Software and Security Engineering

Alastair R. Beresford

With many thanks to Ross Anderson

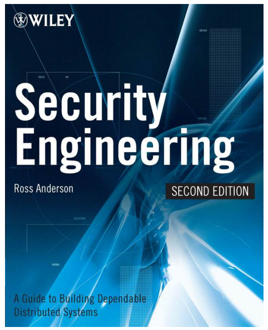
Aims

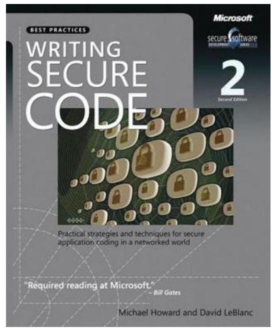
- Introduce software engineering with focus on:
  - Large systems
  - Safety-critical systems
  - Systems to withstand attack by capable opponents
- Illustrate what goes wrong
- Best practice to avoid failure

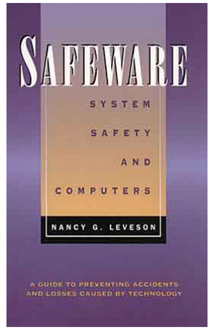
Objectives

- By the end of the course you should be able to:
  - Write programs with tough assurance targets
  - Work effectively as part of a team
- Understand
  - Software development models
  - Development lifecycle
  - Understand bugs, vulnerabilities and hazards

Books







Additional Reading

F.P. Brooks, *The Mythical Man Month*

J. Reason, *The Human Contribution*

S.W. Thames, *Report of the Inquiry into the London Ambulance Service*

S. Maguire, *Writing Solid Code*

H. Thimbleby, *Improving safety in medical devices and systems*

O. Campion-Awwad et al, *The National Programme for IT in the NHS – A Case History*

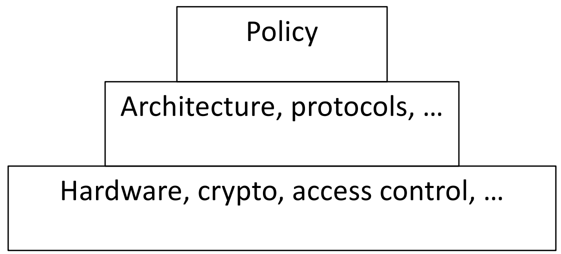
Course Outline – key topics

- Security policy
- Safety case
- Security protocols
- User behaviour
- Bugs
- Software crisis
- Development lifecycle
- Critical systems
- Testability
- Software-as-a-service

What is Security Engineering?

Security engineering is about building systems to remain dependable in the face of malice, error and mischance.

The Design Hierarchy



What are we trying to do?   How?   With what?

Security vs Dependability

Dependability = Reliability + Security

- Malice is different from error
- Reliability and security are often strongly correlated

A system can be…

- equipment or a component (laptop, smartcard, …)
- a collection of products, their operating systems, and some networking equipment
- The above plus applications
- The above plus internal staff
- The above plus external users

Common failure: policy drawn too narrowly

Electric bike should not propel bicycle when speed exceeds 15.5 mph

Subjects and principals

Subject: a physical person

Person: a subject or a legal person (firm)

Principal:

- A person
- Equipment
- A role, including complex roles

Secrecy and privacy

Secrecy: mechanism to control which principals can access information

Privacy: control of your own secrets

Confidentiality: an obligation to protect someone else's secrets.

Anonymity, integrity, authenticity

- *Anonymity*: restrict access to metadata
- *Integrity*: an object has not been altered since the last authorised modification
- *Authenticity* has two common meanings:
  - an object has integrity plus freshness
  - You are speaking to the right principal

Trust is hard; several meanings…

1. A warm fuzzy feeling
2. **A trusted system or component is one that can break my security policy**
3. A trusted system is one I can insure
4. A trusted system won't get me fired when it breaks
5. …

Errors, failures, reliability, accidents

- *Error*: a design flaw or deviation from intended state
- *Failure*: nonperformance of the system when inside specified environmental conditions
- *Reliability*: probability of failure within a specified period of time
- *Accident*: an undesired, unplanned event resulting in a specified kind or level of loss

Hazards and risks

- *Hazard*: a set of conditions in a system or its environment where failure can lead to an accident
- A *critical* system, process or component is one whose failure will lead to an accident
- *Risk* is the probability of an accident
  - Often combined with *unit of exposure*; e.g. a *micromort*
- Uncertainty is where the risk is not quantifiable
- Safety is simple: freedom from accidents

Security policy, profile, and target

- A *security policy* is a succinct statement of protection goals
- A *protection profile* is a detailed statement of protection goals
- A *security target* is a detailed statement of protection goals applied to a particular system

What often passes as 'policy'

1. This policy is approved by Management.
2. All staff shall obey this security policy.
3. Data shall be available only to those with a need-to-know.
4. All breaches of this policy shall be reported at once to Security.

What's wrong with this?

Traditional government approach

- Start from the threat model: an insider who is disloyal or careless.
- Solution: limit the number of people you trust, and make it harder for them to be untrustworthy

Basic idea since 1940: a clerk with 'Secret' clearance can read documents at 'Confidential' and 'Secret' but not at 'Top Secret'

Multilevel Secure Systems (MLS)

- Classify all documents and data with a level, such as *official*, *secret*, *top secret*; or *high* and *low*.
- Principals have clearances; clearance must equal or exceed classification of any documents viewed.
- Enforce handling rules for material at each level.
- Information flows upwards only:
  - No read up
  - No write down

Bell-LaPadula formal model

- Bell-LaPadula (1973):
  - *simple security policy* (no read up)
  - **-policy* (no write down)
- With these two rules, one can prove that a system that starts in a secure state will remain in one
- Aim is to minimise the Trusted Computing Base

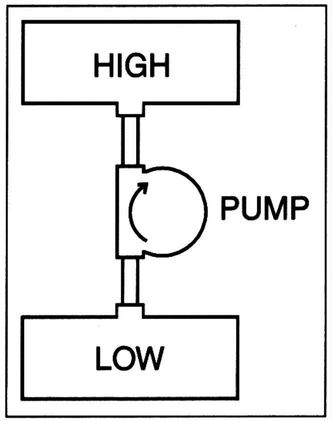
Covert channels cause havoc

- BLP lets malware move from Low to High, just not to signal down again.
- What if malware at High modulates shared resource (e.g. CPU usage) to signal to Low?
- How can you let message traffic pass from Low to High, if any acknowledgement of receipt could be delayed and used to signal?

Moral: covert channel bandwidth is a complex.
It's an emergent property of whole systems!

High assurance MLS system

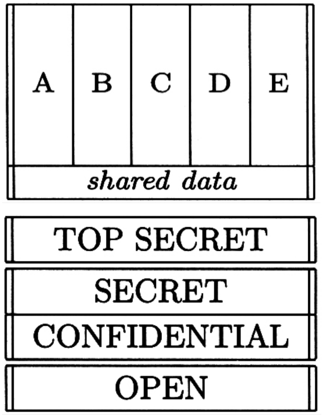
- The pump simplifies the problem: replace the complex emergent property of the whole system with a simple property of a testable component
- Nevertheless, often harder than it looks!



Multilateral Security

Stop lateral flow, examples:

- Intelligence, typically with compartments
- Medical records
- Competing clients of an accounting firm



Biba formal model for integrity

- Biba (1975)
  - Simple integrity policy (no read down)
  - *-integrity policy (no write up)
- Dual of the Bell-LaPadula model
- Examples:
  - Medical devices with *calibrate* and *operate* modes
  - Electricity grid controls with safety at the highest level, operational control as the next, and so on.

Architecture matters

- Lots of legacy protocols trust all network nodes
- Chrysler Jeep recall
- Defence in depth: separate subnets, capable firewalls,

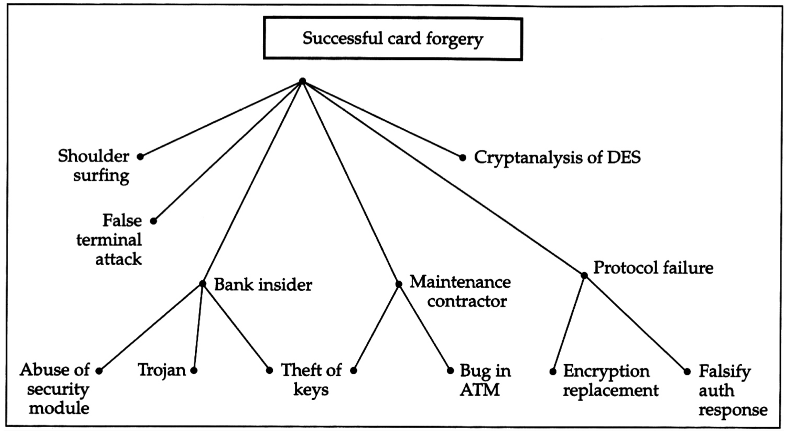
Safety policies

- Industries have their own standards, cultures, often with architectural assumptions embedded in component design
- Plethora of safety legislation
- Sometimes brand new standards, but in more mature industries safety standards tend to evolve
- Two basic ways to evolve:
  - *failure modes and effects analysis*
  - *fault tree analysis*

Failure modes and effects analysis

- Look at each component and list failure modes
- Figure out what to do about each failure
  - Reduce risk by overdesign?
  - Redundancy?
  - …
- Use secondary mechanisms to deal with interactions
- Developed by NASA

Fault tree analysis



Work backwards from bad outcome we must avoid to identify critical components

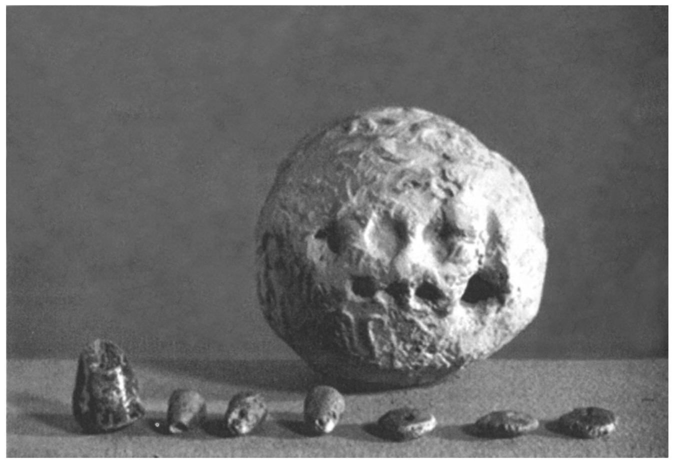
Example: nuclear bomb safety

Don’t want Armageddon caused by a rogue pilot, a stolen bomb, or a mad president, so use independent, simple, technical mechanisms

- Authorisation: president releases code
- Intent: pilot puts key in bomb release
- Environment: N seconds zero gravity

Independent, simple, technical mechanisms

Bookkeeping, 8-4th millennium BCE

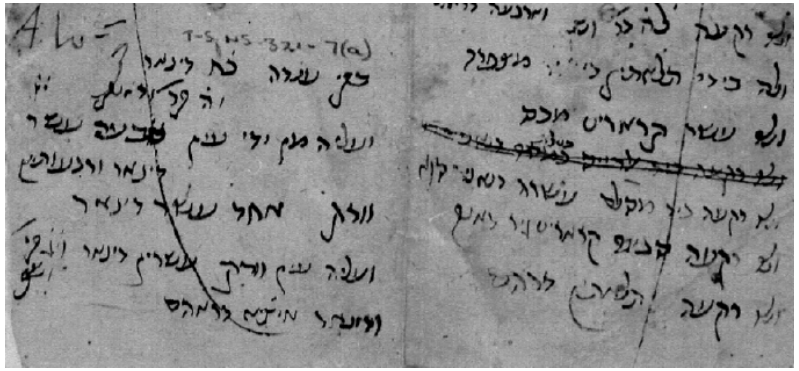


Bookkeeping, circa 1100 AD

- Double-entry bookkeeping: each entry in one ledger is matched by opposite entries in another
- Ensure each ledger is maintained by a different subject so bookkeepers have to collude to defraud

- Example: a firm sells £100 of goods on credit, so credit the sales account, debit the receivables account. Customer subsequently pays, so credit the receivables account, debit the cash account.

Double-entry bookkeeping found in the Genizah Collection

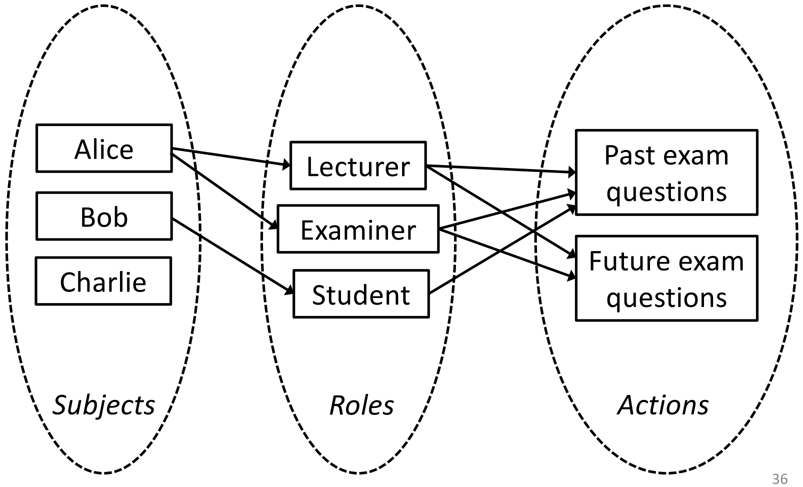


Separation of duties in practice

- Serial:
  - Lecturer gets money from EPSRC, charity, …
  - Lecturer gets Old Schools to register supplier
  - Gets stores to sign order form and send to supplier
  - Stores receives goods; Accounts gets invoice
  - Accounts checks delivery and tell Old Schools to pay
  - Lecturer gets statement of money left on grant
  - Audit by grant giver, university, …

- Parallel: authorization from two distinct subjects

Role-Based Access Control (RBAC) decouples policy and mechanism



Swiss Cheese Model

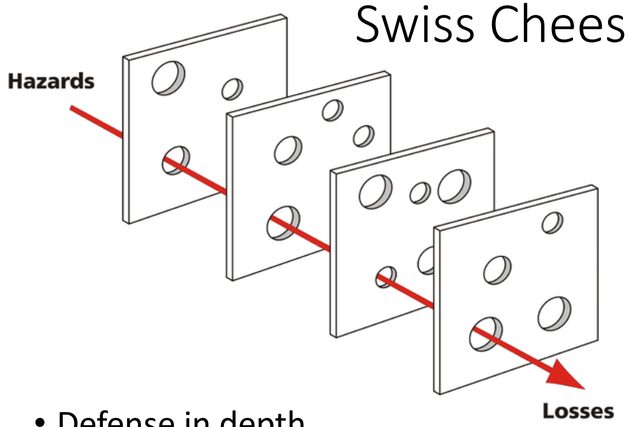


Diagram by Davidmack CC-BY-SA 3.0

- Defense in depth
- Layers could include hardware, software, policy, human factors, etc.

