Secure Programming Lecture 9: Secure Development

David Aspinall, Informatics @ Edinburgh

12th February 2016

Recap

We've looked in detail at two important **vulnerability classes**:

- overflows, stack and heap
- injections, command and SQL

We've seen **secure development processes** from the outside:

- vulnerability advisories, CVE classifications
- a maturity model for secure software development: BSIMM

It's time to delve a bit more into **secure development activities** included in BSIMM.

A *Building Security In* Process

We'll look at a:

Secure Software Development Lifecycle (SSDLC)

due to **Gary McGraw** in his 2006 book *Software Security: Building Security In*.

Work by McGraw and others has been combined in the best practices called Building Security In we saw in BSIMM. This is promoted by the US-CERT.

To avoid debates over specific development processes, BSI indexes best practice activities. But activities relate to lifecycle stages.

McGraw's Three Pillars

In *Building Security In*, Gary McGraw proposes three "pillars" to use throughout the lifecycle:

- **I: Applied Risk Management**
  - process: identify, rank then track risk
- **II: Software Security Touchpoints**
  - designing security ground up, not "spraying on"
  - seven security-related activities
- **III: Knowledge**
  - knowledge as applied information about security
  - e.g., guidelines or rules enforced by a tool
  - or known exploits and attack patterns

Security activities during development

How should secure development practices be incorporated into traditional software development?

0. treat security separately as a new activity (wrong)
1. invent a new, security-aware process (another fad)
2. **run security activities alongside traditional**

In business, "touchpoints" are places in a product/sales lifecycle where a business connects to its customers.

McGraw adapts this to suggest "touchpoints" in software development where security activities should interact with regular development processes.

Security activities during lifecycle

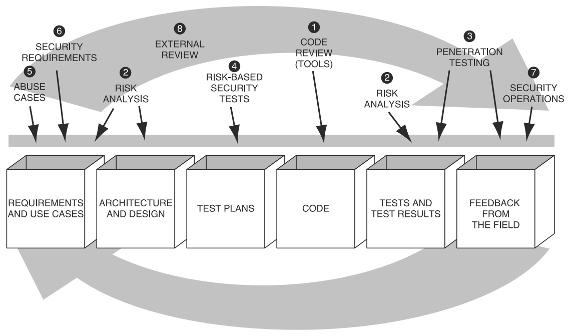
McGraw identified 7 touchpoint activities, connecting to software development artefacts. In lifecycle order:

- **Abuse cases** (in requirements)
- **Security requirements** (in requirements)
- **Risk analysis** (in design)
- **Risk-based security tests** (in test planning)
- **Code review** (in coding)
- **Risk analysis** (in testing)
- **Penetration testing** (in testing and deployment)
- **Security operations** (during deployment)

His process modifies one adopted by Microsoft after the famous *Gates Memo* in 2002.

Exercise. For each touchpoint (detailed shortly), identify the development artefact(s) it concerns.

Touchpoints in the software development lifecycle



The numbers are a ranking in order of effectiveness.

Code review

Most effective step: eliminate problems at source.

Evidence since 1970s shows that bugs are orders of magnitude cheaper to fix during coding than later in the lifecycle (industry is still learning this; code QA processes aren't as widely deployed as you might imagine).

- **Manual code review**
  - can find subtle, unusual problems
  - an onerous task, especially for large code bases
- **Automatic static analysis**
  - increasingly sophisticated tools automate scanning
  - very useful but can never understand code perfectly
  - and may need human configuration, interpretation

Especially effective for simple bugs such as overflows.

Architectural risk analysis

Design flaws are not obvious from staring at code; they need to be identified in the design phase.

Architectural risk analysis considers security during design:

- the security **threats** that attackers pose to **assets**
- **vulnerabilities** that allow **threats** to be realised
- the **impact** and **probability** for a vulnerability exploit
- hence the **risk**, as risk = probability × impact
- **countermeasures** that may be put into place

Example: poor protection of secret keys; risk is deemed high that attacker can read key stored on the filesystem and then steal encrypted document. A countermeasure is to keep encryption keys on dedicated USB tokens.

