[l06] solutions

[l06_t01] software

wim mees

introduction

learning objectives

- know and understand:
  - how to develop software that is secure
  - what is means when a product is certified

what's the problem ?

the year is 1967

Figure 1: summer of love

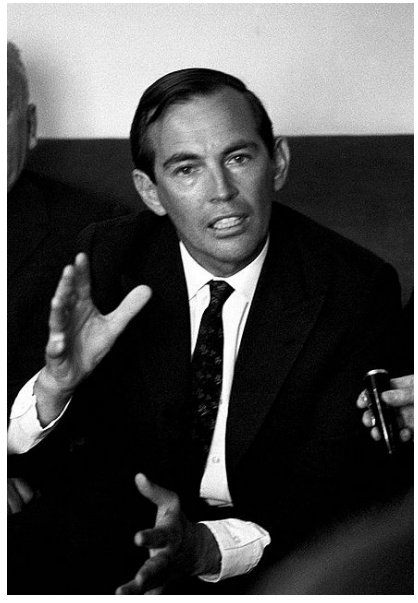

Figure 2: Christiaan Barnard

Christiaan Barnard performs the first human heart transplant operation at Groote Schuur Hospital (Cape Town, South Africa)

20dec1967

Figure 3: Laviolette bridge

Laviolette bridge opens between Trois-Rivières and Bécancour (Quebec, Canada)

software crisis

- low quality
- over budget
- late

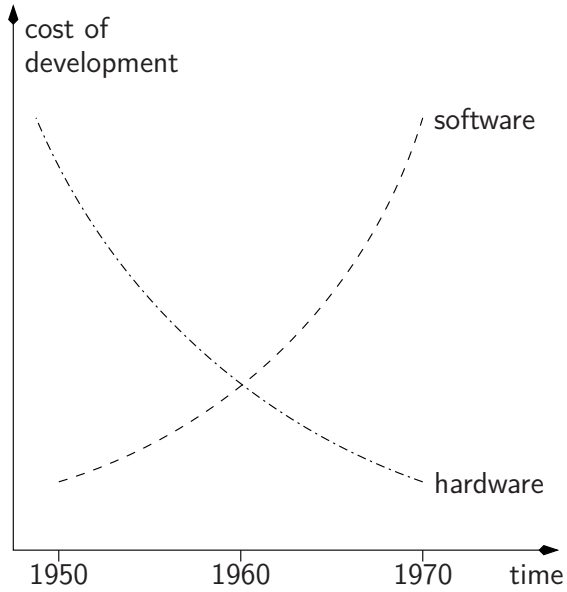

Figure 4: it's official, we have a software crisis

Dijkstra

Edsger W. Dijkstra (1972 Turing Award Lecture)

> *To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

What would Dijkstra have had to say about security and the Internet ?

engineering to the rescue

1968

engineering to the rescue

Figure 5: NATO Software Engineering Conference

1968-1969

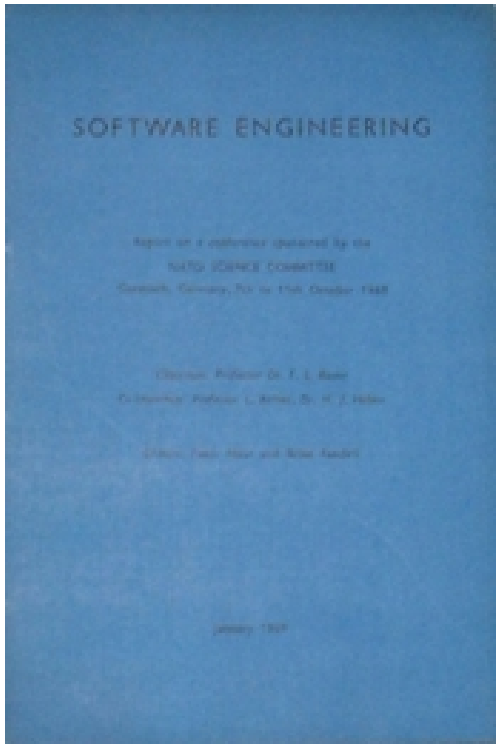

Figure 6: P. Naur and B. Randell (Eds.), Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968

1968-1969

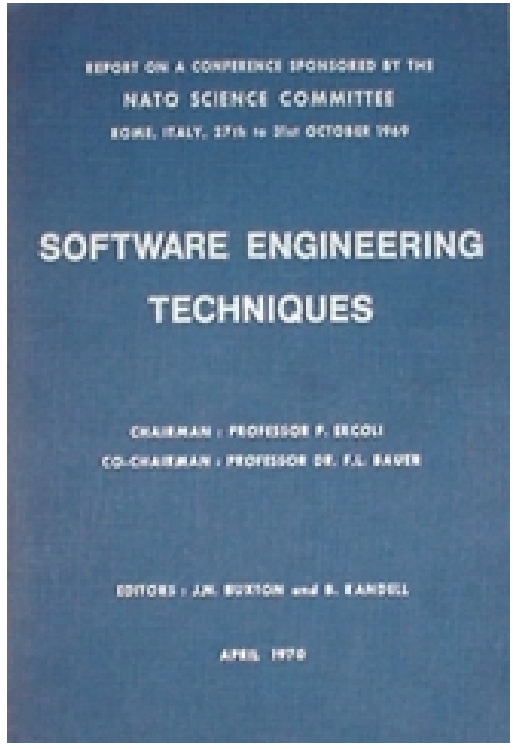

Figure 7: B. Randell and J.N. Buxton (Eds.), Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969

software engineering

software engineering is defined as (ISO/IEC/IEEE Standard 24765):