Summary of security and safety

- What are we trying to do?
- Security: threat model, security policy
- Safety: hazard analysis, safety standard
- Refine to protection profile, safety case
- Typical mechanisms: usability engineering, firewalls, protocols, access controls, …

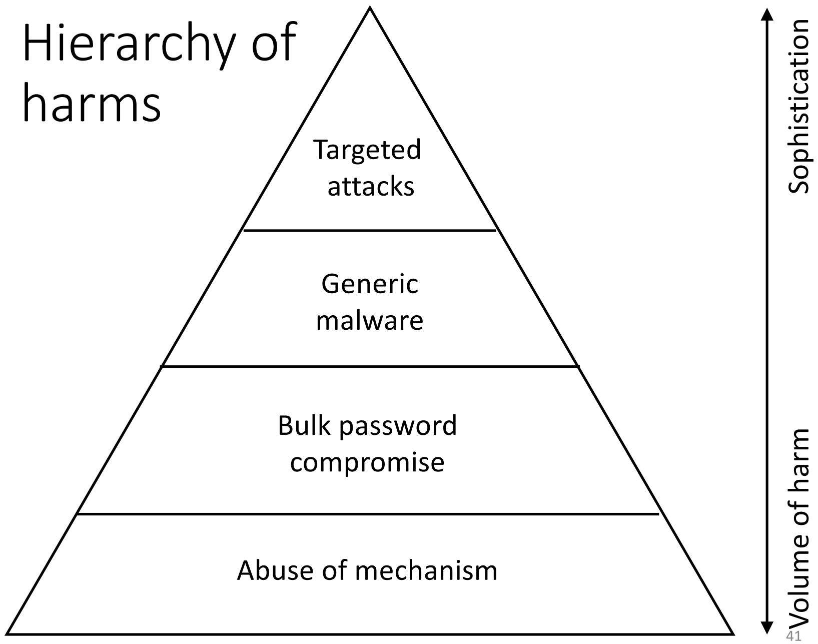
Do not ignore user behaviour

- Many systems fail because users make mistakes
- Banks routinely tell victims of fraud "our systems are secure so it must be your fault"
- Most car crashes are user error; yet we now build cars with crumple zones

Chevrolet 1959 vs 2009

https://www.youtube.com/watch?v=fPF4fBGNK0U

Hierarchy of harms



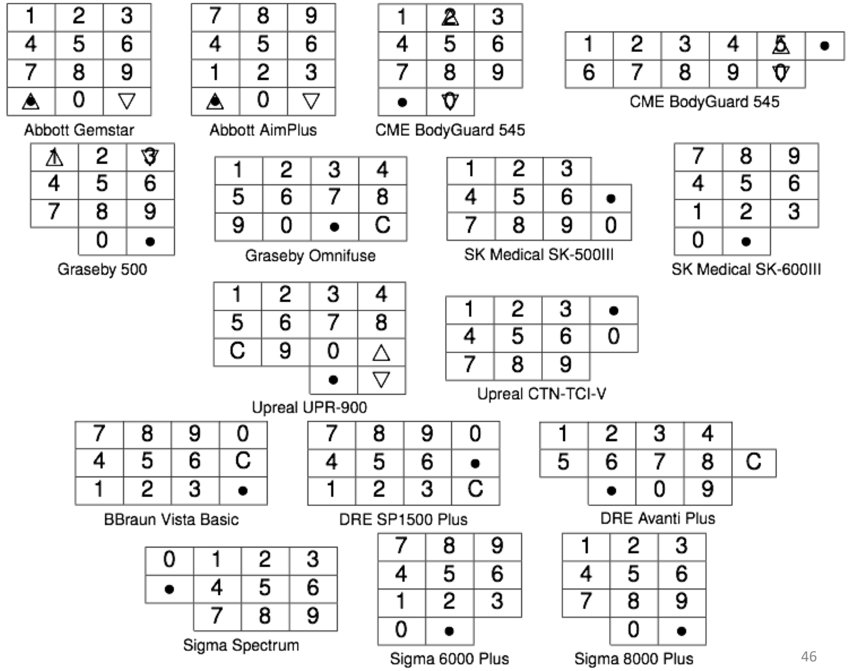
Many abuses of mechanism

- Cyberbullying
- Doxing
- Fake rental apartments
- …

How can we protect against these attacks?

Useable privacy is also hard

- Traditional approaches – anonymisation and consent – are really hard to deliver
- Problem gets harder as systems get larger
- Automated data collection (e.g. from sensors) makes the situation more difficult again



Medical device safety

- Usability problems with medical devices kill about the same number of people as cars do
- Biggest killer nowadays: infusion pumps
- Nurses typically get blamed, not vendors
- Avionics are safer, as incentives are more concentrated
- Read Harold Thimbleby's paper!

Bulk password compromise

- Example: in June 2012, 6.5m LinkedIn passwords stolen, cracked (encryption did not have a salt) and posted on a Russian forum
  - Method: SQL injection (see later)
  - Passwords were reused on other sites, from mail services to PayPal.
  - Reused passwords were used on those third-party sites
- There have been many, many such exploits!
- What can we do about password reuse?

Phishing and social engineering

- Card thieves call victims to ask for PINs
- A well-crafted email sent to company staff, with apparently authority, can get 30% yield
- Some big consequences (see next)
- Think like a crook (see Mitnick reading)

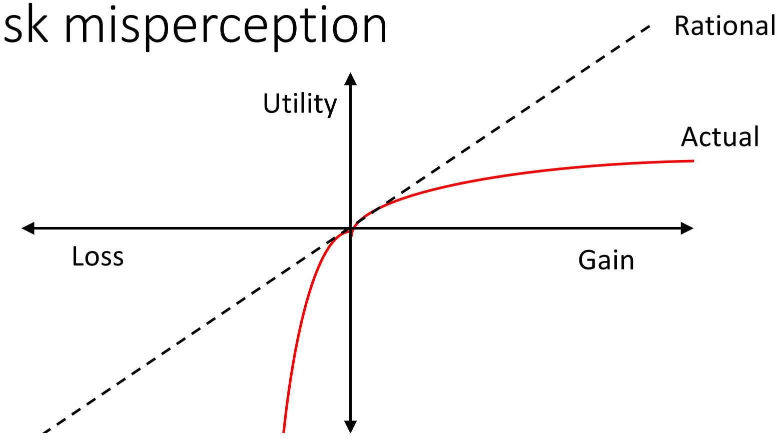
John Podesta email compromise by Fancy Bear (allegedly Russia)

- White House chief-of-staff; chair of Hiliary Clinton's 2016 US Presidential Campaign
- Gmail account was compromised
- 20,000 emails subsequently published by WikiLeaks
- Authenticity of some emails questioned

Cognitive factors

- Many errors arise from our highly adaptive mental processes
  - We deal with novel problems in a conscious way
  - Frequently encountered problems are dealt with using rules we evolve, and are partly automatic
  - Over time, the rules give way to skill
- Our ability to automate routine actions leads to absent-minded slips, or following the wrong rule
- There are also systematic limits to rationality in problem solving – so called *heuristics* and *biases*

Risk misperception



People offered £10 or a 50% chance of £20 usually prefer the former; if offered a loss of £10 or a 50% chance of a loss of £20 they tend to prefer the latter!

Framing decisions about risk, or the *Asian disease problem*

Scenario A, choose between:

a) “200 lives will be saved”

b) “with p=1/3, 600 saved; with p=2/3, none saved”

Here 72% choose (a) over (b).

Scenario B, choose between:

1) “400 will die”

2) “with p =1/3, no-one will die, p=2/3, 600 will die”

Here 78% prefer (2) over (1)

Social psychology

- Authority matters: Milgram showed over 60% of all subjects would torture a ‘student’
- The herd matters: Asch showed most people could deny obvious facts to please others
- Reciprocation is built-in: give a gift, to increase your chance of receiving one

Fraud psychology

All the above plus:

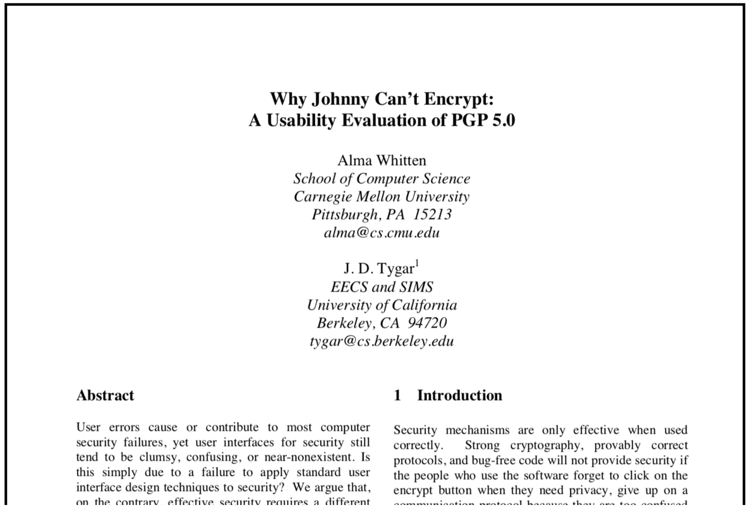
- Appeal to the mark’s kindness
- Appeal to the mark’s dishonesty
- Distract them so they act automatically
- Arouse them so they act viscerally

People only follow advice which confirms their own world view

- Users have different mental models. Explore how your users see the problem – the ‘folk beliefs’
- Given a model of their world view, target approach to appeal to it.

57

Affordances: Johnny Can’t Encrypt



Why Johnny Can’t Encrypt:
A Usability Evaluation of PGP 5.0

Alma Whitten
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
alma@cs.cmu.edu

J. D. Tygar[1]
EECS and SIMS
University of California
Berkeley, CA 94720
tygar@cs.berkeley.edu

Abstract

User errors cause or contribute to most computer security failures, yet user interfaces for security still tend to be clumsy, confusing, or near-nonexistent. Is this simply due to a failure to apply standard user interface design techniques to security? We argue that, on the contrary, effective security requires a different

1 Introduction

Security mechanisms are only effective when used correctly. Strong cryptography, provably correct protocols, and bug-free code will not provide security if the people who use the software forget to click on the encrypt button when they need privacy, give up on a

58

The power of default

Most people don’t opt in or out; they go with default

Can exploit this for good (or evil):

- Pensions
- Privacy settings in an online service
- Use of crypto
- …

Therefore defaults may be contentious

59

Economics versus psychology

Most people don’t worry enough about computer security, and worry too much about terrorism

How could we fix this, and why is it not likely to be?

60

The compliance budget

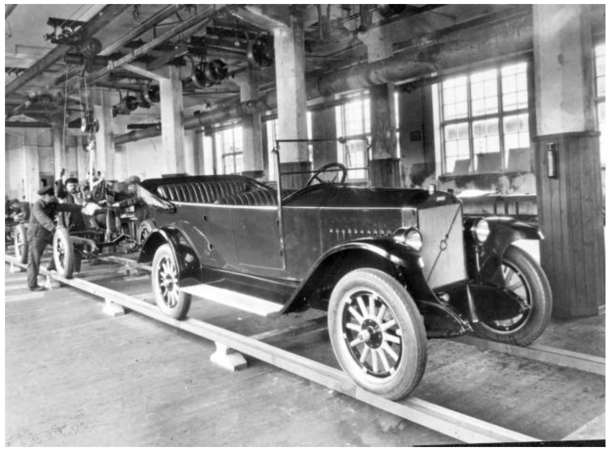
- ‘Blame and train’ as an approach is suboptimal
- It’s often rational to ignore warnings
- People will spend only so much time obeying rules, so choose the rules that matter
- Violations of rules also matter: they’re often an easier way of working, and sometimes necessary
- The ‘right’ way of working should be easiest: look where people walk, and lay the path there

Where should the path be?

Differences between people

- Ability to perform certain tasks varies widely across subgroups of the population, including by
  - Age
  - Gender
  - Education
  - …
- Yet all customers receive complex password rules and anti-phishing advice

More accidents with Volvos?



Volvo ÖV 4, April 1927

Understanding error helps us build better systems

- Significant psychology research into errors
- Slips and lapses
  - Forgetting plans, intentions (strong habit intrusion)
  - Misidentifying objects, signals
  - Retrieval failures ("its on the tip of my tongue")
  - Premature exits from action sequences (using the ATM)
- Rule-based mistakes; applying the wrong procedure
- Knowledge-based mistakes; heuristics and biases

Training and practice reduce errors

Inexplicable errors, stress free, right cues	10^{-5}
Regularly performed simple tasks, low stress	10^{-4}
Complex tasks, little time, some cues needed	10^{-3}
Unfamiliar task dependent on situation, memory	10^{-2}
Highly complex task, much stress	10^{-1}
Creative thinking, unfamiliar complex operations, time short & stress high	~1

Passwords are cheap, but…

- Will users enter passwords correctly?
- Will they remember them?
- Will they choose a strong password?
- Will the write them down?
- Will the password be different in each context?
- Can the user be tricked into revealing passwords?

User studies are important

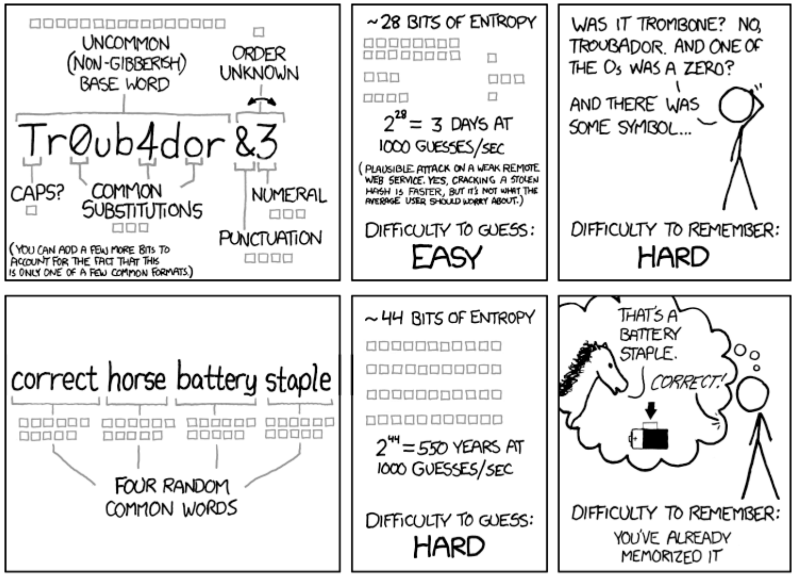
Experiment to see if first-year NatScis could be trained to use passwords effectively. Three groups:

- Control group of 100 (+100 more observed)
- Green group: use a memorable phrase
- Yellow group: choose 8 chars at random

Expected strength: Y > G > C; got Y = G > C

Expected resets: Y > G > C; got Y = G = C

We had 10% non-compliance



Hardware and online support to limit brute force is challenging

- Online services and tamperproof hardware can be used to limit brute-force guessing, such as
  - Bank card PIN (3 guesses on card; 3 online)
  - iPhone PIN (timeouts)
  - Login attempts to webservices (timeouts; care required)
  - …

If the typical person has five cards with the same PIN, how many wallets do you need to find before you get lucky?

Mitigate worst effects of a stolen password file

- Use key stretching techniques such as PDBKF2:

```
public PBEKeySpec(char[] password, byte[] salt,
        int iterCount, int keyLength)
```

- Establish breach reporting laws
- Externalise the problem with Oauth
- Use other factors to determine whether login legit

Password recovery is a weak point

- Password recovery often involves basic info which doesn't change:
  - What was the name of your first school?
  - What was the name of your first pet?
  - …
- Little ability to change this information
- Accounts for public figures are especially vulnerable

Externalities need consideration

- One firm's action has side-effects for others
- Password sharing a conspicuous example; we have to enter credentials everywhere
- Everyone wants recovery questions too
- Many firms train customers in unsafe behaviour from clicking on external links or redirecting the browser to third-party domains for payment
- Much 'training' amounts to victim blaming

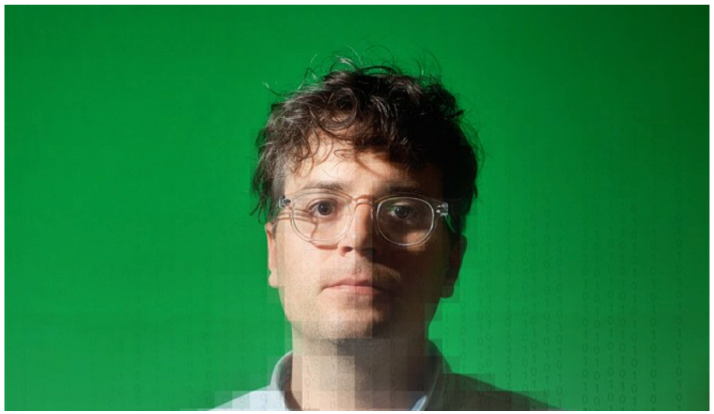
Iterative guessing of card details with botnet on websites works

- Of Alexa top 500 websites, 26 use Primary Account Number (PAN) and expiry date
- 37 use PAN + postcode (numeric digits only for some, add door number for others)
- 291 ask for PAN, expiry date and CVV2

There is enough variation in requirements across websites that you can iteratively generate valid credentials

MAT HONAN GEAR 08.06.12 08:01 PM

HOW APPLE AND AMAZON SECURITY FLAWS LED TO MY EPIC HACKING



Amazon ⇢ Apple ID ⇢ Gmail ⇢ Twitter

(And all they wanted was his three letter Twitter handle!)

