance documents
- Life-cycle support
- Tests
- Vulnerability Assessment
- Maintenance of assurance
General principles

• Security must be designed into the system from the beginning, not patched in later
• Security can be audited only relative to knowledge in specific point of time
• Enumerating badness does not work
• Paranoia is a virtue
• Many simple bugs can appear as vulnerabilities when the data to trigger it comes from another security domain (untrusted source)
• Complexity is your enemy
Outline of next topics

- Security features
- Open Source software and security
- Disclosure of vulnerabilities
- Top vulnerability classes
Security features

- Security features $\neq$ secure features
- Reliable software does what it is supposed to do
- Secure software does what it is supposed to do, and nothing else
Is Open Source good for security?

- More reviewers ⇒ more errors are found
  - And fixed?
  - What makes people review the code?
  - Closed source programs can also have stringent review procedures

- Possibility of checking the quality of code
  - "No strcat comes though the door!"
  - "Oh, the scientists again!"

- More incentives to making the code clean

- Open source by itself does not guarantee quality!
  - Many open source projects never get beyond one-man stage
Open Source and security

- Attacker has more information than with closed source
  - He can reverse engineer anyway
- Attacker has less advantages over good guys
- Trojan horses — where do they come from?
- Chance of fixing if vendor does not care or is dead
  - Or pay someone to fix it
- Conclusion: open source has the potential of getting better over time
Disclosure of vulnerabilities

- Prehistory: report or don’t report, most vendors pretended nothing happened
  - False sense of security
- Full disclosure
  - Vendors need to start patching
  - Finder gets fame
- Responsible disclosure
  - Vendors are given some time to fix before disclosure
  - Coordinating fixes
  - Finder still gets fame
- "No more free bugs"
  - Selling of vulnerabilities
OWASP Top 10 vulnerabilities (2017)

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. Cross-Site Scripting (XSS)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Request Forgery (CSRF)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring
CWE/SANS TOP 25 Most Dangerous Software Errors

1. Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)
2. Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)
3. Improper Neutralization of Input During Web Page Generation (‘Cross-site Scripting’)
4. Unrestricted Upload of File with Dangerous Type
5. Cross-Site Request Forgery (CSRF)
6. URL Redirection to Untrusted Site (‘Open Redirect’)

CWE/SANS TOP 25 Most Dangerous Software Errors

7. Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)

8. Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)

9. Download of Code Without Integrity Check

10. Inclusion of Functionality from Untrusted Control Sphere

11. Use of Potentially Dangerous Function

12. Incorrect Calculation of Buffer Size

13. Uncontrolled Format String

14. Integer Overflow or Wraparound
CWE/SANS TOP 25 Most Dangerous Software Errors

15. Missing Authentication for Critical Function
16. Missing Authorization
17. Use of Hard-coded Credentials
18. Missing Encryption of Sensitive Data
19. Reliance on Untrusted Inputs in a Security Decision
20. Execution with Unnecessary Privileges
21. Incorrect Authorization
22. Incorrect Permission Assignment for Critical Resource
23. Use of a Broken or Risky Cryptographic Algorithm
24. Improper Restriction of Excessive Authentication Attempts
25. Use of a One-Way Hash without a Salt