> *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software.*

waterfall model

theory

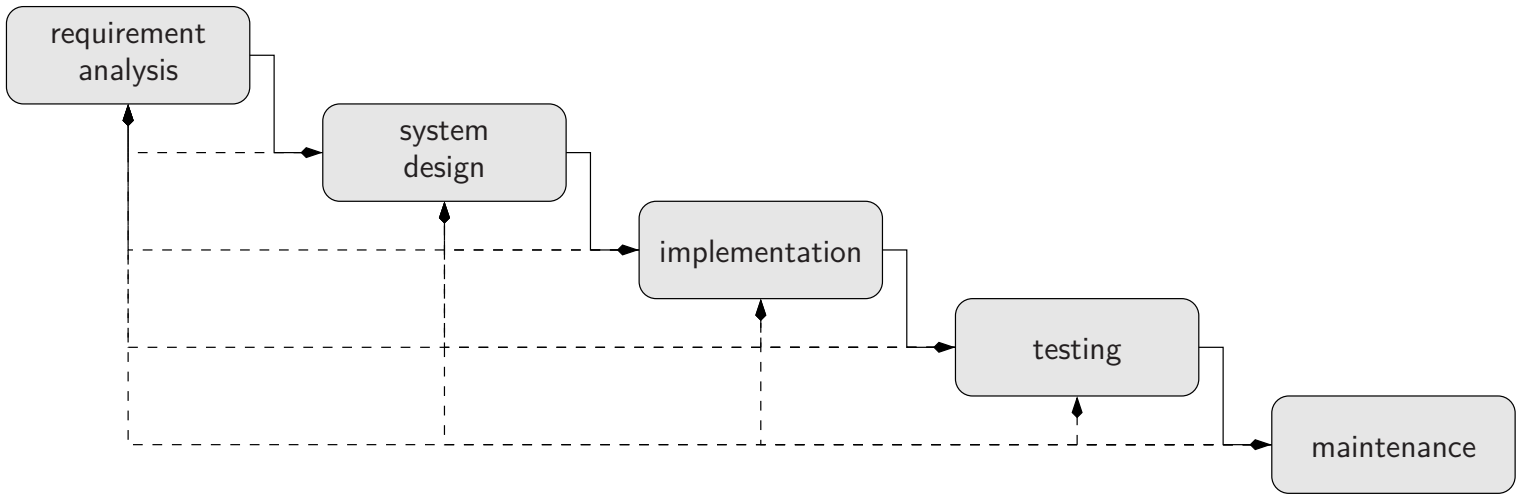

Figure 8: waterfall model

waterfall model

practice

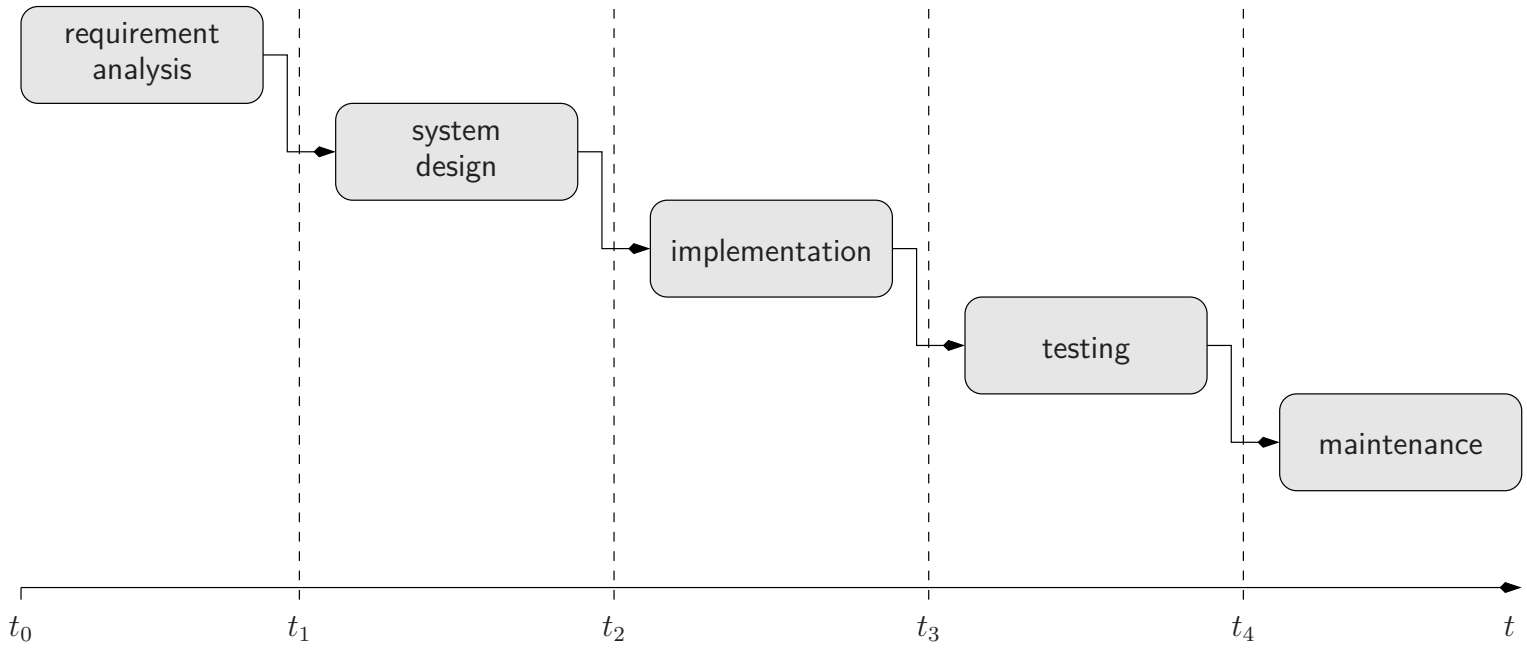

Figure 9: waterfall model

software engineering

and all the methodologies and standards that followed...

- Waterfall evolution (DoD-STD-2167A, MIL-STD-498, ...)
- Boehm's Spiral Model
- Rapid Application Development (RAD)
- Evolutionary Prototyping Model
- Incremental Commitment Model
- Rational Unified Process (RUP)