- Twitter: find personal website, then Gmail, home address
- Gmail: account recovery gave "m••••n@me.com"
- Amazon: call with name, address, email to associate a new credit card number (fake) to the account
- Amazon: call (again) with name, address, credit card number and associate new email address with the account
- Amazon: Use web password reset to new email address; get last four digits of all credit cards in the account
- Apple: Call with billing address and last four digits credit card to get temp password for "m••••n@me.com"
- Gmail: reset password sent to "m••••n@me.com"
- Twitter: reset password sent to Gmail

Security protocols

- Security protocols are another intellectual core of security engineering
- They are where cryptography and system mechanisms (such as access control) meet
- They introduce an important abstraction, and illustrate adversarial thinking
- They often implement policy directly
- And they are much older then computers…

Ordering wine in a restaurant

1. Sommelier presents wine list to host
2. Host chooses wine; sommelier fetches it
3. Host samples wine; then it's served to guests

Security properties?

Car unlocking protocols

Static	Non-interactive	Interactive
$T \rightarrow E: K$	$T \rightarrow E: T, \{T,N\}_K$	$E \rightarrow T: N$ $T \rightarrow E: \{T,N\}_K$

N: *nonce;* a sequence number, random number or timestamp

E: engine unit

T: car key fob or *transponder*

K: secret key shared between E and T

$\{x\}_K$: encrypt *x* with K

Identify Friend or Foe (IFF)

- Basic idea: fighter challenges bomber
  - $F \rightarrow B: N$
  - $B \rightarrow F: \{N\}_K$
- What can go wrong?

Person-in-the-middle attack…

- Basic idea: fighter challenges bomber

  F → B: N

  B → F: $\{N\}_K$

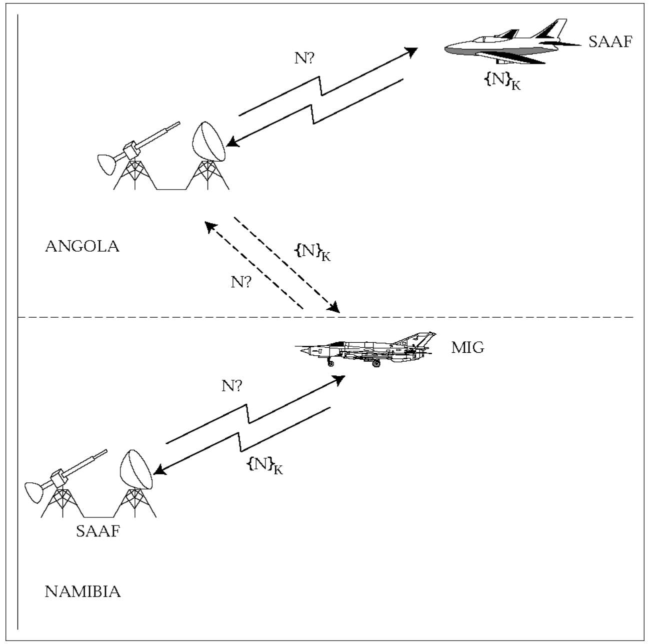
- What if the bomber reflects the challenge back at the fighter's wingman?

  F → B: N

  B → *F*: N

  F → B: $\{N\}_K$

  B → F: $\{N\}_K$



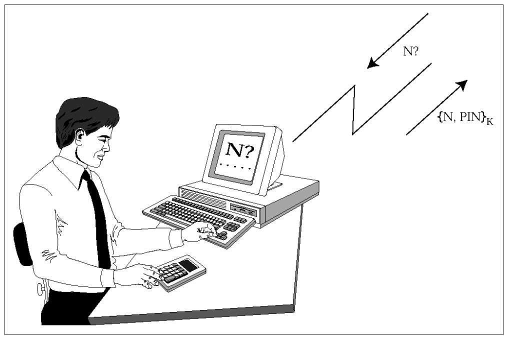
Two-factor authentication (2FA)

T → U: N

U → C: N, PIN

C → U: $\{N, PIN\}_K$

U → T: $\{N, PIN\}_K$



T: terminal — U: user

C: calculator — K: key known to bank and P

PIN: secret known to bank and U

Card authentication protocol

- Allows EMV cards to be used in online banking
- Users compute codes for access, authorisation
- A good design would take PIN and challenge / data, encrypt to get response
- But the UK one first tells you if the PIN is correct
- What can go wrong with this?

Alice and Bob want to talk. They each share a key with Sam. How?

- Alice contacts Sam and asks for a key for Bob
- Sam sends Alice a key encrypted in a blob only she can read, and the same key also encrypted in another blob only Bob can read
- Alice calls Bob and sends him the second blob

How can they check the protocol's fresh?

Kerberos uses tickets to support communication between parties

$A \rightarrow S$: A, B
$S \rightarrow A$: $\{T_S, L, K_{AB}, B, \{T_S, L, K_{AB}, A\}_{KBS}\}_{KAS}$
$A \rightarrow B$: $\{T_S, L, K_{AB}, A\}_{KBS}, \{A, T_A\}_{KAB}$
$B \rightarrow A$: $\{T_A+1\}_{KAB}$

A: Alice — B: Resource (e.g. printer)
S: Server — T_S: Server timestamp
K_{AS}: Secret key shared between A and S
K_{BS}: Secret key shared between B and S
K_{AB}: Shared session key for A and B
L: Lifetime of the session key

Europay-Mastercard-Visa (EMV) How might you attack this?

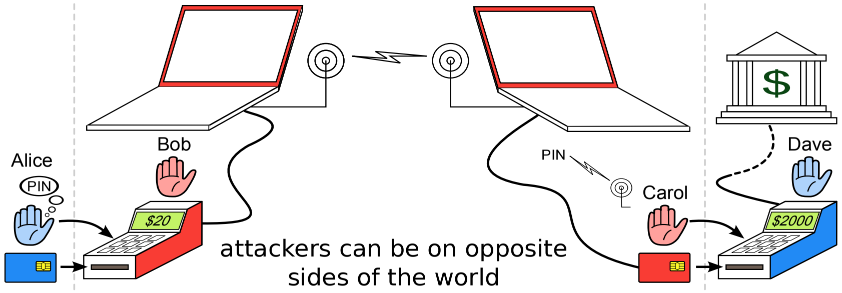
$C \rightarrow M$: $sig_B\{C, card\_data\}$
$M \rightarrow C$: N, date, Amt, PIN (if PIN used)
$C \rightarrow M$: $\{N, date, Amt, trans\_data\}_{KCB}$
$M \rightarrow B$: $\{\{N, date, Amt, trans\_data\}_{KCB}, trans\_data\}_{KMB}$
$B \rightarrow M \rightarrow C$: $\{OK\}_{KCB}$

C: Card — $sig_Y\{x\}$: message x digisigned by Y
M: Merchant — $\{x\}_K$: Message x encrypted under K
B: Bank — K_{XY}: Shared key between X and Y

Replace insides of the terminal with your own electronics

- Capture card details and PINs from victims
- Use to perform man-in-the-middle attack in real time on a remote terminal in a merchant selling expensive goods

The relay attack: unstoppable but unrealistic – too hard to scale



Magstripe fraud is scalable

Photo credit: Brian Krebs, krebsonsecurity.com

- Install fake terminal and collect card data and PINs
- Either physically or wirelessly collect data

The no-PIN attack (2010)

C → M: sig_B\{C, exp\}
M → Ć: N, date, Amt, PIN_request
Ć → C: N, date, Amt
C → M: \{N, date, Amt, trans_data\}$_{KCB}$
M → B: \{\{N, date, Amt, trans_data\}$_{KCB}$, trans_data'\}$_{KMB}$
B → M: \{OK\}$_{KCB}$

Ć: MITM card shim
C: Card — $sig_Y\{x\}$: message *x* digisigned by *Y*
M: Merchant — $\{x\}_K$: Message x encrypted under *K*
B: Bank — K_{XY}: Shared key between *X* and *Y*

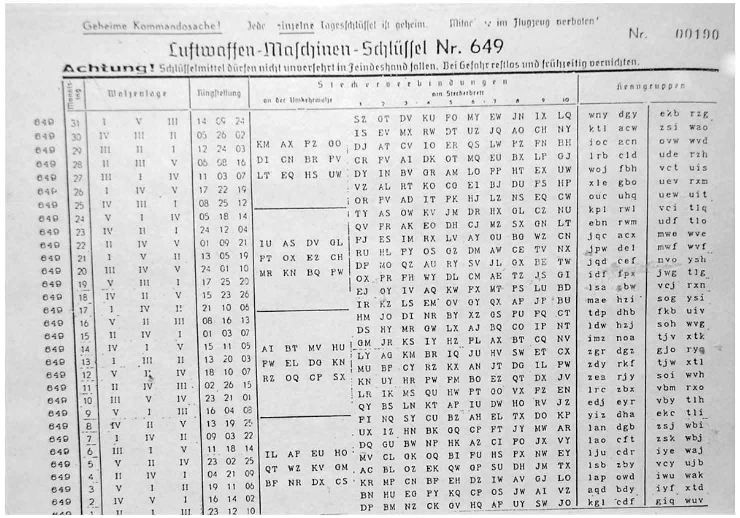
Fixing the no-PIN attack: simpler protocol required

- In theory might compare card data with terminal data at terminal, acquirer, or issuer
- In practice has to be the issuer since incentives for terminal and acquirer are poor
- Barclays introduced a fix July 2010; removed December 2010. Banks asked for student thesis to be taken down from web instead.
- Eventually fixed for UK transactions in 2016
- Real problem: EMV spec now far too complex

The preplay attack (2014)

- In EMV, the terminal sends a random number N to the card along with the date d and the amount Amt
- The card authenticates N, d and Amt using the key it shares with the bank, K_{CB}
- What happens if I can predict N for date d?
- Answer: if I have access to your card I can precompute an authenticator for Amt and d

Symmetric key cryptography requires careful sharing of keys



Public key cryptography

Allows two parties with no prior knowledge of each other to jointly establish a shared secret key over an insecure channel

Examples include Diffie-Hellman and RSA

Diffie Hellman revision

Alice and Bob publicly agree to use p = 23, g = 5

1. Alice chooses secret integer a = 4, then $A \rightarrow B$: $g^a \bmod p = 5^4 \bmod 23 = 4$
2. Bob chooses secret integer b = 3, then $B \rightarrow A$: $g^b \bmod p = 3^4 \bmod 23 = 10$
3. Alice computes $10^4 \bmod 23 = 18$
4. Bob computes $4^3 \bmod 23 = 18$

Alice and Bob now agree the secret integer is 18

Example derived from https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange

Physical public key crypto with locks

- Anthony sends a message in a box to Brutus. Since the messenger is loyal to Caesar, Anthony puts a padlock on it
- Brutus adds his own padlock and sends it back to Anthony
- Anthony removes his padlock and sends it to Brutus, who can now unlock it

Is this secure?

Asymmetric public-key crypto

- Separate keys for encryption and decryption
- Publish *encryption* key widely (the "public key") allowing anyone to create an encrypted message; only holder of *decryption* key ("private key") can decode the message and read it
- Digital signatures are the other way around: only you can sign but anyone can verify

Public-key Needham-Shroeder

- Proposed in 1978:

  $A \rightarrow B: \{N_A, A\}_{KB}$
  $B \rightarrow A: \{N_A, N_B\}_{KA}$
  $A \rightarrow B: \{N_B\}_{KB}$

- N_A and N_B are nonces generated by A and B respectively
- K_A and K_B are public keys for A and B respectively
- The idea is to use $N_A \oplus N_B$ as a shared key

Is this okay?

MITM attack found 18 years later

$A \rightarrow C: \{N_A, A\}_{KC}$

$C \rightarrow B: \quad \{N_A, A\}_{KB}$

$B \rightarrow C: \quad \{N_A, N_B\}_{KA}$

$C \rightarrow A: \{N_A, N_B\}_{KA}$

$A \rightarrow C: \{N_B\}_{KC}$

$C \rightarrow B: \quad \{N_B\}_{KB}$

The fix is explicitness. Put all names in all messages.

Binding keys to principals is hard

- Physically install binding on machines
  - IPSEC, SSH
- Trust on first use; optionally verify later
  - SSH, Signal, simple Bluetooth pairing
- Use certificates with trusted certificate authority
  - Sam signs certificate to bind Alice's key with her name
  - Certificate = sig_S\{A, K_A , Timestamp, Length\}
  - Basis of Transport Layer Security (TLS) as used in HTTPS
- Use certificate pinning inside an app
  - Used by some smartphone apps

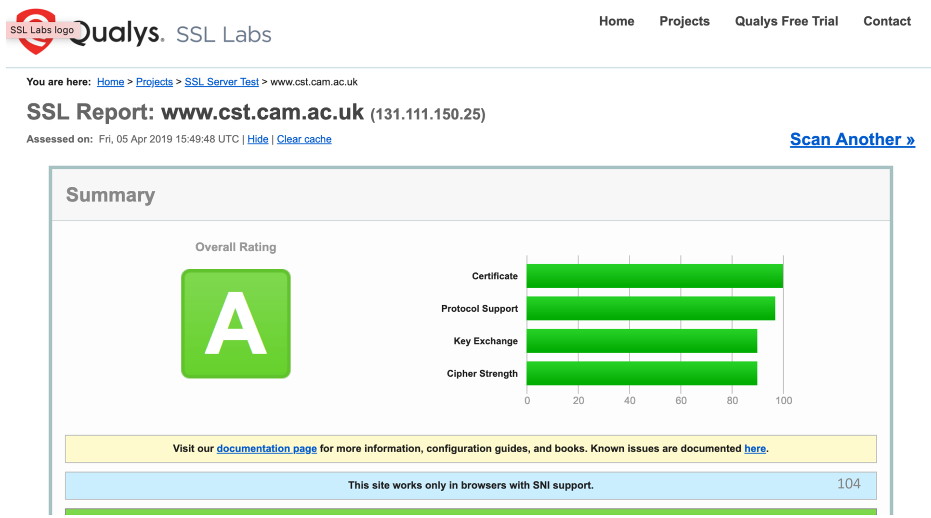
Transport Layer Security (TLS)

- Uses public key cryptography and certificates to establish a secure channel between two machines
- Protocol proven correct (Paulson, 1999)
- Yet, the protocol is broken annually
- Often a large number of root certificate authorities. Are these all trustworthy?

DigiNotar went bust after issuing bogus certificates

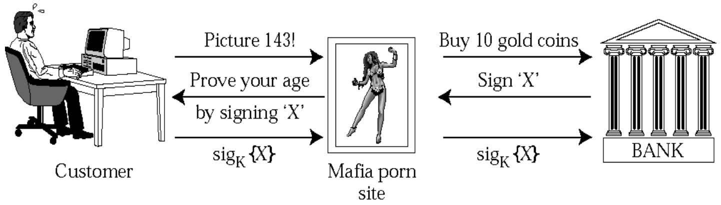
- Dutch certificate authority
- More than 300,000 Iranian Gmail users targeted
- More than 500 fake certificates issued
- Major web browsers blacklisted all DigiNotar certs

TLS security landscape is complex



Chosen protocol attack

The Mafia asks people to sign a random challenge as proof of age for porn sites!



