Mag. iur. Dr. techn. Michael Sonntag

Secure Development

Security and Privacy

VSE Prag, 7 - 11.6.2010

Institute for Information Processing and
Microprocessor Technology (FIM)
Johannes Kepler University Linz, Austria

E-Mail: sonntag@fim.uni-linz.ac.at
http://www.fim.uni-linz.ac.at/staff/sonntag.htm

© Michael Sonntag 2010

Agenda

- The proactive security process
  - → Education
  - → Design/Development/Test/Maintenance phase
- Security principles: SD^3
  - → Minimizing the attack surface
  - → Defence in depth
  - → Least privilege
  - → Reluctance to trust
  - → Security through obscurity
  - → Secure defaults
- Marketing caveats
- Good practices and guidelines

Proactive Security Development Process:
Combating insecurity during the whole process

- What is secure code?
  - → Not: Security code, code that implements security features
    - » "Adding SSL" will **not** render your program secure!
  - → Code designed to withstand attacks by malicious persons
- Main problem: Developing secure code takes longer, but does not generate any revenue
  - → Tight schedule (time & money) → "We add it later!"
    - » The program will already be vulnerable
    - » Successful attacks cost money too …
- To counter these and other problems, security must be tightly integrated into the whole development process
  - → Design: Can this be made secure, how will this affect users?
  - → Development: Ensuring less bugs are introduced
  - → Testing: Avoid regressions, testing security
  - → Maintenance: Patches for bugs and new attack methods

The defenders dilemma

- Attackers can choose any point they want → The weakest
  - → The defender must secure each and every aspect!
  - → Attackers usually have no time pressure
- Attackers can search for unknown vulnerabilities
  - → Defenders can fix only those holes, they know of
  - → Defenders can only code against know techniques
    - » And not every developer can know all of them!
- Attackers can strike any time
  - → Defenders must watch continuously and without pause
  - → Attackers can wait for a new attack to be published and employ it before the patches (if available!) have been installed
- Attackers do not have to follow the rules
  - → Defenders must keep their communication correct according to protocols and standards
  - → Attackers may send one-way, illegally encoded, … data

Education

- Typical training involves functionality only
  - → How to implement an algorithm, write a UI
    - » How to implement a security feature, e.g. encryption
      - – Example: How RSA works and which procedures to call in C/C++/…
  - → Finding out, what customers really want
- What should be included in education:
  - → How to use a security feature to combat specific attacks
    - » How can you employ RSA to prevent man-in-the-middle attacks?
  - → Systemic view on security
    - » Interdependency of random number generators and encryption
  - → How vulnerabilities look like
    - » To be able to recognize them in your code, e.g. buffer overflow
  - → Security mindset: Not forgetting about it
    - » All aspects should include at least some thought about security

Security education for all: Developers **+** Designers, Testers, …!

Design phase

- Defining security goals:
  - → What attacks are likely, resources of attackers, special data connections to somewhere else, data to protected higher, user group (all/selected employees, end users, ..), where will the program be deployed (Inter-/Intranet/…), consequences of security breaches (tolerance level), administrator knowledge, existing security infrastructure, interoperability with other security systems, protecting users from their own actions, …
- Security is a feature and must be documented as such
  - → RFC's always include a section on "Security Considerations"
    - » Does your design contain such a section too?
  - → "Keeping security in mind" is not enough: Reserve time for it!
- Features must be designed to work with security
  - → Example: SMTP. A nice and simple protocol, but retrofitting security in it is extremely difficult!

Development phase

- Provide upgrade paths to replace insecure features
  - → Extension mechanisms need to be built in from the start
- Updates should be easy/automatic/without downtime, …
  - → Allow dynamic replacement or easy and fast restart
  - → Allow small elements to be replaced easily, instead of the whole program (reinstallation) → Keeping the configuration!
- Use version control system and restrict update access
  - → Not everyone should be able to change the code
  - → Every change should be annotated (who, why, …)
- Use a peer review system
  - → Every LoC / change should be checked by another person
    - » Ideally by one for correctness (feature) and another for security!
- Define secure coding guidelines
  - → What functions/algorithms must/may not be used
    - » Automation possible in some parts

Development phase

- Mandatory education and reviewing old defects
  - → Everyone should know, what was done wrong in the past
  - → Why did it occur?
  - → What can be done to prevent similar problems in the future?
  - → General education in secure programming
- External security reviews (optional; depending on needs)
  - → Different perspective, unfamiliar with the code
- Manage the bug count
  - → Its is bad to have too many open bugs (≈ 5 open /developer)
    - » Fix the ones found before searching for more
- Introduce a bug management system
  - → Allows keeping track and creating statistics
    - » Try to learn from the statistics
      - – E.g. "Why so many DoS vulnerabilities?"