- V-shaped Defense System Software Development (DoD-STD-2167A)
- Software Life Cycle Processes (ISO/IEC 12207)
- Systems Engineering - System Life Cycle Processes (ISO/IEC 15288)
- ...

quid security ?

where is security in this picture ?

- at the start: security requirements, regulations, ...
- at the end: security acceptance testing, certification, ...

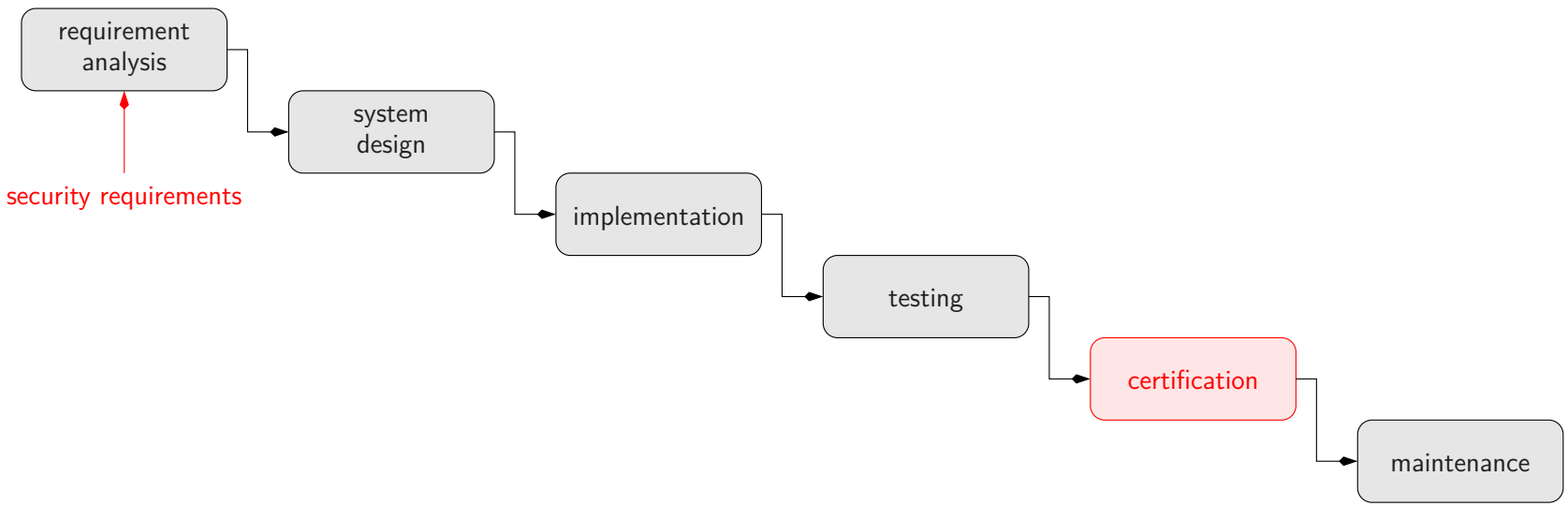

Figure 10: security in the waterfall model

the inevitable legacy

greenfield

brownfield

Figure 11: greenfield versus brownfield

x-off-the-shelf ?

