TIC 4302 Information Security Practicum II

Secure SDLC

Adopted from:

“Secure Programming Lecture 9: Secure Development”

By David Espinal - Informatics @ Edinburgh

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 used in BSIMM. This is promoted by the US-CERT.

To avoid debates over specific development processes, BSI indexes best practice activities. The 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.

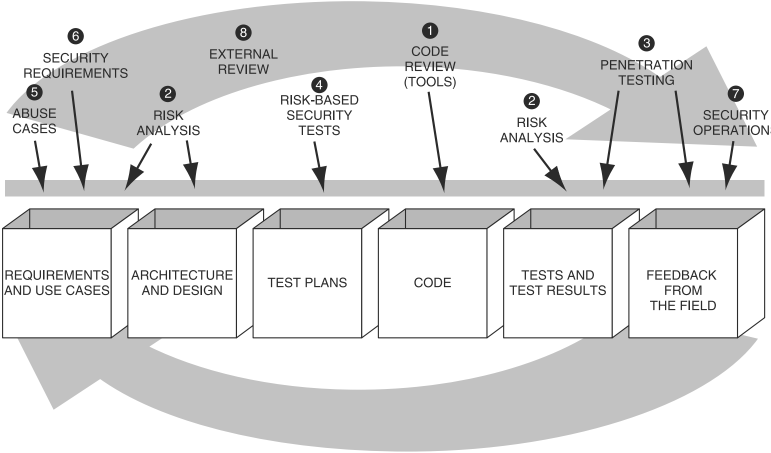
Security activities during lifecycle

McGraw identified 7 touchpoint activity areas, 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.

Touchpoints in the software development lifecycle



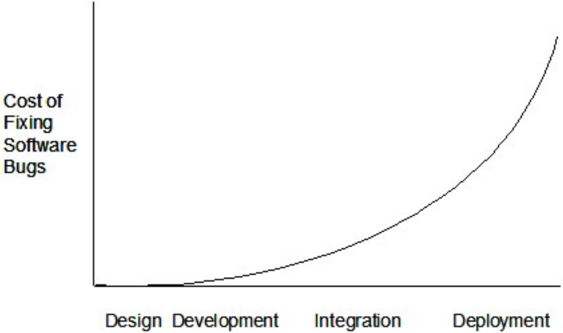
The numbers are a ranking in order of effectiveness.

Code review

Most effective step: eliminate problems at source.

Evidence since 1970s shows 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.



Code review types

- **Manual code review**
  - can find subtle, unusual problems
  - an onerous task, especially for large code bases
  - but adopted dev cycle in some agile processes (e.g., Google)
- **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 starting 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.

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?*

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: something gets “better”
- **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
- Black box pen testing too limited
  - Modern professional pen testing uses source

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

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).

One approach: use *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 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 (avoid DoS)
  - do not reveal web server configuration on erroneous requests (avoid leaks)

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

Expert in:

- Incident handling
- Range and mechanisms of vulnerabilities
- Understanding and deploying desirable patches
- Configuring firewalls, IDS, virus detectors, etc

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 currently more 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.

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/`.