Risk analysis in general

- Several approaches:
  - financial loss oriented (cost versus damage)
  - mathematical (or pseudo-mathematical) risk ratings
  - qualitative methods using previous knowledge
- If possible, should use specialist non-developers
  - requires understanding business impact
  - perhaps legal and regulatory framework
  - devs often strongly opinionated, fixed assumptions

Question. A risk analysis often begins by looking at value of assets. Why is this not enough?

Common steps in risk analysis

1. Study system (specs, design docs, code if ready)
2. Identify threats and attacker types/routes
3. List possible vulnerabilities in the software
4. Understand planned security controls (& risks. . .)
5. Map attack scenarios (routes to exploit)
6. Perform impact analysis
7. Using likelihood estimates, **rank risks**
8. Recommend countermeasures in priority/cost order

Particular risk analysis methods refine these.

In steps 2 and 3, may use checklists of threat types and previously known vulnerabilities; also general "goodness" guidelines.

Security design guidelines

Saltzer and Schroeder (1975)'s classic principles:

1. **Economy of mechanism**: *keep it simple, stupid*
2. **Fail-safe defaults**: *e.g., no single point of failure*
3. **Complete mediation**: *check everything, every time*
4. **Open design**: *assume attackers get the source & spec*
5. **Separation of privilege**: *use multiple conditions*
6. **Least privilege**: *no more privilege than needed*
7. **Least common mechanism**: *beware shared resources*
8. **Psychological acceptability**: *are security ops usable?*

Exercise. If you haven't studied these already, you should review them in detail.

Microsoft STRIDE approach

STRIDE is a mnemonic for categories of threats in Microsoft's method:

- **S**poofing: *attacker pretends to be someone else*
- **T**ampering: *attacker alters data or settings*
- **R**epudiation: *user can deny making attack*
- **I**nformation disclosure: *loss of personal info*
- **D**enial of service: *preventing proper site operation*
- **E**levation of privilege: *user gains power of root user*

Exercise. Remember the definitions of the familiar CIA security properties (confidentiality, integrity, availability). Explain which properties each threat type attacks.

The STRIDE approach

STRIDE uses *Data Flow Diagrams* to chase data through a system.

- Consider each data flow, manipulation, or storage:
  - Are there vulnerabilities of type S,T,R,I,D,E?
  - Are there routes to attack?
- Design mitigations (countermeasures)

STRIDE was designed as a developer-friendly mechanism

- devs may not know end user's risk tolerance
- so de-emphasises risk assessment, business impact

See MSDN magazine, Nov 2006.

Penetration testing

Current dominant methodology (alongside bolt-on protection measures, outside the lifecycle). Effective because it considers a program in final environment.

- **Finds real problems**
  - demonstrable exploits easily motivates repair costs
  - process "feels" good
- **Drawback: no accurate sense of coverage**
  - ready made pen testing tools cover only easy bugs
  - system-specific architecture and controls ignored

Beware Dijkstra's famous remark: *Testing shows the presence, not the absence of bugs*. Just running some standard pen-testing tools is a very minimal test.

Example: by feeding data to form elements, a browser plugin pen testing tool uncovers XSS vulnerabilities.

Bad use of Pen Testing

- Black-box pen testing by consultants is limited
  - They may know tools but not system being tested
  - Judgements about code can be limited
- Developers only patch problems they're told about
  - Patches may introduce new problems
  - Patches often only fix symptom, not root cause
  - Patches often go un-applied

Good use of Pen Testing

McGraw advocates using pen testing:

- At the unit level, earlier in development:
  - automatic fault-injection with *fuzzing* tools
- Before deployment, as a last check
  - not a first check for security, after deployment!
  - risk-based, focus on configuration and environment
- Metrics-driven: tracking problem reduction
  - not imagining zero=perfect security
  - use exploits as regression tests
- For repairing software, not deploying work-arounds

Security testing

Security testing complements QA processes which ensure main functional requirements are error free.

- **Test security functionality**
  - security provisions tested using standard methods
  - integrated by considering with main requirements
- **Tests based on attack patterns or identified abuse cases**
  - apply risk analysis to prioritize
  - consider attack patterns

Traditional testing vs security testing

Focus on:

- **Explicit** functional requirements
  - check use cases, operate as expected
  - *customer can add/remove items from cart*
- **Sometimes explicit** non-functional requirements
  - check usability, performance
  - *user experience (UX) is pleasing*
  - *updating cart takes at most 5 seconds*

Testers check a reasonably clear list of desired behaviours.