Figure 12: xOTS

in- or outsourcing

Figure 13: outsourcing

secure software development lifecycle

secure development lifecycle

Microsoft Security Development Lifecycle

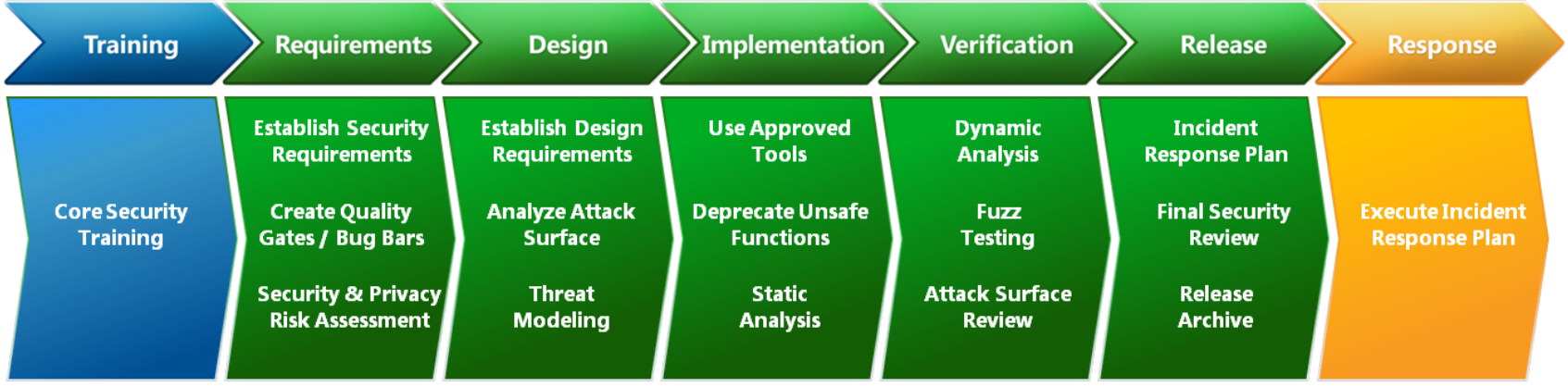

Figure 14: https://www.microsoft.com/en-us/SDL/

secure development lifecycle

training for building better software

- secure design
- threat modelling
- secure coding
- security testing
- best practices surrounding privacy

secure development lifecycle

requirements phase

- establish security and privacy requirements
  - makes it easier to identify key milestones and deliverables and as a result minimize schedule disruptions
- create quality gates / bug bars
  - define minimum acceptable levels of security and privacy quality at the start
  - this helps a team understand risks associated with security issues, identify and fix security bugs during development, and apply the standards throughout the entire project
- perform security and privacy risk assessments
  - examine software design based on costs and regulatory requirements
  - this helps a team identify which portions of a project will require threat modelling and security design reviews

secure development lifecycle

design phase

- establish design requirements
  - considering security and privacy concerns early helps minimize the risk of schedule disruptions and reduce a project's expense
- attack surface analysis / reduction
  - reducing the opportunities for attackers to exploit a potential weak spot or vulnerability requires thoroughly analysing overall attack surface and includes disabling or restricting access to system services, applying the principle of least privilege, and employing layered defences wherever possible
- use threat modelling
  - apply a structured approach to threat scenarios
  - this helps a team more effectively and less expensively identify security vulnerabilities, determine risks from those threats, and establish appropriate mitigations

secure development lifecycle

implementation phase

- use approved tools
  - publishing a list of approved tools and associated security checks (such as compiler/linker options and warnings) helps automate and enforce security practices easily at a low cost
  - keeping the list regularly updated means the latest tool versions are used and allows inclusion of new security analysis functionality and protections
- deprecate unsafe functions
  - analyse all project functions and APIs and ban those determined to be unsafe
  - use for instance code scanning tools to check code for functions on the banned list, and then replace them with safer alternatives
- perform static analysis
  - analysing the source code prior to compile provides a scalable method of security code review and helps ensure that secure coding policies are being followed

secure development lifecycle

verification phase

- perform dynamic analysis
  - performing run-time verification checks software functionality using tools that monitor application behaviour for memory corruption, user privilege issues, and other critical security problems
- fuzz testing
  - inducing program failure by deliberately introducing malformed or random data to an application helps reveal potential security issues prior to release while requiring modest resource investment
- attack surface review
  - reviewing attack surface measurement upon code completion helps ensure that any design or implementation changes to an application or system have been taken into account, and that any new attack vectors created as a result of the changes have been reviewed and mitigated including threat models

secure development lifecycle

release phase

- create an incident response plan
  - crucial for helping to address new threats that emerge over time
  - includes identifying appropriate security emergency contacts and establishing security servicing plans for code inherited from outside
- conduct final security review
  - usually includes examining threat models, tools outputs, and performance against the quality gates and bug bars defined during the requirements phase
- certify release and archive
  - ensure security and privacy requirements were met
  - archiving all pertinent data is essential for performing post-release servicing tasks