Bugs are found in and around code

- Bugs in the code
  - Arithmetic
  - Syntactic
  - Logic
  - Concurrency
- Bugs around the code
  - Code injection
  - Usability traps

Patriot missile failures in Gulf War I

German Air Force; CC-BY-SA, Darkone, Wikipedia

Afgan National Army; PD, Davric, Wikipedia

- Failed to intercept an Iraqi Scud missile in first Gulf War on 25th February 1991
- Scud struck US barracks in Dhahran; 28 dead
- Other Scuds hit Saudi Arabia, Israel

Caused by arithmetic bug

- System measured time in 1/10 sec, truncated from $0.0001100110011..._b$
- Accuracy upgraded as system upgraded from air-defence to anti-ballistic-missile defence
- Code not upgraded everywhere (assembly)
- Modules out by 1/3rd sec after 100h operation
- Not found in testing as spec only called for 4h tests

Lesson: Critical system failures are typically multifactorial

Syntactic bugs arise from features of the specific language

For example, in Java:

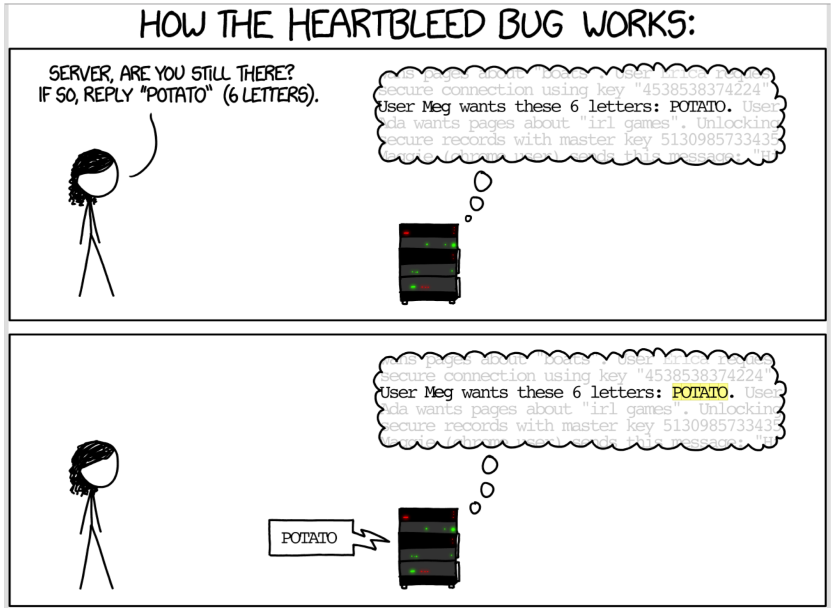
`1 + 2 + ""` evaluates to `"3"`

`"" + 1 + 2` evaluates to `"12"`

This is due to coercion from primitive integers to `java.lang.String`

Apple's goto fail bug (2014)

```
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;
}
```

Credit: https://xkcd.com/1354/

Heartbleed allows clients to read the contents of server memory

Therefore a malicious client could read:

- Secret keys of any TLS certificates used by server
- User creds such as email address and passwords
- Confidential business documents
- Personal data

The attack left no trace of use in server logs

Notification and clean-up difficult

12th March 2012	Bug introduced (OpenSSL 1.0.1)
1st April 2014	Google secretly reports vuln
3rd April 2014	Codenomicon reports vuln
7th April 2014	Fix released
7th April 2014	Public announcement
9th May 2014	57% of website still using old TLS certificates
20th May 2014	1.5% of 800,000 most popular websites still vulnerable

Intel AMT Bug

- AMT allows sysadmins remote access to a machine, even when turned off (but mains power on)
- Provides full access to machine, independent of OS
- A sketch of the protocol for authentication between machine and remote party is as follows:

C $\rightarrow$ S: "Hi. I'd like to connect"

S $\rightarrow$ C: "Please encrypt *X* with our secret key"

C $\rightarrow$ S: "Here are the first *x* bytes of $\{X\}_{KCS}$"

Concurrency bug: time of check to time of use failure (TOCTOU)

Check

```
…
File file = new File(args[0]);
if(!file.canWrite())
    return;
```

Use

```
RandomAccessFile fp = new
         RandomAccessFile(file, "rw");
fp.writeChars("Some replacement text");
fp.close();
…
```

Adapted example from https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

Clallam Bay Jail inmates perform code injection on payphones

1. Inmate typed in the number they wished to call
2. Inmate selected whether the recipient spoke Spanish or English
3. Inmate was asked to say their name, "Eve", say
4. The phone then dialled the number and read out a recorded message in chosen language and appended inmate name to the end:

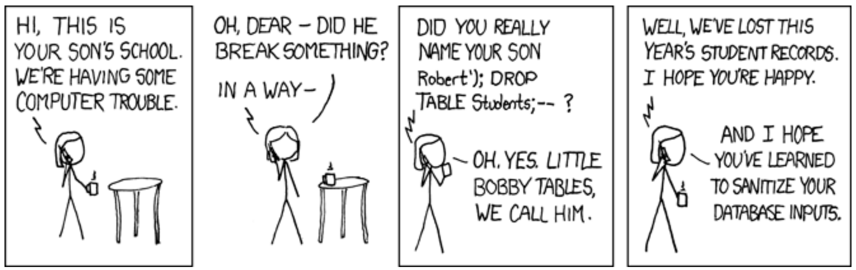
> "An inmate from Clallam Jail wishes to speak with you. Press three to accept the collect call charges. The inmate's name is" … "Eve"

Okay Google, what's a Whopper?

The Morris Worm: breaking into computers at scale (1988)

- Exploited vulnerabilities in sendmail, fingerd, rsh
- Used a list of common weak passwords
- Gov assessment: $100k to $10M in damage
- 6,000* machines infected
- Internet partitioned for days to prevent reinfection
- Robert Morris was the first person convicted under the 1986 Computer Fraud and Misuse Act.
  - 3 year suspended sentence
  - 400 hr community service
  - $10k fine.

SQL Injection attack: failure to sanitize untrusted inputs




```
String sql =
  "INSERT INTO Students (Name) VALUES ('"
  + studentName
  + "');";
```

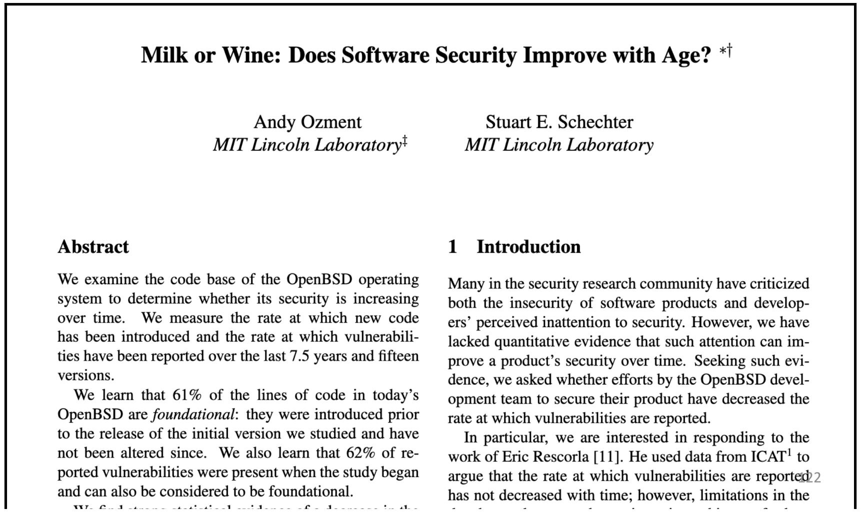

Software countermeasures: systems and tools

- Operating system protections
  - Data execution prevention
  - Address space layout randomisation
  - …
- Tools, e.g. Coverity
  - Static analysis
  - Dynamic analysis
  - Testing frameworks
  - …
- Automated update systems to install patches

Software countermeasures: reducing bug number and severity

- Defensive programming
- Secure coding standards
  - See Howard and LeBlanc on MS standards for C
- Contracts, e.g. in the Eiffel language
- API analysis
  - Combining API calls may lead to vulnerabilities
  - Challenging for APIs accessible over the Internet

We cannot write code without latent vulnerabilities



Milk or Wine: Does Software Security Improve with Age? *†

Andy Ozment, *MIT Lincoln Laboratory*‡
Stuart E. Schechter, *MIT Lincoln Laboratory*

Abstract

We examine the code base of the OpenBSD operating system to determine whether its security is increasing over time. We measure the rate at which new code has been introduced and the rate at which vulnerabilities have been reported over the last 7.5 years and fifteen versions.

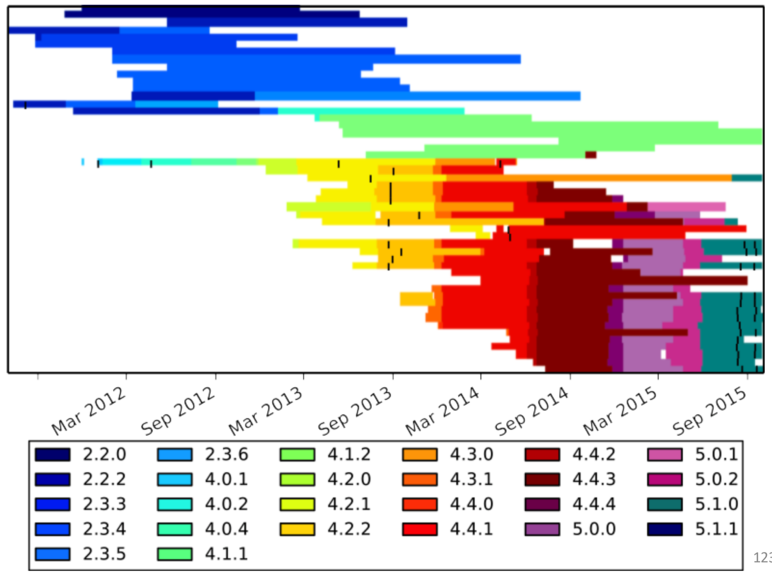
We learn that 61% of the lines of code in today's OpenBSD are *foundational*: they were introduced prior to the release of the initial version we studied and have not been altered since. We also learn that 62% of reported vulnerabilities were present when the study began and can also be considered to be foundational.

1 Introduction

Many in the security research community have criticized both the insecurity of software products and developers' perceived inattention to security. However, we have lacked quantitative evidence that such attention can improve a product's security over time. Seeking such evidence, we asked whether efforts by the OpenBSD development team to secure their product have decreased the rate at which vulnerabilities are reported.

In particular, we are interested in responding to the work of Eric Rescorla [11]. He used data from ICAT[1] to argue that the rate at which vulnerabilities are reported has not decreased with time; however, limitations in the

OS versions of 50 LG handsets

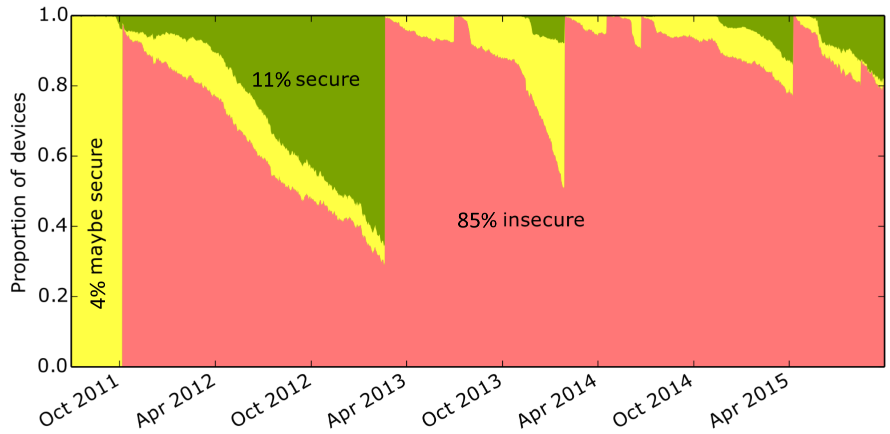


Link OS versions to database of vulnerabilities

Match OS version information to OS and Build Number to put each handset into one group:

- Insecure
- Maybe secure
- Secure

On average, 85% are vulnerable



The *Software Crisis*

- Software still lags behind hardware's potential
- Many large projects are late, over budget, dysfunctional, or abandoned (CAPSA, NPfIT, DWP, Addenbrookes, …)
- Some failures cost lives (Therac 25) or billions (Ariane 5, NPfIT)
- Some expensive scares (Y2K, Pentium)
- Some combine the above (LAS)

London Ambulance Service disaster

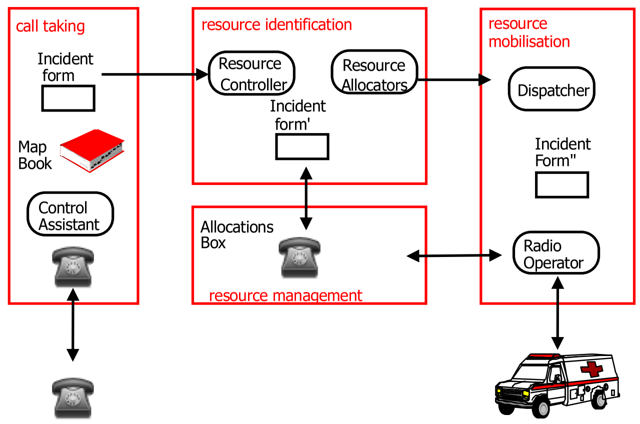
- Widely cited example of project
- Many aspects of the failure widely repeated since

- Attempt to automate ambulance dispatch in 1992
- Result left London without service for a day
- Number estimated deaths ran as high as 20
- CEO being sacked; public outrage

Project background

- Attempt to automate in 1980s failed – system failed load test
- Industrial relations poor; pressure to cut costs
- Public concern over service quality
- South West Thames Regional Health Authority decided on fully automated system: responder would "email" ambulance
- Consultancy study said this might cost £1.9m and take 19 months, *provided a packaged solution could be found*. AVLS would be extra

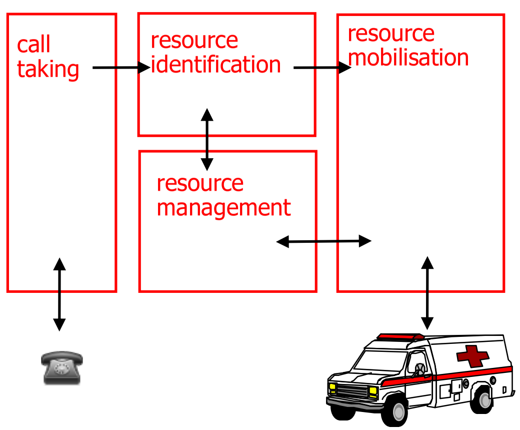
Original dispatch system worked on paper with regional control



Many problems with original system

- It took 3 minutes to dispatch an ambulance
- It required 200 staff (out of 2700 in total).
- There were errors, especially in deduplication
- Queues and bottlenecks, especially with the radio
- Call-backs tiresome

Computer-aided dispatch system



- Large
- Real-time
- Critical
- Data rich
- Embedded
- Distributed
- Mobile components

Tender process was poor

- Idea of a £1.5m system stuck; idea of AVLS added; proviso of a packaged solution forgotten; new IS director hired
- Tendered on 7th Feb 1992; completion due Jan 1992
- 35 firms looked at tender; 19 proposed; most said timescale unrealistic, only partial automation possible by early 1992
- Tender awarded to consortium of Systems Options Ltd, Apricot and Datatrak for £937,463
  - £700K cheaper than next lowest bidder!

Phase one: design work 'done' in July and contract signed in August

Minutes of a progress meeting in June recorded:

- A 6-month timescale for an 18-month project
- A lack of methodology
- No full-time LAS users providing domain knowledge
- Lead contractor (System Options) relied heavily on cozy assurances of subcontractors

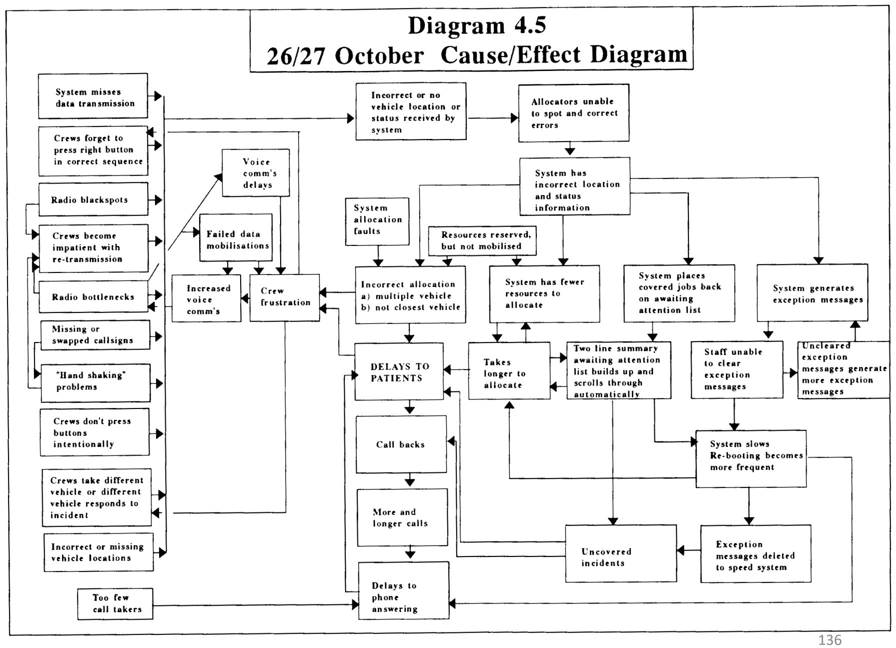
Unsurprisingly LAS told in December that only partial automation by January deadline – front end for call taking, gazetteer, docket printing

Phase two: full automation

- Server never stable in 1992; client and server lockup
- Radio messaging with blackspots and congestion; couldn't cope with established working practices
- Management decided to go live on 26th Oct 1992
- Independent review had called for volume testing, implementation strategy, change control, …all ignored
- CEO: "No evidence to suggest that the full system software, when commissioned, will not prove reliable"
- On 26 Oct, room was reconfigured to use terminals, not paper. There was no backup…