Test phase

- Follow general testing guidelines
- Test two aspects:
  - → Security mechanisms: Correct implementation of functionality
    - » Standard practices; as all the other functional testing
  - → Risk-based testing: Simulating an attacker
    - » Therefore even more important to do black-box testing!
    - » Usually requires specific expertise and experience
      - – "Normal" QA people might be insufficiently knowledgeable
- Mandatory tests for vulnerabilities
  - → Every vulnerability discovered (regardless when) must be "patched" with a matching test verifying it is closed
- Use automated analysis tools
  - → Looking for dangerous patterns; automated cracking tools
    - » Especially the latter will otherwise be used by the attackers …
- Test not only exploits, but also information disclosure

Maintenance phase

- Ideally: Continue active searching for vulnerabilities
  - → At least: Monitor forums, CERTs etc. for vulnerabilities discovered by third persons
  - → Also: Don't forget checking previous version for bugs discovered in the new version
- Provide a list of (potential) security problems/limitations
  - → These should not be dangerous, or there should already have been a patch (or the program not shipped at all!)
- Install a well-defined response process
  - → Who is responsible for what before when?
- Fix bugs and try to ensure that patches are actually installed
  - → See automatic updates before!
  - → When discovering a bug, check the rest of the program for a similar problem as well
  - → Who wrote the code should also fix it (→ Learning!)

Security principles

- Security should be designed right into the application
  - → This requires security principles as guidelines
- Security is a crosscut-issue: It applies to all code sections
- Security principles are:
  - → Minimizing the attack surface
  - → Defence in depth
  - → Least privilege
  - → Reluctance to trust
  - → Obscurity ≠ Security
  - → Secure defaults
  - → Security issue response process
  - → Never mix data and code

(Minimizing the attack surface … Secure defaults:) We will cover those here briefly!

(Security issue response process, Never mix data and code:) Also very important, but not covered here

- These principles are valid for all phases of the software development cycle and all participants
  - → Including managers, designers, coders, testers, …

- SD³: Secure by Design, Default and Deployment
  - → Design: There should be no vulnerabilities "built-in", only accidentally introduced later in implementation
    - – Includes also implementation and test phase!
    - » Example: Introduce a threat model and design "against" it
    - » Example: Regression tests, code simplification, penetration tests
  - → Default: After installation it should be secure
    - » Anything necessary (passwords) should be checked for security
    - » Making it unsafe = sole decision and work of the end user!
      - – Also helps against legal problems
    - » Example: Least privilege
  - → Deployment: Managing it should not decrease security
    - » Easy updates, difficult to introduce insecure configuration
    - » Ensure patches do not introduce new bugs or vulnerabilities
    - » Example: Information on how to secure the system (or keep it so)

Minimizing the attack surface

- The more open ports (protocols supported, file formats accepted, daemons, communication methods, registry keys etc.), the more possibilities for attacks
  - → Every point of entry to the program is a danger
  - → One point is easier to secure than multiple ones
    - » Especially regarding interdependencies and race conditions!
- Single points of access are also single points of failure
  - → So if anything goes wrong there, everything stops
- Try to have only one point of access for each class
  - → Reduce privileges of code
  - → A single port for all communication
  - → A single type of dynamically generated webpages
  - → Disable optional functionality (80/20 rule)
  - → Require authentication for entry points
  - → Limit to actual needs (e.g. if using TCP don't listen for UDP)

Defence in depth

- If everything fails, there should be another layer of protection
  - → Do not depend on a single line of defence
    - » Example: When the firewall fails, you app. should still be secure
  - → Even if some of your own defences are breached, other parts should remain (at least partially) secure
    - » Example: DoS attack successful → Still no DB corruption
  - → Multiple redundant security measures (→ Costs!)
- Examples:
  - → Internal routers/switches can do some filtering as well
  - → 802.1x → Keep unknown devices out; but still require authentication for users and individual services
  - → Perimeter firewall and personal firewalls
  - → AV software on firewall, mail gateway and clients
  - → IDS for detecting what managed to get past the firewall
  - → Logging to gather information about successful breaches

Least privilege

- Require only the minimum privileges absolutely necessary
  - → BIG problem on Windows
  - → Reduces interaction with other programs as well
    - » Unwanted/unintentional/improper usage is restricted too
  - → Services run as root: If breached, attacker is administrator!
    - » Most programs will be attacked successfully sometimes; least privilege reduces the consequences (defence in depth!)
- Recipe:
  - → What resources must be accessed/special tasks performed?
    - » ACLs on registry keys and files installed are important too!
  - → What needs to be done with these?
  - → What are the minimum permissions to complete these?
    - » Grant only those permissions necessary to complete the task
  - → Testing: Run/install a program as a "normal" user and log
    - » What does not work? Is it necessary? Reconfiguration possible? What elevated privileges are just barely sufficient?