secure development lifecycle

response phase

- execute incident response plan
  - essential for protecting customers from software security or privacy vulnerabilities that emerge

software testing

- *"Static Application Security Testing"* (SAST)
  - style checking
  - semantic analysis
  - flow analysis
  - comment density
  - cyclomatic complexity
  - fuction metrics
- *"Dynamic Application Security Testing"* (DAST)
  - fuzzing

security debt

- similar to technical debt
- present in all software, two types:
  - known: identified but yet to be addressed
  - unknown: yet to be discovered
- results from trade-off between:
  - fix everything before shipping (a.k.a. never ship)
  - fast time-to-market by fixing only critical issues
- sources:
  - self: our own development
  - supply chain: outsourced development
  - dependency: COTS components used
- result: continuously growing mountain of low/medium bugs

security debt

accelerated discovery

- requirements review
- design review
- static analysis: examine code and documentation without running the program
  - automated (e.g. lint & sons)
  - manual
- dynamic analysis: interact with the running program
  - automated (e.g. fuzzing)
  - manual (e.g. pen-testing)
- increase awareness and knowledge
- identify lessons from root cause analysis

measuring technical debt

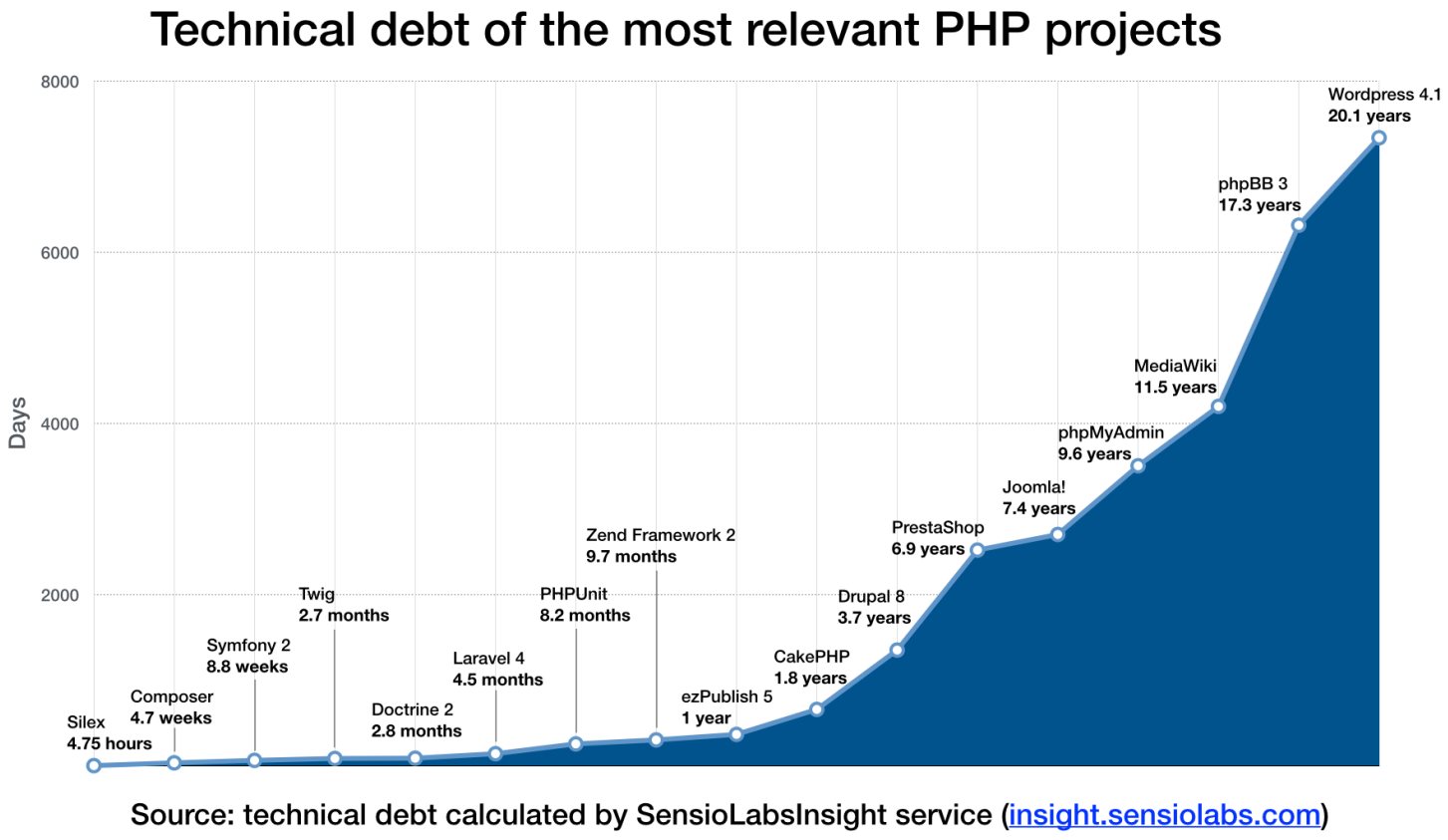

Figure 15: http://blog.insight.sensiolabs.com/2014/11/04/technical-debt-relevant-projects.html