Circle of disaster on 26/7th October

- System progressively lost track of vehicles
- Exception messages scrolled off screen and were lost
- Incidents held as allocators searched for vehicles
- Callbacks from patients increased causing congestion
- data delays → voice congestion → crew frustration → pressing wrong buttons and taking wrong vehicles → many vehicles sent to an incident, or none
- System slowdown and congestion leading to collapse



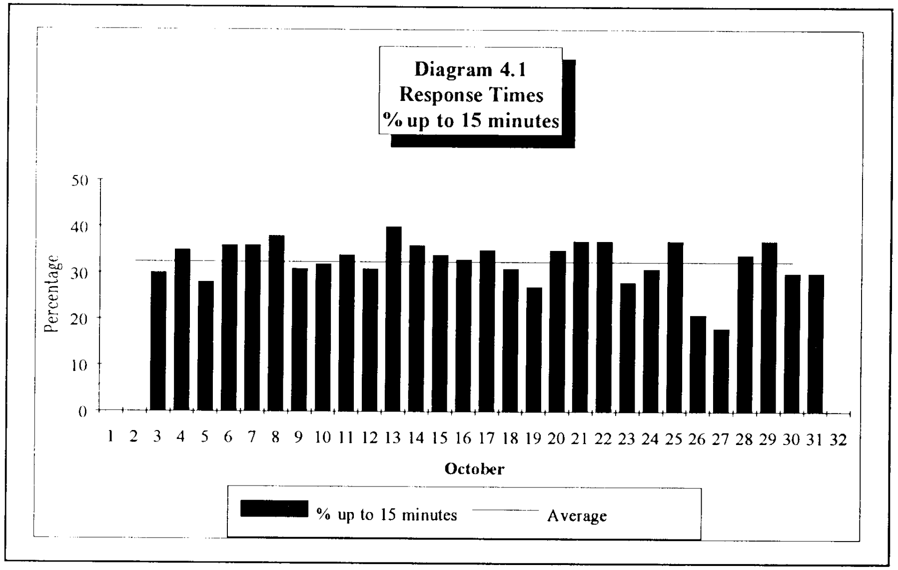


Diagram 4.1
Response Times
% up to 15 minutes
Percentage
0
10
20
30
40
50
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
October
% up to 15 minutes
Average

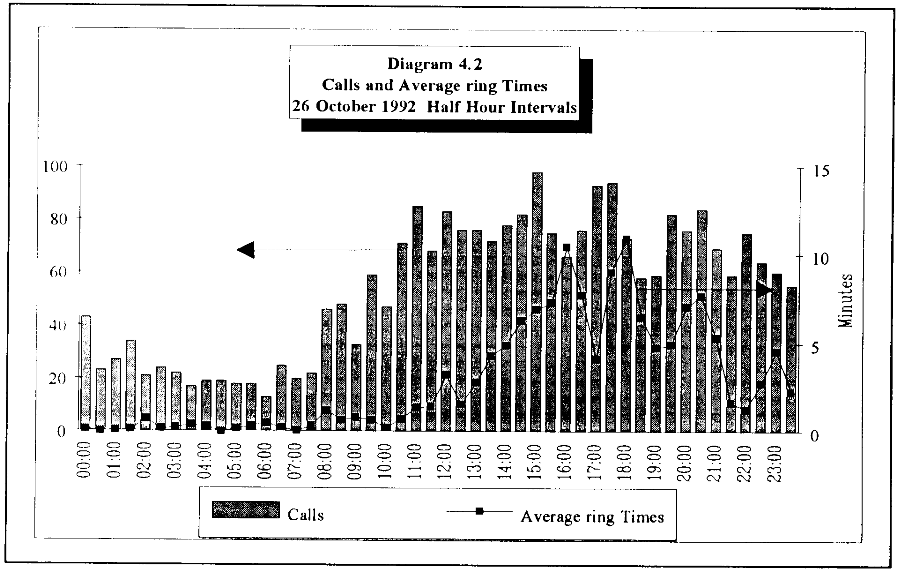


Diagram 4.2
Calls and Average ring Times
26 October 1992 Half Hour Intervals
0
20
40
60
80
100
0
5
10
15
Minutes
00:00
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00
09:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
Calls
Average ring Times

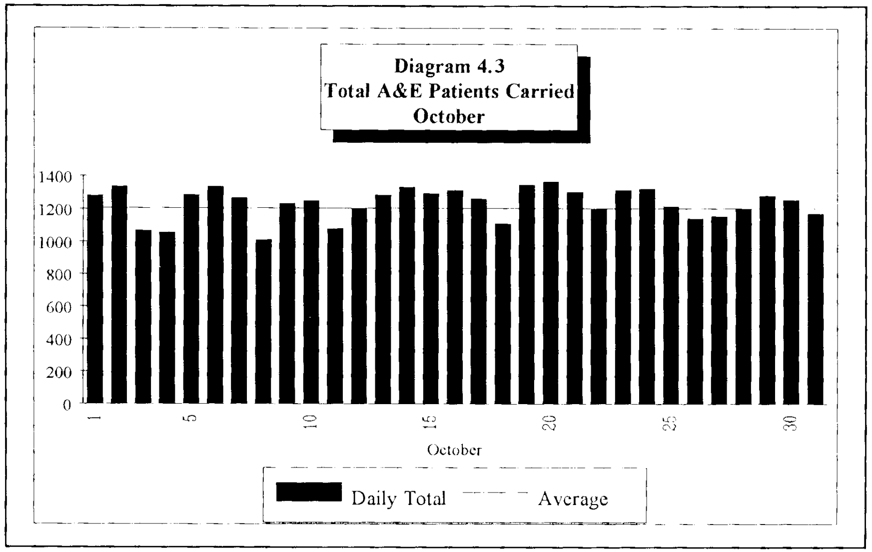


Diagram 4.3
Total A&E Patients Carried
October
0
200
400
600
800
1000
1200
1400
1
5
10
15
20
25
30
October
Daily Total
Average

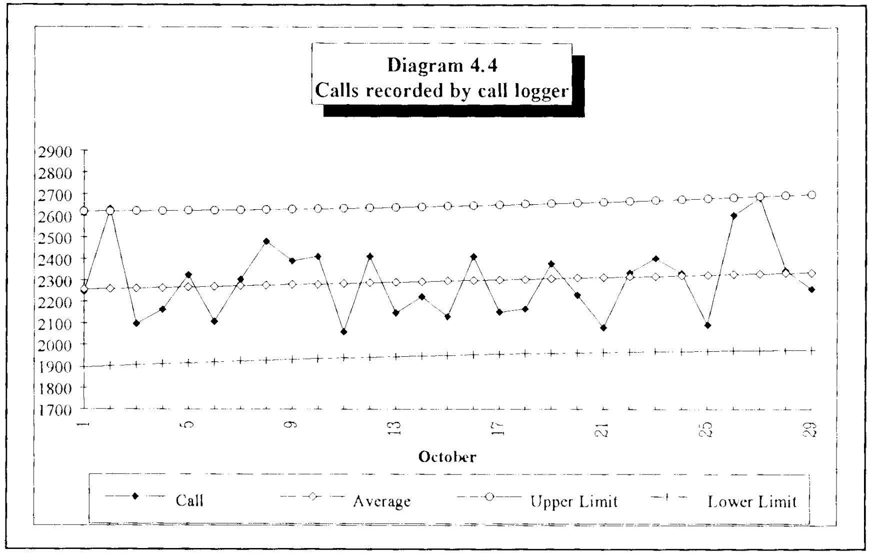


Diagram 4.4
Calls recorded by call logger
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
1
5
9
13
17
21
25
29
October
Call
Average
Upper Limit
Lower Limit

Collapse likely resulted in deaths

- One ambulance arrived to find the patient dead and taken away by undertakers
- Another answered a ‘stroke’ call after 11 hours and 5 hours after the patient had made their own way to hospital
- …
- Chief executive resigns

Specification mistakes

- LAS ignored advice on cost and timescale
- Procurers insufficiently qualified and experienced
- No systems view
- Specification was inflexible but incomplete: it was drawn up without adequate consultation with staff
- Attempt to change organisation through technical system
- Ignored established work practices and staff skills

Project management mistakes

- Confusion over who was managing it all
- Poor change control, no independent QA, suppliers misled on progress
- Inadequate software development tools
- Ditto data comms, with effects not foreseen
- Poor interface for ambulance crews
- Poor control room interface

Operational mistakes

- System went live with known serious faults
  - slow response times
  - workstation lockup
  - loss of voice comms
- Software not tested under realistic loads or as an integrated system
- Inadequate staff training
- No effective back-up system in place

NHS National Programme for IT

Idea: computerise and centralise all record keeping for every visit to every NHS establishment

- Like LAS, an attempt to centralise power and change working practices
- Earlier failed attempt in the 1990s
- The February 2002 Blair meeting
- Five LSPs plus national contracts: £12bn
- Most systems years late or never worked
- Coalition government: NPfIT 'abolished'

Universal Credit: fix poverty trap

Idea: Hundreds of welfare benefits which means there is often little incentive to get a job.

- Initial plan was to go live in October 2013
- A significant problem: big systems take seven years not three; doesn't align with political cycle
- Complexity was huge, e.g. depended on real-time feed of tax data from HMRC, which in turn depended on firms
- Descended into chaos; see NAO report

Smart meters: more centralisation

Idea: expose consumers to market prices, get peak demand shaving, make use salient

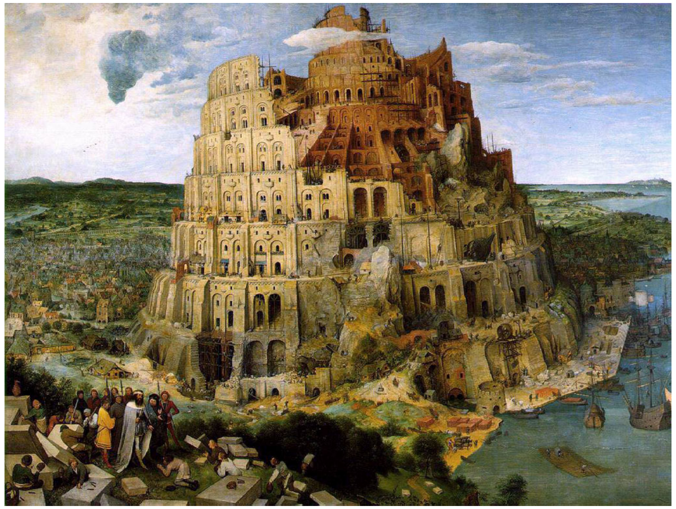
- 2009: EU Electricity Directive for 80% by 2020
- 2009: Labour £10bn centralised project to save the planet and help fix supply crunch in 2017
- 2010: Experts said we just can't change 47m meters in 6 years. So excluded from spec
- Coalition government: wanted deployment by 2015 election! Planned to build central system Mar–Sep 2013 (then: Sep 2014 …)
- Spec still fluid, tech getting obsolete, despair …

Software engineering is about managing complexity at many levels

- Bugs arise at micro level in challenging components
- As programs get bigger, interactions between components grow at $O(n^2)$ or even $O(2^n)$
- The 'system' isn't just the code: complex socio-technical interactions mean we can't predict reactions to new functionality

Most failures of really large systems are due to wrong, changing, or contested requirements

Project failure, circa 1500 BCE



On contriving machinery

"It can never be too strongly impressed upon the minds of those who are devising new machines, that to make the most perfect drawings of every part tends essentially both to the success of the trial, and to economy in arriving at the result"

Charles Babbage

Bank of England, 1870

Dun, Barlow & Co, 1876

Sears, Roebuck and Company, 1906

- Continental-scale mail order meant specialization
- Big departments for single bookkeeping functions
- Beginnings of automation

First National Bank of Chicago, 1940

The software crisis, 1960s

- Large, powerful mainframes made complex systems possible
- People started asking why project overruns and failures were so much more common than in mechanical engineering, shipbuilding, etc.
- The term *software engineering* coined in 1968
- The hope was that we could things under control by using disciplines such as project planning, documentation and testing

Those things which make writing software fun also make it complex

- Joy of solving puzzles and building things from interlocking parts
- Stimulation of a non-repeating task with continuous learning
- Pleasure of working with a tractable medium, 'pure thought stuff'
- Complete flexibility – you can base the output on the inputs in any way you can imagine
- Satisfaction of making stuff that's useful to others

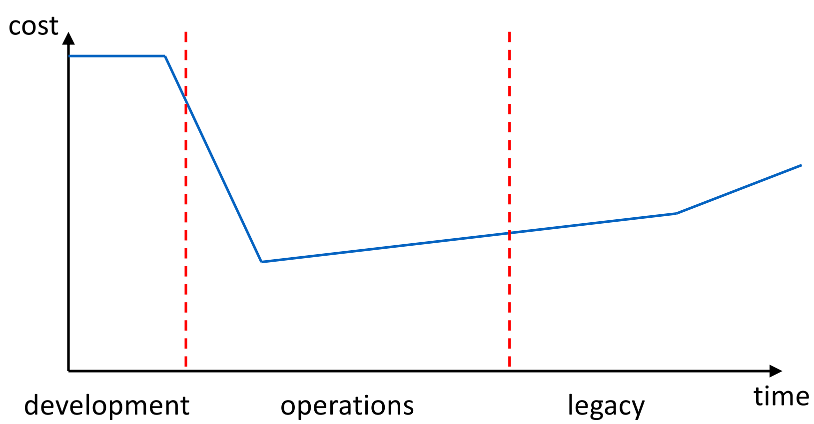
How is software different?

- Large computer systems become qualitatively more complex, unlike big ships or long bridges
- The tractability of software leads customers to demand flexibility and frequent changes
- This makes systems more complex to use over time as features accumulate, and interactions have odd effects
- The structure can be hard to visualise or model
- The hard slog of debugging and testing piles up at the end, when the excitement's past, the budget's spent and the deadline's looming

Software economics can be nasty

- Consumers buy on sticker price
- Businesses buy based on total cost of ownership
- Vendors use lock-in tactics
- Complex outsourcing

Cost of software: development 10%, maintenance 90%



Measuring cost of code is hard

First IBM measures (1960s)

- 1.5 KLOC per developer-year (operating system)
- 5 KLOC per developer-year (compiler)
- 10 KLOC per developer-year (app)

AT&T measures

- 0.6 KLOC per developer-year (compiler)
- 2.2 KLOC per developer-year (switch)

KLOC is a poor measure

1.
```
//Print out hello
for (int i = 0; i < 4; i++) {
  System.out.println("Hello, world");
}
```

2.
```
for (int i = 0; i < 4; i++) { System.out.println("Hello, world");}
```

3.
```
System.out.println("Hello, world");
System.out.println("Hello, world");
System.out.println("Hello, world");
System.out.println("Hello, world");
```

Alternatives:

- Halstead (entropy of operators/operands)
- McCabe (graph entropy of control structures)
- Function point analysis

Early lessons: use a high-level language, productivity varies

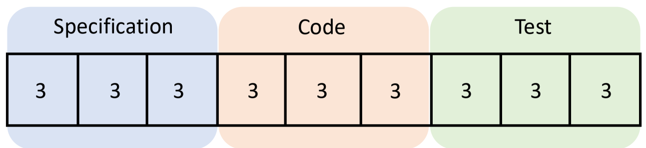
- Huge variations in productivity between individuals
- The main systematic gains come from using an appropriate high-level language since they reduce accidental complexity; programmer focuses on intrinsic complexity
- Get the specification right: it more than pays for itself by reducing the time spent on coding and testing

Barry Boehm surveyed relative costs of software development (1975)

	Spec	Code	Test
C3I	46%	20%	34%
Space	34%	20%	46%
Scientific	44%	26%	30%
Business	44%	28%	28%

- All stages of software development require good tools

Mythical Man-Month: *"adding manpower to a late project makes it later"*



Example project with 3 developers and 9 months. Initial estimate is 6 person-months each for spec, code and test.

- But spec ends up taking 9 PMs. What do you do?