"The system shall. . . "

Traditional testing vs security testing

Focus on:

- **Rarely explicit** non-functional *non*-requirements
  - check many undefined, unexpected behaviours are impossible

Testers check an *un*clear list of *un*desirable behaviours are absent.

"The system shall not. . . "

A strategy for security testing

1. Understand the **attack surface** by enumerating:
   - program inputs
   - environment dependencies
2. Use **risk analysis** outputs to prioritize components
   - (usually) highest: code accessed by anonymous, remote users
3. Work through **attack patterns** using fault-injection:
   - use manual input, *fuzzers* or *proxies*
4. Check for **security design errors**
   - privacy of network traffic
   - controls on storage of data, ACLs
   - authentication
   - random number generation

Automating security tests

Just as with functional testing, we can benefit from building up suites of *automated security tests*.

1. Think like an attacker
2. Design test suites to attempt malicious exploits
3. Knowing system, try to violate specs/assumptions

This goes beyond random *fuzz testing* approaches.

Specially designed **whitebox fuzz testing** is successful at finding security flaws (or, generating exploits).

Rough idea: apply *dynamic test generation*, using symbolic execution to generate inputs that reach error conditions (e.g., buffer overflow).

Abuse cases

Idea: describe the desired behaviour of the system under different kinds of abuse/misuse.

- Work through **attack patterns**, e.g.
  - illegal/oversized input
- Examine **assumptions** made, e.g.
  - interface protects access to plain-text data
  - cookies returned to server as they were sent
- Consider **unexpected events**, e.g.
  - out of memory error, disconnection of server

Specific detail should be filled out as for a use case.

Related idea: **anti-requirements**.

Security requirements

Security needs should be explicitly considered at the requirements stage.

- **Functional security requirements**, e.g.
  - use strong cryptography to protect sensitive stored data
  - provide an audit trail for all financial transactions
- **Emergent security requirements**, e.g.
  - do not crash on ill-formed input
  - do not reveal web server configuration on erroneous requests

Security operations

Security during operations means managing the security of the deployed software.

Traditionally this has been the domain of **information security** professionals.

The idea of this touchpoint is to combine expertise of **infosecs** and **devs**.

Information security professionals

By now, many different types, expert in:

- Incident handling, response (**SOC** team)
- Craft knowledge: malware, vulnerabilities
- Understanding and deploying desirable patches
- Configuring firewalls, IDS, virus detectors, UTMs, SIEMs.

But are rarely *software* experts.

Taking part in the development process can **feed back knowledge from attacks**, or join in **security testing**.

Infosec people understand pentesting from the outside and less from inside. Network security scanners are more evolved and effective than application scanners.

Coders

Expert in:

- Software design
- Programming
- Build systems, overnight testing

But rarely understand *security in-the-wild*.

Coders focus on the main product, easy to neglect the deployment environment. E.g., VM host environment may be easiest attack vector.

Summary

This lecture outlined some SSDLC activities.

The descriptions were quite high-level.

Exercise. For each of the touchpoints, find specific documented examples of their use in a development process. (McGraw's book has some, but there are plenty of other sources).

Exercise. Practice thinking about the touchpoints by constructing scenarios. Consider the development of a particular piece of software or a system. Imagine what some of the touchpoints might uncover or recommend.

Review questions

- Describe **5 secure development lifecycle activities** and the points in which they would be used in a compressed 4-stage agile development method (use case, design, code, test).
- What kinds of security problem is *code review* better at finding compared with *architectural risk analysis*?
- Why is risk analysis difficult to do at the coding level?
- What is the main drawback of penetration testing, especially when it is applied as an absolute measure of security of a software system?

References and credits

Material in this lecture is adapted from

- *Software Security: Building Security In*, by Gary McGraw. Addison-Wesley, 2006.
- *The Art of Software Security Testing*, by Wysopal, Nelson, Dai Zovi and Dustin. Addison-Wesley, 2007.
- *Build Security In*, the initiative of US-CERT at `https://buildsecurityin.us-cert.gov/`.