Reluctance to trust: "All other systems are insecure or evil"

- Depend only on yourself – All other programs are prone to fail or will be used as points to attack you
  - → Everything you receive from another system should be viewed with suspicion: There is no other "trusted" system
    - » Example: If the DB is on a different server (or a different program!) you should validate its input to you
      - – There might be a bug in it, it could have been hacked, …
    - » Make sure it is actually the system it claims to be
      - – Attacks on the DNS infrastructure are very dangerous!
  - → At least define who you trust, why, and in which respect
- Users are inherently evil
  - → E.g. Web 2.0: Client-side validation is useful & good for users (fast feedback), but no replacement for server-side validation!
    - » If possible, clients should also be distrustful of servers!
- See also social engineering!
- Be careful of the transitivity of trust

Security through obscurity

- Security depends on nobody knowing how it works
  - → Keeping algorithm, access scheme, key generation, .. Secret
  - → Secrecy of design, implementation, deployment
- Problem: Attackers will find out how it works!
  - → It might be difficult and take longer, but it will happen
- Controversial issue!
  - → Secrecy might help + delay, but you should not depend on it
  - → Secrecy is good, but **relying on it** is bad!
- Recommendations:
  - → Only use openly available/described cryptography
    - » Never invent your own cryptographic algorithm
      - – It is extremely difficult to design even a "good" one!
  - → Vulnerabilities are usually kept secret until a patch is available
    - » Exploits available/common knowledge → Publish immediately
  - → Rely on data secrecy, not the operations performed on it

Secure defaults

- When software is installed, it should be secure
  - → "Dangerous" options should be enabled explicitly by the user
    - » Example: Password aging and complexity checks enabled; anonymous login (guest accounts) disabled; random initial password (instead of standard vendor password!); standard deny until enabled explicitly
  - → If not, provide prominent warning messages
    - » Example: Configuration pages in web apps should be deleted after configuration (or access restricted on several levels)
- Optional parameters should be secure
  - → Not "logical" or "most often used" value, but "most secure"
- Do not check for failures, check for success
  - → Everything else is the default, which is to deny the action
- Use appropriate ACLs for all resources you create
  - → Especially registry keys and (temporary) files

Marketing caveats: What to look out for

- When evaluating security products, look out for
  - → "New encryption algorithm": If its new, its untested
    - » If it’s a custom development, its practically by definition insecure
  - → "Secret security features": If they are not willing to tell you, how can you assess their quality (security by obscurity)?
  - → Technobabble: New and exciting terms and trademarks
  - → Recoverable keys: If they can, so can the attackers
    - » If they are deposited by a third party → Do you accept this?
  - → "We are compliant with …": They paid a lot of money to consultants and might be secure against specific attacks
  - → "Unbreakable": Nothing is unbreakable (except one-time-pad)
    - » Always look especially at the circumstances, for which this "unbreakability" is guaranteed (availability, impersonation, …?)!
      - – E.g., only when the USB key is not stolen; not a local user; …
      - – E.g. RSA → key length>=2048, no new mathematical attacks, not enough money for special hardware, …
    - » Similar: "Bulletproof solution"

Good practices

- Take care of password storage and crypt. key generation
- Encrypt all sensitive data
- Reduce the permissions your programs run under
- Employ secure programming languages
  - → Or actively search for security problems/use special software
- Default to deny
- Disable everything not actively needed
- Be distrustful of everything from outside your program
- Check everything on the server/in your own program
- Enforce secure configuration by end-users
- Check for insecurities – attack your own software
- Use security precautions at many stages
- Do not depend on secrecy
- Keep it simple

Guidelines

- Security and complexity are often inversely proportional
  - → Keep functionality, devel. process, security process simple
- Security and usability are often inversely proportional
  - → Mind your users: Can they even understand a security dialog?
- Good security now is better than perfect security never
  - → Better do something than nothing at all (because you can't get it completely secure anyway)
- False sense of security is worse than true sense of insecurity
  - → Insecurity helps you fixing it
- Security is only as strong as the weakest link
  - → "Distribute" security evenly: A singular extremely hard point is useless if it can be circumvented
- Concentrate on known, probable threats
  - → Defend against the known and usual attacks first

Avolio, F.: Best Practices in Network Security
http://www.networkcomputing.com/1105/1105f2.html

Conclusions

- Security must be integrated into the whole development process from the design on
  - → This requires education for all stakeholders
    - » In different areas; but all are affected
- Moderate security is cheap to implement, as it only depends on the "how" of the development, not an additional effort
  - → "Good" security will cost something, as more time for design and implementation is necessary
    - » Also, some features might have to change/be removed
- Do not burden users with security:
  - → They rarely possess the knowledge for a good decision
  - → Automation is therefore important

Security is an investment, not an expense!

Questions?

Thank you for your attention!

© Michael Sonntag 2010

Literature

- Michael Howard, David LeBlanc: Writing Secure Code.
- Matt Curtin: Snake Oil Warning Signs. http://www.interhack.net/people/cmcurtin/snake-oil-faq.html