Mythical Man-Month: *"adding manpower to a late project makes it later"*

Specification				Code		Test		
3	3	3	3	3	6	3	3	3
				Train				

We try to catch up:

- Train 3 more developers in the first month, then use all 6 developers in the next month
- But: work of 3 developers in 2 months can't be done by 6 developers in 1 – interaction costs maybe $O(n^2)$

Time to first shipment is cube root of developer-months (Boehm, 1981)

$$T = 2.5\sqrt[3]{\mathrm{d}}$$

where T is time to first shipment and d is developer months

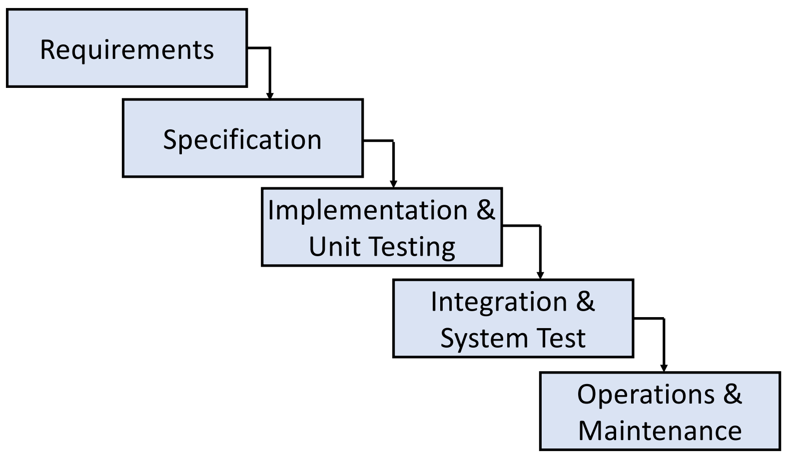
- With more time, costs rise slowly
- With less time, costs rise sharply
- Hardly any projects succeed at $\frac{3}{4}T$
- Some projects still fail

The Software Tar Pit

Take a structured, modular approach

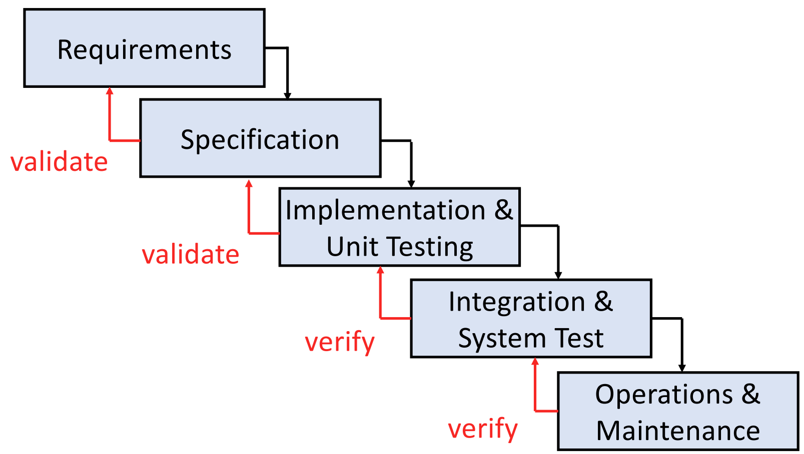
- Only practical way forward is *modularisation*
- Divide a complex system into small components
- Define clear APIs between them
- Lots of methodologies based on this idea:
  - SSDM
  - Jackson
  - Yourdon,
  - UML,
  - …

The Waterfall Model (1970)



169

The Waterfall Model (1970)



170

Waterfall Model has advantages

- Compels early clarification of system goals
- Supports charging for changes to the requirements
- Works well with many management and tech tools
- Where it's viable it's usually the best approach
- The really critical factor is whether you can define the requirements in detail in advance. Sometimes you can (Y2K bugfix); sometimes you can't (HCI)

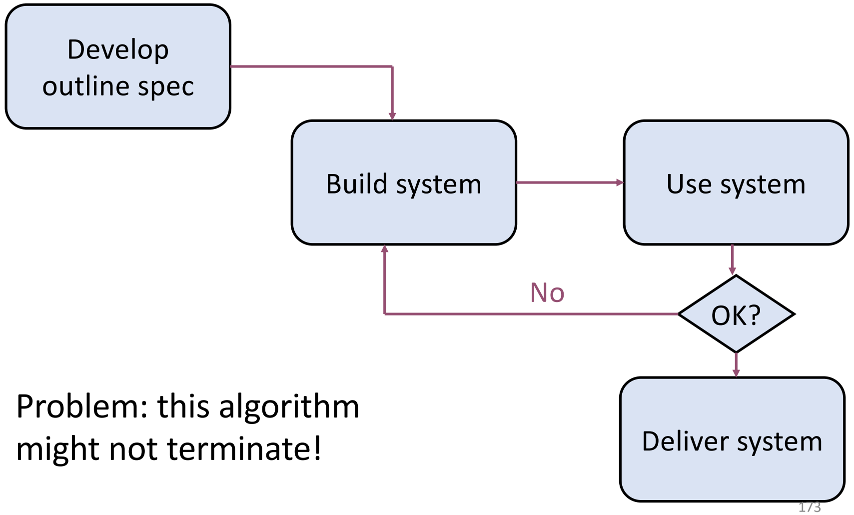
171

Waterfall fails where iteration is required, such as:

- Requirements not yet understood by developers
- Not yet understood by the customer
- The technology is changing
- The environment (legal, competitive) is changing
- …

172

Iterative development

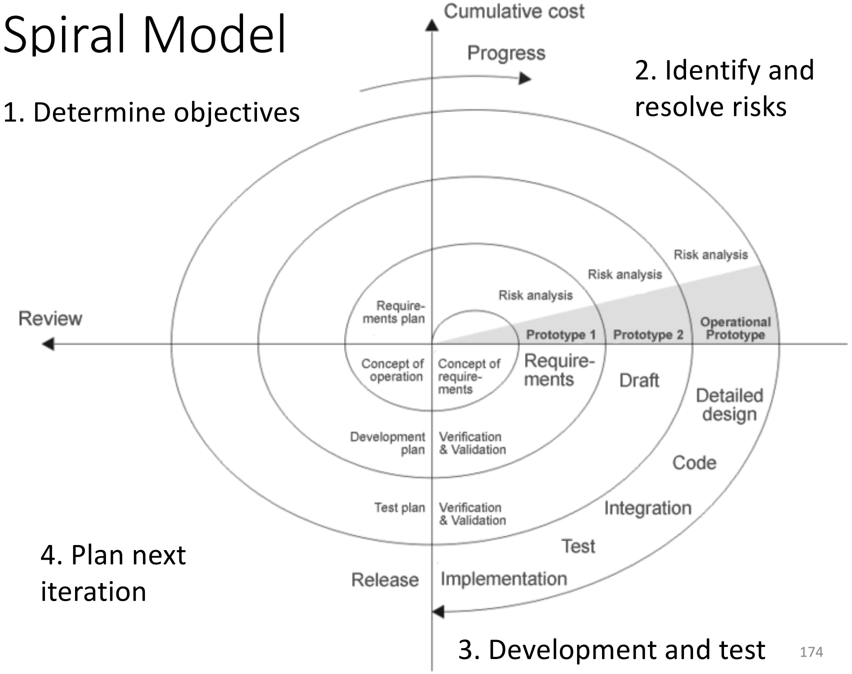


Problem: this algorithm might not terminate!

Spiral Model

1. Determine objectives

2. Identify and resolve risks



4. Plan next iteration

3. Development and test

Spiral model invariants

- Decide in advance on a fixed number of iterations
- Each iteration is done top-down
- Driven by risk management (i.e. prototype bits you don't yet understand)

Evolutionary model

- By the 1990s some codebases had become so big and complex they *had* to evolve
- Solution: use *automatic regression testing*
- Firms now have huge suites of test cases which run against daily builds of software
- Development cycle is then to add changes, check them into a repository, and test them

The Integrated Development Environment (IDE) includes…

- Code and documentation under version control (Git)
- Code review (Gerrit)
- Automated build system (Maven)
- Continuous integration (Jenkins)
- Dev / Test / Prod deployment (Webserver)

Content-heavy apps benefit from four host types

		Content	
		Latest	*Stable*
Software	*Latest*	**Test**	**Dev**
	Stable	**Staging**	**Prod**

Assurance of critical software: must study how things fail

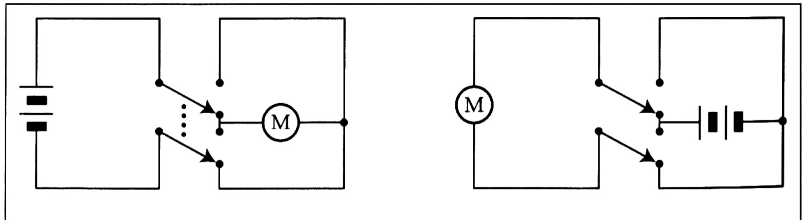
- Critical software avoids certain class of failures with high assurance
- *Safety-critical systems*: failure could cause, death, injury or property damage
- *Security-critical systems*: failure could allow leakage of confidential data, fraud, …
- *Real-time systems*: software must accomplish certain tasks on time

Critical computer systems have much in common with mechanical systems (bridges, brakes, locks)

Tacoma Narrows, 7th Nov 1940

https://www.youtube.com/watch?v=j-zczJXSxnw

Hazard elimination



(a) (b)

- Which motor reversing circuit is the safe above?
- Some architecture and tool choices can eliminate whole classes of software hazards, e.g. using a garbage collector to eliminate and memory leaks.
- But usually hazards involve more than just software

Ariane 5, 4th June 1996

- Ariane 5 accelerated faster than Ariane 4, causing an error in float-to-integer conversion
- The backup inertial navigation set core dumped, which was interpreted by as flight data
- Full nozzle deflection → 20° angle of attack → booster separation

Multi-factor failure

- Many safety-critical systems are also real-time systems used in monitoring or control
- Exception handling is often tricky
- Criticality of timing makes many simple verification techniques inadequate
- Testing is often really hard

Emergent properties

- In general, safety is a system property and has to be dealt with holistically
- The same goes for security, and real-time performance too
- A very common error is not getting the scope right
- For example, designers don't consider human factors such as usability and training

Therac-25: radiotherapy machine

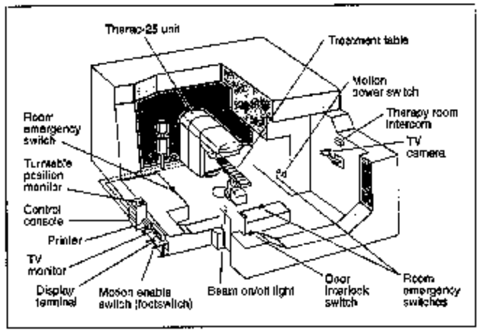
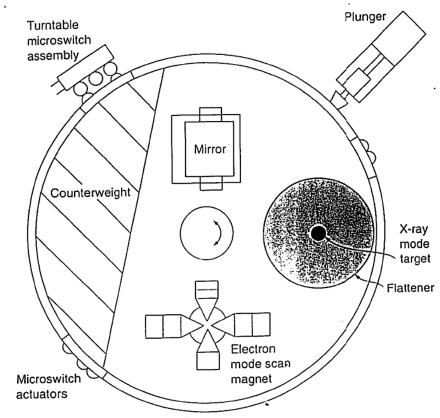


Figure 1. Typical Therac-25 facility.

- Three people died in six accidents
- Example of fatal programming error
- Usability issues
- Poor safety engineering

Therac had two operating modes



- 25 MeV electron focused beam to generate X-rays
- 5-25 MeV spread electron beam for skin treatment

Safety requirement: don’t fire focused beam at humans

Therac-25 used software to enforce safe operation

- Previous models (Therac-6 and 20) used mechanical interlocks to prevent high-intensity beam use unless X-ray target in place
- The Therac-25 replaced these with software
- Fault tree analysis arbitrarily assigned probability of 10^{-11} to ‘computer selects wrong energy’
- Code was poorly written, unstructured and not properly documented

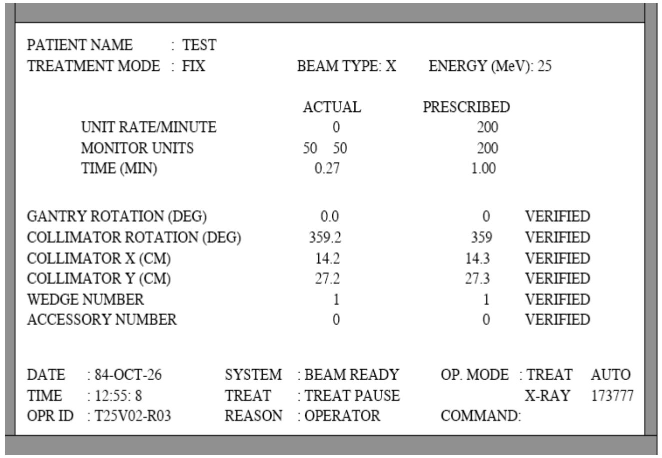
Therac-25 caused injuries

- Marietta, GA, June 1985: woman’s shoulder burnt. Settled out of court. FDA not told
- Ontario, July 1985: woman’s hip burnt. AECL found microswitch error but could not reproduce fault; changed software anyway
- Yakima, WA, Dec 1985: woman’s hip burned. ‘Could not be a malfunction’

Therac-25 killed three people

- East Texas Cancer Centre, March 1986: man burned in neck and died five months later of complications
- Same place, three weeks later: another man burned on the face and died three weeks later
- Hospital physicist managed to reproduce flaw: if parameters changed too quickly from X-ray to electron beam, the safety interlock failed
- Yakima, WA, January 1987: man burned on the chest and died due to different bug now thought to have caused Ontario accident

Therac-25: East Texas deaths due to editing beam type too quickly



Therac-25: root cause analysis

- Manufacturer ignored safety aspects of software
- Confusion between reliability and safety
- Lack of defensive design
- Inadequate reporting, follow-up or regulation
- Unrealistic risk assessments
- Inadequate software engineering practices
- Manufacturer left the medical equipment business

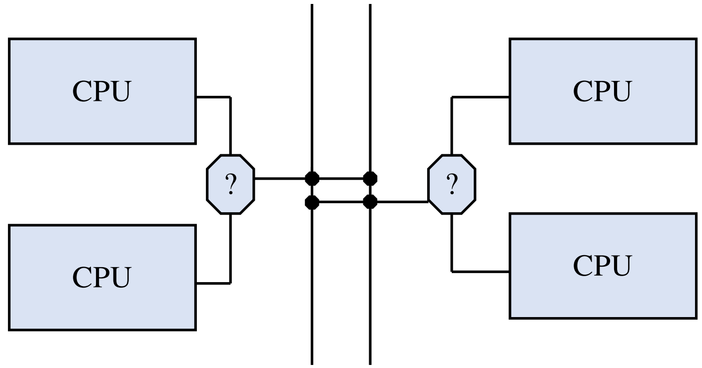
Software safety myths: cheaper, easy to change, reliable

- Computers are cheaper than analogue devices
  - Shuttle software cost $\$10^8$ pa to maintain
- Software is easy to change
  - Exactly! But it's hard to change safely…
- Computers are more reliable
  - Shuttle software had 16 potentially fatal bugs found since 1980 – and half of them had flown
- Increasing reliability increases safety
  - They're correlated but not completely

Software safety myths: reuse, formal methods, testing and automation

- Reuse increases safety
  - Counter examples: Ariane 5, Patriot and Therac-25
- Formal verification can remove all errors
  - Not even for 100-line programs
- Testing can make software arbitrarily reliable
  - For MTBF of 10^9 hours you must test $>10^9$ hours
- Automation can reduce risk
  - Also an opportunity for new types of failure

Stratus computer: redundant hardware for non-stop processing



Redundant hardware does not solve software engineering issues

- Hardware can still fail; backup inertial navigation failed first on the Ariane rocket
- Redundant hardware creates additional software engineering issues
- Redundant software (*multi-version programming*) sounds promising…
- But: errors are correlated, dominated by failure to understand requirements (Leveson)
- Implementations often give different answers

Redundancy in the Boeing 737

Panama crash with 47 fatalities 6th June 1992

Lower photo: CC-BY-SA Markus Vitzethum

- Need to know which way up
- New EFIS (each pilot), WW2 artificial horizon (top right)
- EFIS failed due to loose wire
- Both EFIS fed off same inertial navigation set
- Pilots watched EFIS, not AH
- And again: Korean Air cargo 747, Stansted 22nd Dec 1999

Kegworth crash, 47 fatalities 8th January 1989

- Fan blade broke
- Crew shutdown wrong engine
- Emergency landing at East Midlands
- Opened throttle on final approach: no power
- Initially blamed wiring; later cockpit design

Aviation is actually an easy case

- It's a mature evolved system
- Stable components: aircraft design, avionics design, pilot training, air traffic control …
- Interfaces are stable
- Crew capabilities are well known
- The whole system has good incentives for learning – much better than with medical devices
- Excellent regulation and reporting

Still complex social-technical system that exhibits failure

Understand and prioritise hazards

Example from the motor industry:

1. *Uncontrollable*: outcomes can be extremely severe and not influenced by human actions
2. *Difficult to control*: very severe outcomes, influenced only under favourable circumstances
3. *Debilitating*: usually controllable, outcome at worst severe
4. *Distracting*; normal response limits and outcome to minor
5. *Nuisance*: affects customer satisfaction but not normally safety

Managing safety and security across the software lifecycle

- Develop a safety case or security policy
- Design a management plan
- Identify critical components
- Develop test plans, procedures, training
- Plan for and obtain certification
- Integrate all the above into your development methodology (waterfall, spiral, evolutionary, …)

Most mistakes occur outside the technical phases

Challenging parts are often:

- Requirements engineering
- Certification
- Operations
- Maintenance

This is due to the interdisciplinary nature of these parts, involving technical staff, domain experts, users, cognitive factors, politics, marketing, …

The Internet of Things: safety now includes security

- Cars, medical devices, electricity grid all have 10+ year lifetimes as well as formal certification
- All contain software; will be Internet connected
- Apparent conflict between safety and security
  - E.g. first DDoS attack (Panix ISP) was from driven from hacked Unix machines with medical certification
- Good security requires us to move to monthly patching, yet this conflicts with the safety case

Software engineering tools help us manage complexity

Homo sapiens uses tools when some parameter of a task exceeds our native capacity. So:

- Heavy object: raise with lever
- Tough object: cut with axe
- …
- Software complexity: ?