case study: Apple SSL security update

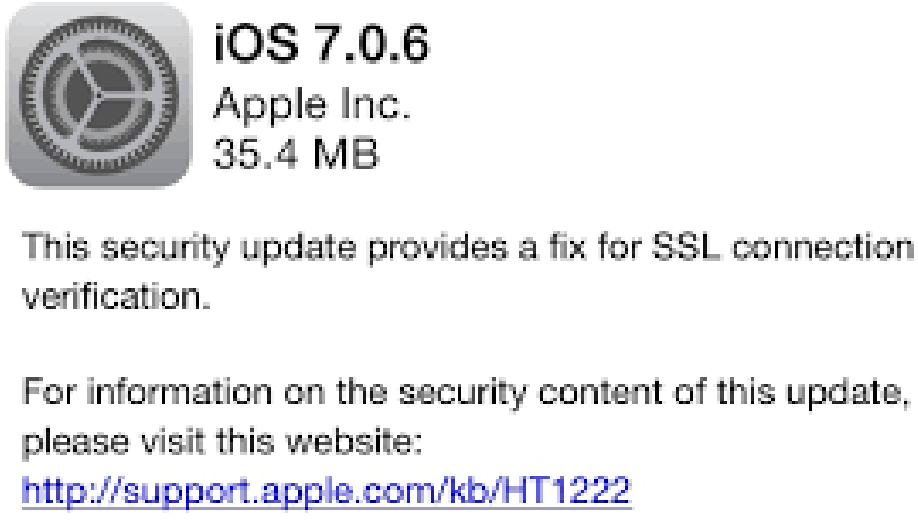

Figure 16: does technical debt also imply security debt ?

case study: Apple SSL bug

```
static OSStatus SSLVerifySignedServerKeyExchange
  (SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
    uint8_t *signature, UInt16 signatureLen)
{ OSStatus          err;
  ...
  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
    ...
fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
}
```

case study: Apple SSL bug

- published in Feb 2014 (now fixed)
- effect:
  - APT L2 MITM attacks go undetected
  - https/pops/imaps/... on a public WiFi
- vulnerable: recent versions of
  - OSX (10.9.0 and 10.9.1)
  - iOS (6.1.5, 7.0.4, and 7.0.5)
- questions:
  - deliberate / accidental ?
  - no unreachable code detection ?
  - no unit testing ?
- note:
  - code was available at http://opensource.apple.com/.../sslKeyExchange.c