Good tools eliminate *incidental* and manage *intrinsic* complexity

Incidental complexity: dominated programming in the early days, including writing programs in assembly. Better tools eliminate such problems.

Intrinsic complexity: the main problem today, since we now write complex systems with big teams. There are no solutions, but tools help, including structured development, project management tools, …

High-level languages remove incidental complexity

- 2 KLOC per year goes much farther than assembler
- Code easier to understand and maintain
- Appropriate abstraction: data structures, functions, objects rather than bits, registers, branches
- Structure finds many errors at compile time
- Code may be portable; or at least, the machine-specific details can be contained

Huge performance gains possible, now realised

High-level languages support structure and componentisation

Much historical work on both languages and language features, including:

- "*Goto statement considered harmful*" (Dijkstra, 1968)
- Structured programming with Pascal (Wirth, 1971)
- Object-oriented programming (see OOP course)
- …

Don't forget: this is to *manage intrinsic complexity*

Formal methods finds bugs, but it is fallible

History:

- Turing talked about proving programs correct
- Floyd-Hoare logic; Floyd (1967), Hoare (1969)
- HOL; Gordon (1988)
- Z notation
- BAN logic
- …

Static analysis tools are a useful result of formal methods

DOI:10.1145/1646353.1646374

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later
Using Static Analysis to Find Bugs in the Real World

like all static bug finders, leveraged the fact that programming rules often map clearly to source code; thus static inspection can find many of their violations. For example, to check the rule "acquired locks must be released," a checker would look for relevant operations (such as `lock()` and `unlock()`) and inspect the code path after flagging rule disobedience (such as `lock()` with no `unlock()` and double locking).

For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).

A glance through the literature reveals many ways to go about static bug finding.[1,2,4,7,8,11] For us, the central religion was results: If it worked, it was good, and if not, not. The ideal: check millions of lines of code with little manual setup and find the maximum number of serious true errors with the minimum number of false reports. As much as possible, we avoided using annotations or specifications to reduce

Chief programmers (IBM, 1970s)

Aim: avoid loss of great programmers to management and capitalise on wide productivity variance

- Teams consisting of chief programmer, apprentice, toolsmith, librarian, admin assistant, etc.
- Can be effective during implementation
- But each team can only do so much

Egoless programming: minimize personal factors (Weinberg, 1971)

- Code should be owned by the team
- Direct opposite to the Chief Programmer approach
- Groupthink can entrench bad practice deeply

Literate programming (Knuth, 1984)

- Treat programs as literature, readable by humans
- Primarily a work of literature, with code added
- Literate programs are compiled in two ways:
  - *Weaving*: a comprehensive human-readable document about the program and its maintenance.
  - *Tangling*: the machine executable code
- Literate programming is *not* documentation embedded in code, such as Javadoc.

Capability Maturity Model (Humphrey, 1989)

1. **Initial** (chaotic, ad hoc, individual heroics) – the starting point for use of a new process
2. **Repeatable** – the process is able to be used repeatedly, with roughly repeatable outcomes
3. **Defined** – the process is defined/confirmed as a standard business process
4. **Managed** – the process is managed according to the metrics described in the Defined stage
5. **Optimized** – process management includes deliberate process optimization/improvement

Extreme programming (Beck, 1999)

- Iterative development with short cycles
- Automated build and test suites
- Frequent points to integrate new requirements
- Solve the worst problem, repeat
- Avoid programming a feature until needed
- Programming in pairs, one keyboard and screen
- Extensive code review

Agile software development (2001)

Four values:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Also twelve principles (see related work), including frequent release, daily meetings, working software as measure of progress, regular reflection, etc.

The specification still matters

Curtis (1988) found causes of failure were:

1. Thin spread of application domain knowledge
2. Fluctuating and conflicting requirements
3. Breakdown of communication, coordination

Causes were very often linked, and the typical progression to disaster was $1 \rightarrow 2 \rightarrow 3$

Specification is hard: thin spread of application domain knowledge

- How many people understand everything about running a phone service, bank or hospital?
- Many aspects are jealously guarded secrets
- Some fields try hard to be open, e.g. aviation
- With luck you might find a real ‘guru’
- You should expect mistakes in specification

Specification is hard: fluctuating and conflicting requirements

- Competing products, new standards, fashion
- Changing environment (takeover, election, …)
- New customers (e.g. overseas) with new needs
- …

The specification can kill you

- Spec-driven development of large systems leads to communication problems since N people means N(N-1)/2 channels and 2^N subgroups
- Big firms have hierarchy; if info flows via ‘least common manager’, bandwidth will be inadequate
- Proliferation of committees, staff departments causing politicking, blame shifting
- Management attempts to gain control result in restricting many interfaces, e.g. to the customer

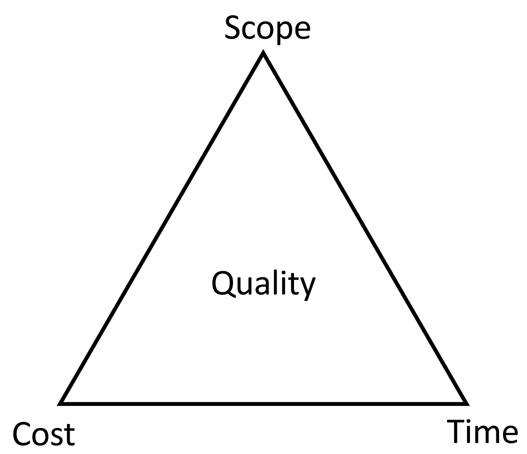
Project management: plan, motivate, control

A manager’s job is to:

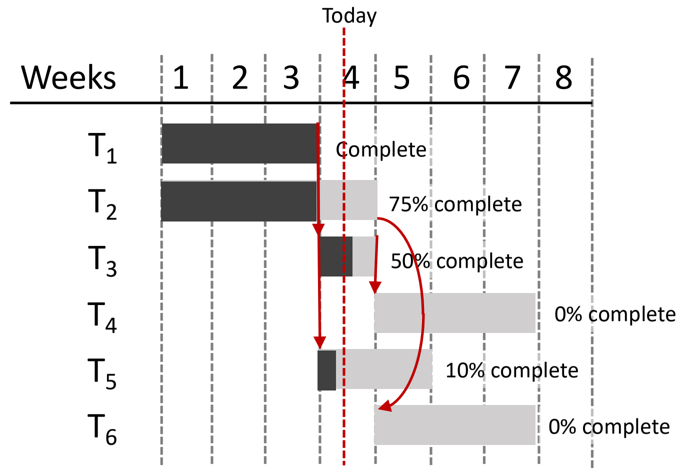
- Plan
- Motivate
- Control

- The skills involved are interpersonal, not technical; but managers must retain respect of technical staff
- Growing software managers a perpetual problem! (Managing programmers is like herding cats.)
- Nonetheless there are some tools that can help

Project management triangle

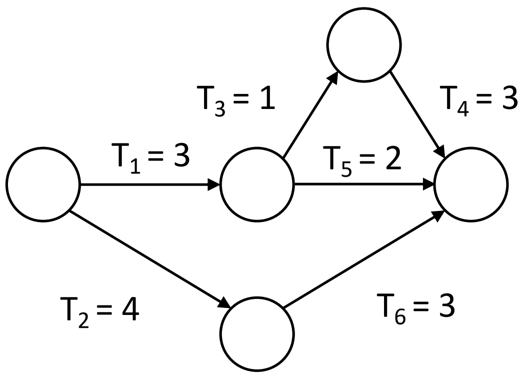


Gantt charts: tasks and milestones



Can be hard to visualise dependencies in large charts

PERT charts: show critical paths



Which paths are critical?

Motivating people in groups

- People can slack in groups (*free rider*, *social loafing*)
- Competition no good: people who don't think they will win stop trying
- Dan Rothwell's three C's of motivation:
  - Collaboration – everyone has a specific task
  - Content – everyone's task clearly matters
  - Choice – everyone has a say in what they do
- Many other factors

Testing: half the effort (and cost)

Happens at many levels:

- Design validation, UX prototyping
- Module test after coding
- System test after daily build
- Beta test / field trial
- Subsequent litigation

Cost per bug rises dramatically down this list!

Design for testability, use CI and automate regression testing

Regression Tests: check that new versions of the software give same answers as old versions

- Customers more upset by failure of a familiar feature than at a new feature which does not work
- Without regression testing, 20% of bug fixes reintroduce failures in already tested behaviour
- Test the inputs that your users actually generate
- In hard-core Agile philosophy, tests *are* the spec

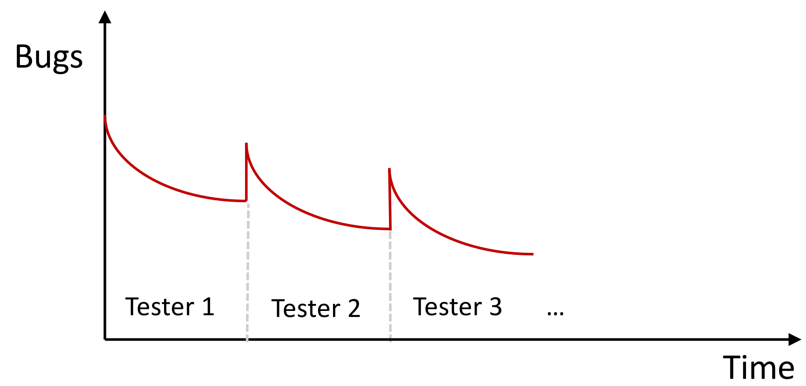
A MTBF of *x* requires testing for *x*

- Reliability growth models help us assess MTBF, number of bugs remaining, economics of further testing, etc.
- Failure rate due to one bug can be modelled as

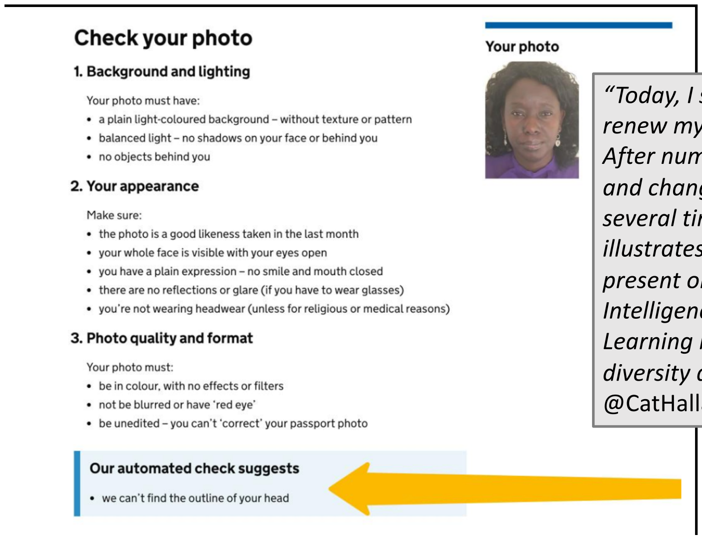
$$f_r = e^{\frac{k}{T}}$$

  and with many bugs these sum $f_r = \frac{k}{T}$
- So for 10^9 hours MTBF, must test >10^9 hours

Changing testers finds more bugs



Think about diversity & inclusion



"Today, I simply wanted to renew my passport online. After numerous attempts and changing my clothes several times, this example illustrates why I regularly present on Artificial Intelligence/Machine Learning bias, equality, diversity and inclusion"
@CatHallam1

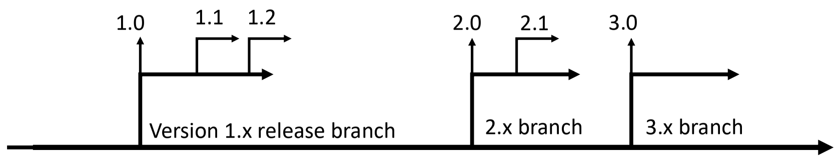
Tests should exercise the conditions when system is in use

- Many failures result from unforeseen input or environment conditions (e.g. Patriot)
- Random testing – *fuzzing* – now good practice
- Incentives matter: commercial developers look for friendly certifiers, while military, NASA, DoE arrange hostile review
- So: to whom do you have to prove what?

Keeping all documents in sync is hard

- How will you deal with management documents (budgets, PERT charts, staff schedules)?
- Engineering documents (requirements, hazard analyses, specifications, test plans, code)?
- Possible partial solutions:
  - High tech: integrated development environment
  - Bureaucratic: plans and controls department
  - Social consensus: style, comments, formatting

Release management: from development code to production



- Main focus is on stability
- Add copy protection, rights management
- Critical decision: patch old version or force upgrade?

Change control and operations: important and can be overlooked

- Change control and config are critical; often poor
- Objective: manage testing and deployment
- Someone must assess risk and be responsible for:
  - Live running
  - Manage backup
  - Recovery
  - Rollback
  - …
- DevOps integrates development and operations

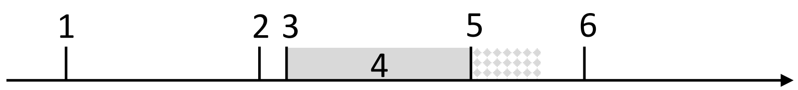
Vulnerability disclosure: the modern consensus is *coordinated disclosure*

Possible options for discoverer:

1. Disclose without notice: a *zero day*
2. Publicly disclose after a fixed delay: *coordinated* or *responsible disclosure*
3. Publicly disclose after vendor fix
4. No disclosure, but then vendor can't fix

Vendors use bug bounty programmes to discourage 1.

Vulnerability lifecycle



1. Engineer introduces a bug
2. Someone discovers it
3. Coordinated disclosure; disclose at once; or exploit
4. Primary exploit window
5. Patch released
6. Public notification of bug

- What about orphaned devices or Mirai?

Shared infrastructure provides benefits & implies responsibilities

- We share a lot of code through open source operating systems, libraries and tools
- Huge benefits but also interaction issues
- Can you cope with an emergency bug fix?
- How do you feed your fixes back to others?
- Do you encourage coordinated disclosure?
- Are you aware of different license terms?

Beware of agency issues

- Employees often optimize their own utility, not project utility (recall London Ambulance Service)
- Bureaucracies are machines for avoiding blame
- Risk reduction becomes compliance
- Tort law reinforces herding: negligence judged ‘by the standards of the industry’
- So firms do the checklists, use fashionable tools, hire the big consultants…

Focus on outcomes over process

- Metrics easier for regular losses (risk)
- But rare catastrophes are hard (uncertainty)
- How reassuring are fatality statistics? E.g. Train Protection Systems, Tesla
- Accidents are random, but security exploits are not
- Product liability for death or injury is strict

Focus on process over outcomes

- Necessary to adapt as environment changes
- Security development lifecycle is established
- Safety rating maintenance
- Blame avoidance is what bureaucracies do
- Public sector is really keen on compliance
- But leaves a gap of residual risk and uncertainty

Getting incentives right is both important and hard to do

- The world offers hostile review, which we tackle in stages
- Some use hostile reviewers deliberately
- Standard contract of sale for software in Bangalore: seller must fix bugs for 90 days
- Businesses avoid risk (regulatory games)

UK's Digital Service Standard: an example pulling it all together

- Understand user needs
- Do ongoing research
- Have a multidisciplinary team
- Use agile methods
- Iterate & improve frequently
- Evaluate tools and systems
- Understand security & privacy issues
- Make all new source code open
- Use open standards and common platforms
- Test the end-to-end service
- Make a plan for being offline
- Make sure users succeed first time
- Make the user experience consistent with GOV.UK
- Encourage everyone to use the digital service
- Collect performance data
- Identify performance indicators
- Report performance data on the Performance Platform
- Test with the minister

The future is challenging: how to we provide safety and security?