case study: heartbleed

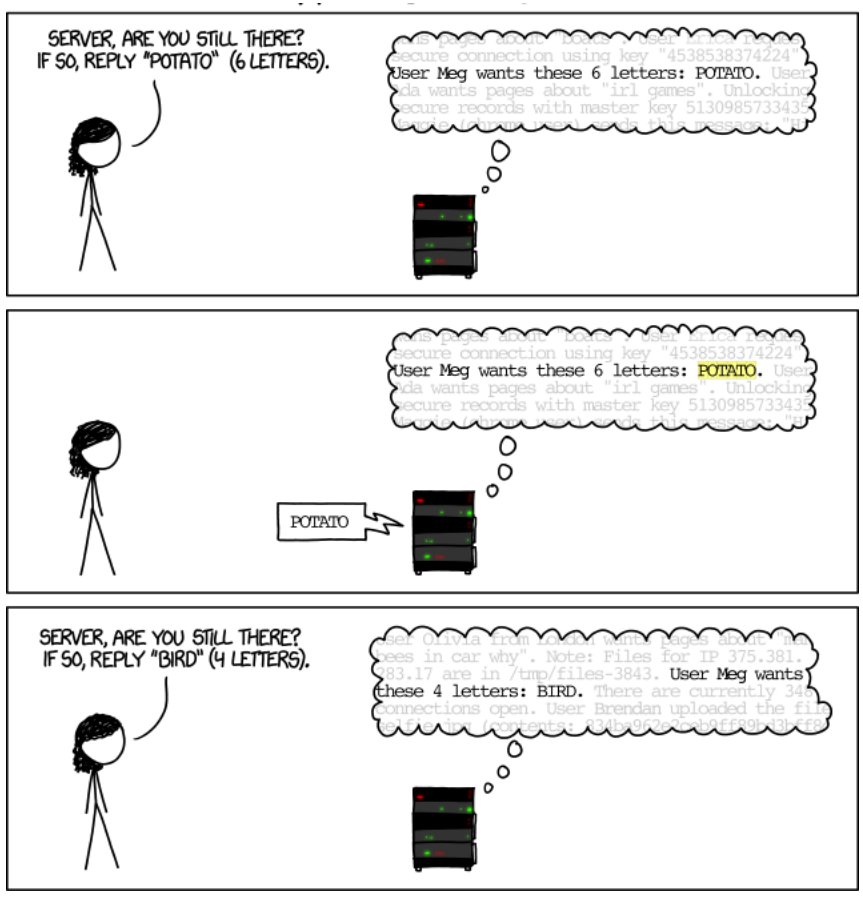

Figure 17: http://xkcd.com/1354/

case study: heartbleed

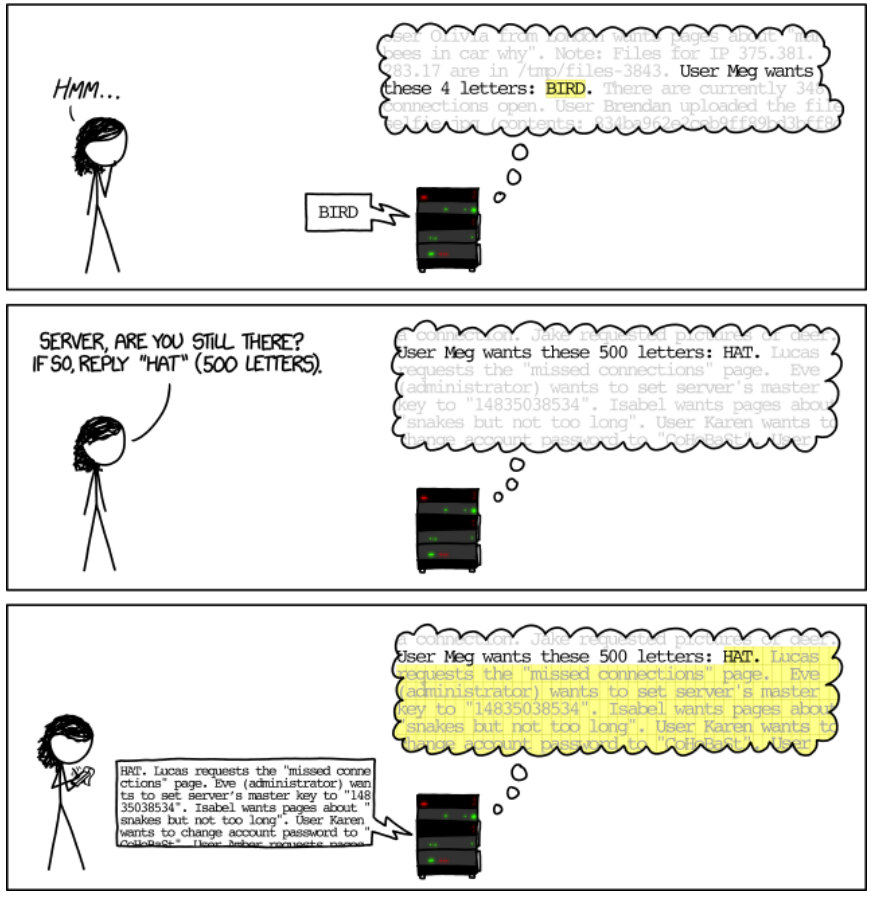

Figure 18: http://xkcd.com/1354/

case study: heartbleed

- heartbleed = catastrophic bug in OpenSSL
- published in Apr 2014 (now fixed)
- but what about legacy systems, critical infrastructure, embedded devices, IoT, ...
- attacker can request any number of times a copy of the content of up to 64kB of memory from a vulnerable server
- limited to process owning openssl, compromises server's private key, user keys, content being exchanged, ...
- furthermore: since free/malloc does not erase memory, content from other processes may be accessible
- comment from Theo de Raadt (openbsd): because on *some* platforms performance may suffer, protective wrappers around malloc/free (such as in libc for openbsd) were made ineffective by OpenSSL team...

NIST SP 800-64

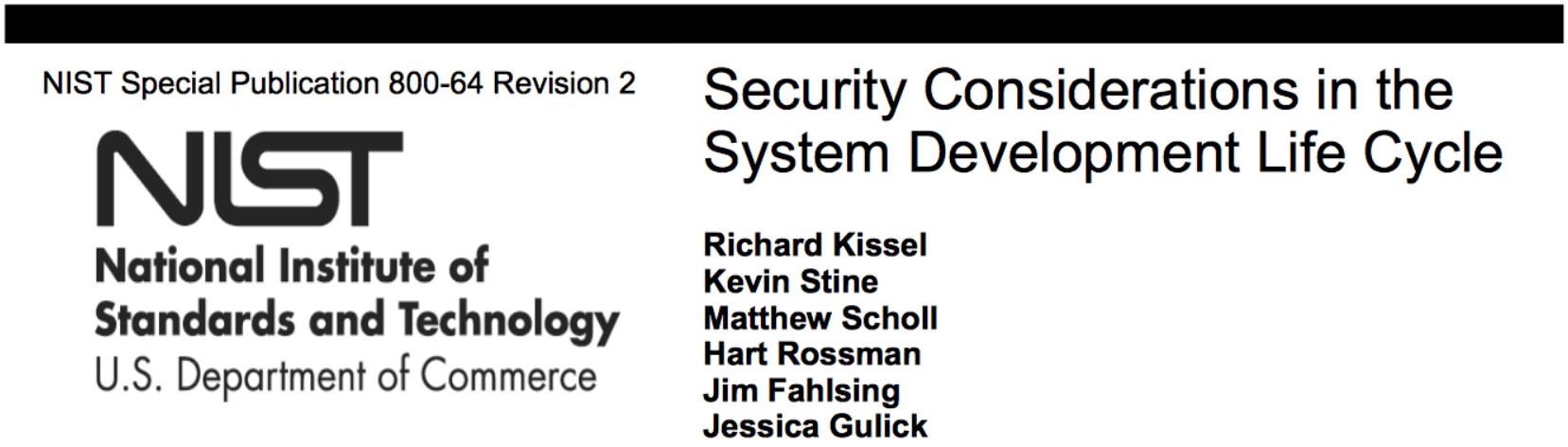

Figure 19: NIST

NIST SP 800-64

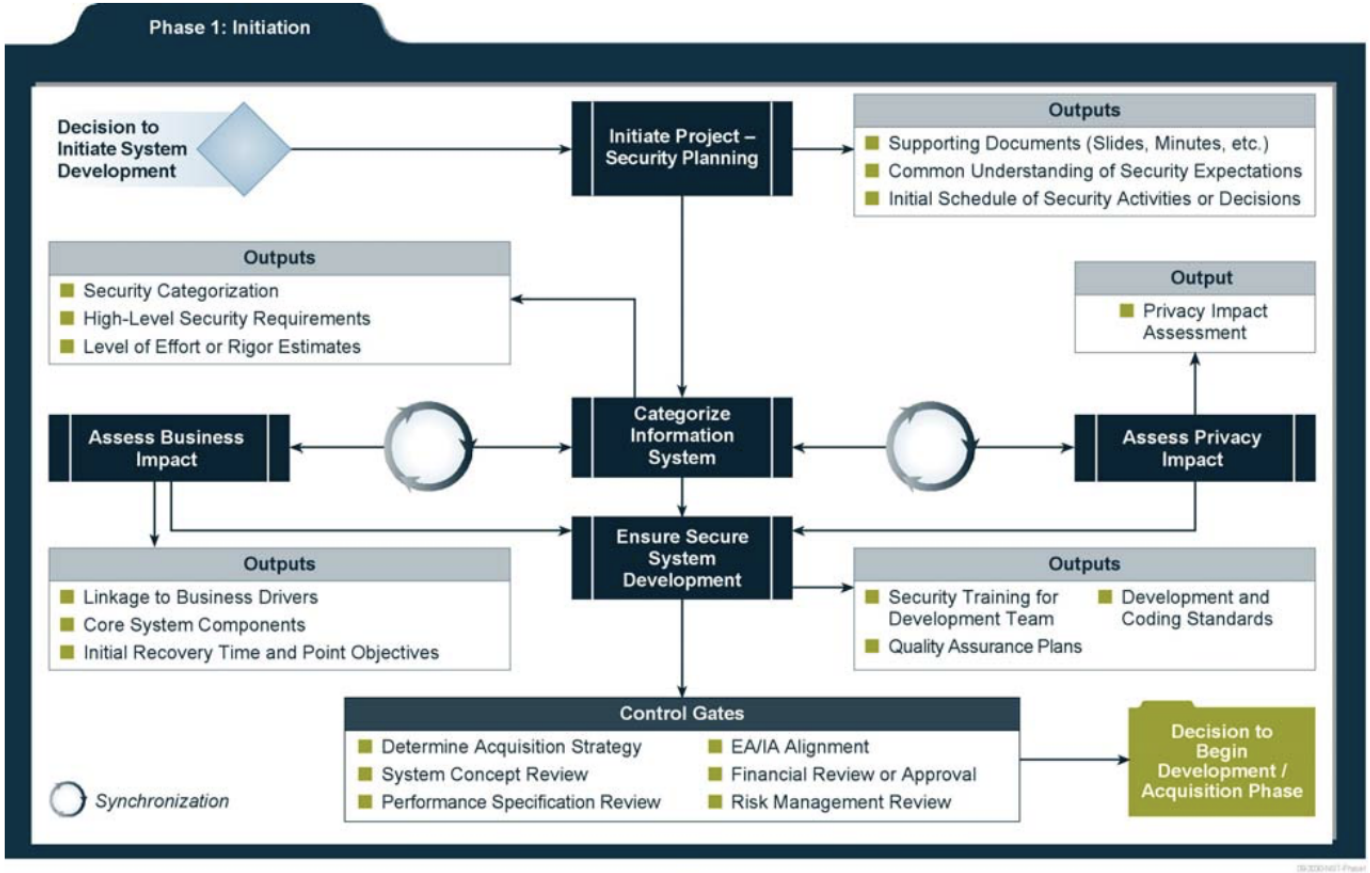

Figure 20: NIST

conclusions

conclusions

Figure 21: questions or comments ?