- Car manufactures must do pre-market testing
- Cars now contain lots of safety critical software
- Security requires us to patch bugs when they are found, yet this might invalidate safety case
- How will today's car get patches in 2039? 2049?
- What new tools and ideas do we need?

Software engineering is about managing complexity

- Security and safety engineering are going in the same direction
- We can cut incidental complexity using tools, but the intrinsic complexity remains
- Top-down approaches can sometimes help, but really large systems evolve
- Safety and security are often emergent properties
- Remember: all software has latent vulnerabilities

Software and security engineering stretches well beyond the technical

- Complex systems are social-technical
- Institutions and people matter
- Confluence of safety and security may make maintenance the limiting factor

Software as a Service Engineering

Richard Sharp

Director of Studies for Computer Science, Robinson College

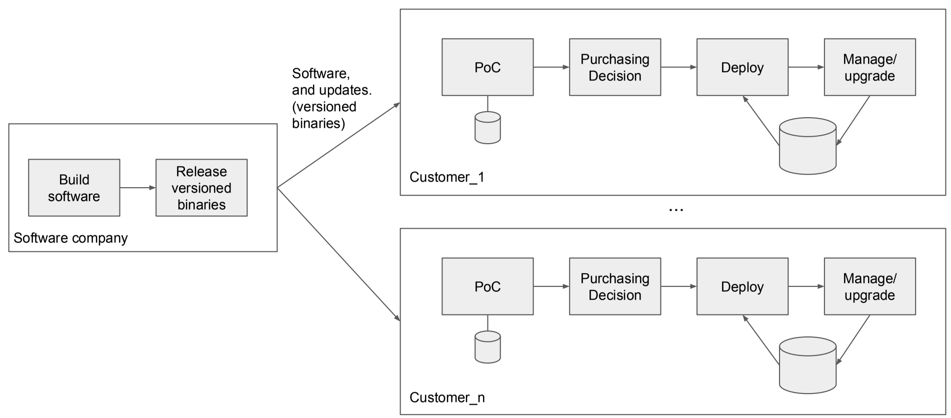
What is Saas?

SaaS (Software as a Service) refers to software that is

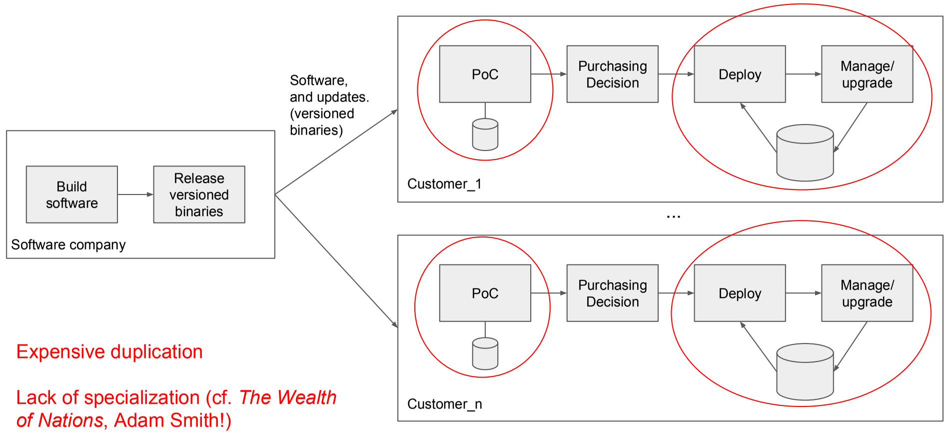
hosted centrally and *licensed to customers on a subscription basis*.

Users access SaaS software via *thin clients*, (often web browsers).

Traditional software distribution



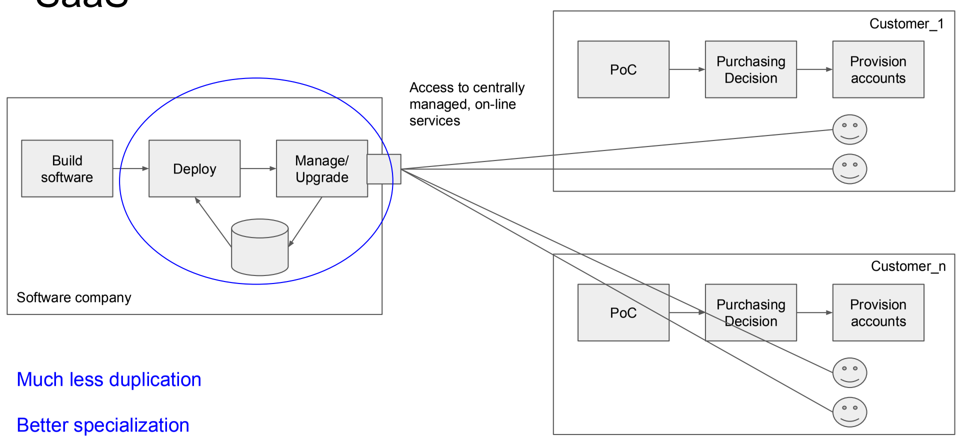
Traditional software distribution



Expensive duplication

Lack of specialization (cf. *The Wealth of Nations*, Adam Smith!)

SaaS

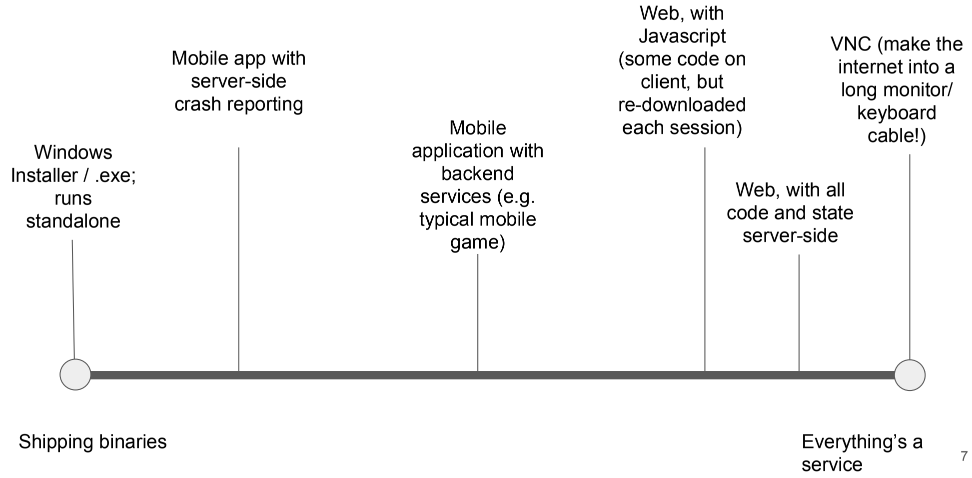


Much less duplication

Better specialization

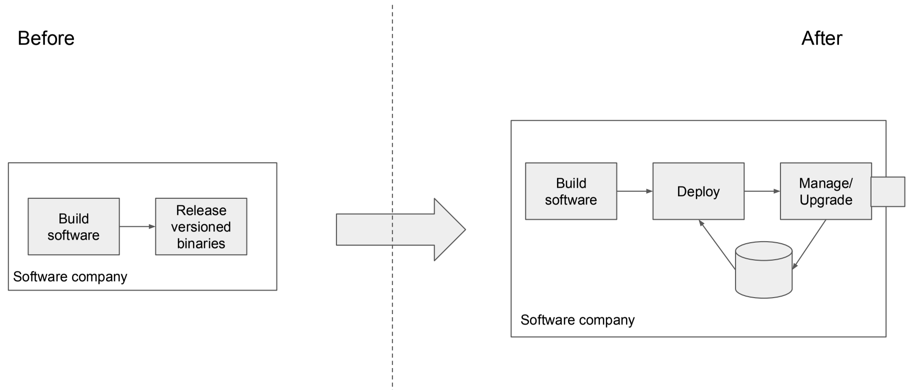
Plus central management of state so much simpler

In reality it's a spectrum



Impact of SaaS on the Software Engineering Process

Impact on the 'software company'



Impact on the 'software company'

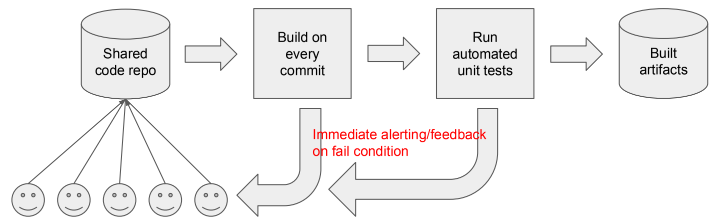
- Now have to worry about building software *and running it*
- Have to continue evolving/upgrading the software with *zero downtime*

But the good news:

- 'Software release' no longer an all-or-nothing discrete event
  - Provides new ways to manage quality and reduce risk
- Continuous visibility into user behavior
  - Provides user/commercial insights back into iterative software development process
- State and runtime environment fully controlled by service provider
  - Improves quality and makes upgrades a lot easier

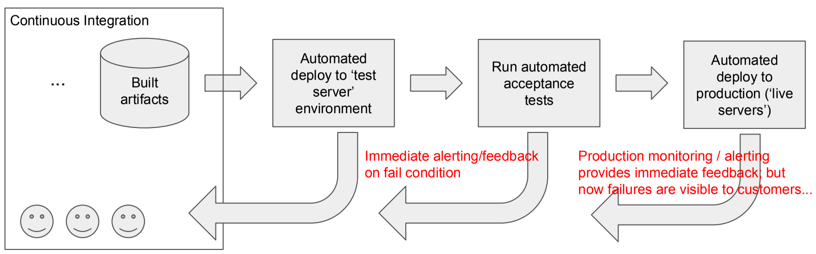
Managing Continuous Deployment Without Downtime

Continuous Integration (CI): short integration cycles lead to greater throughput

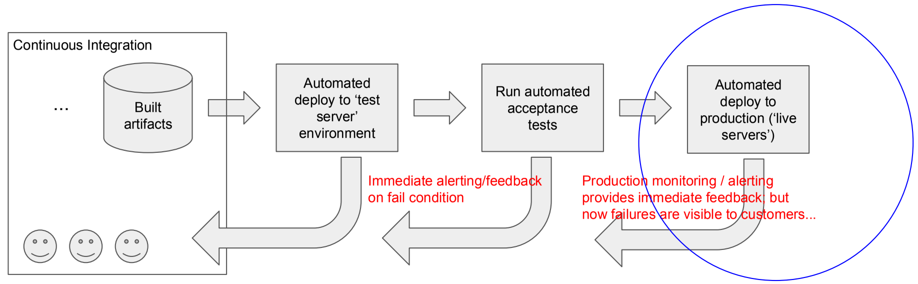


Developers commit to shared dev 'mainline' branch frequently (e.g. at least once a day)

Continuous Deployment (CD): bring ‘deploy’ into the ‘short cycle’

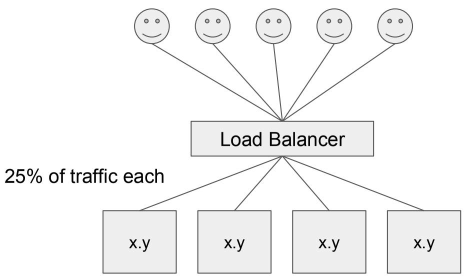


Continuous Deployment (CD): bring ‘deploy’ into the ‘short cycle’



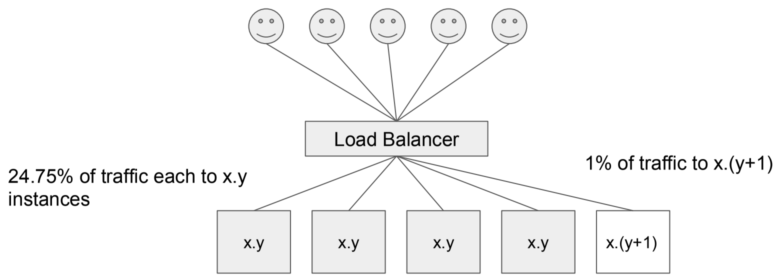
How to do this while reducing risk?
How to do this while ‘always on’?

Rolling deploy

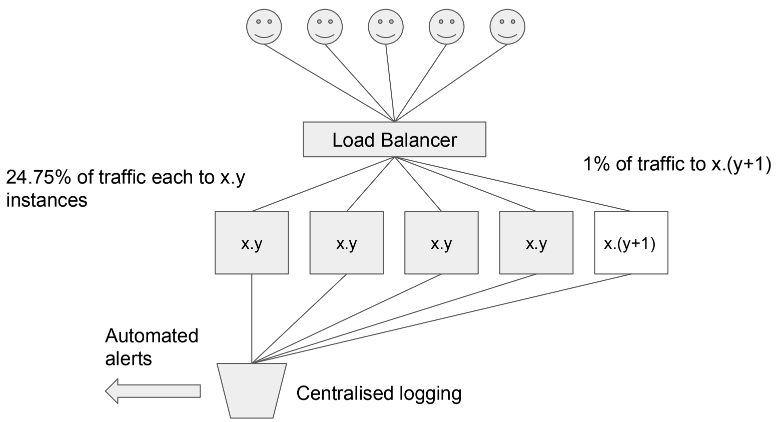


Note: these resources are usually running in a cloud platform. So virtual machines, load balancers, storage, network etc. can all be provisioned and configured through the cloud platform’s APIs.

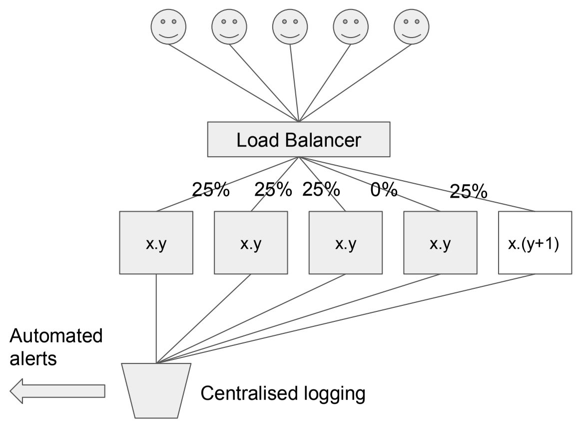
Rolling deploy: 1) Deploy ‘canary’ (limit exposure/risk)



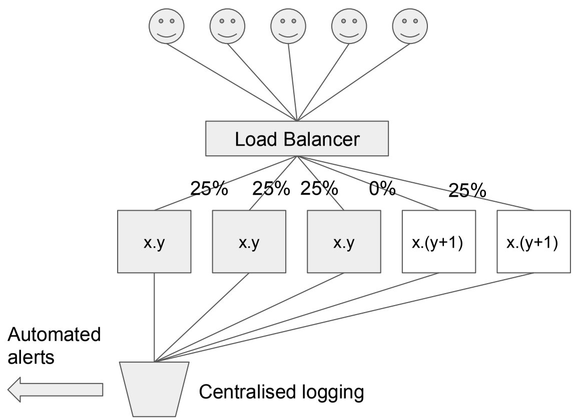
Rolling deploy: 2) Automated monitoring of error rates - OK?



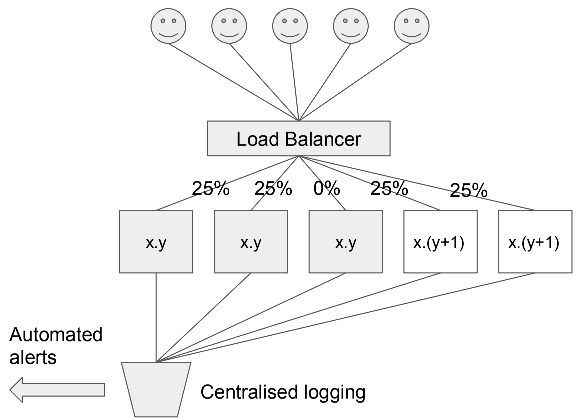
Rolling deploy: 3) Move traffic from old instance to new



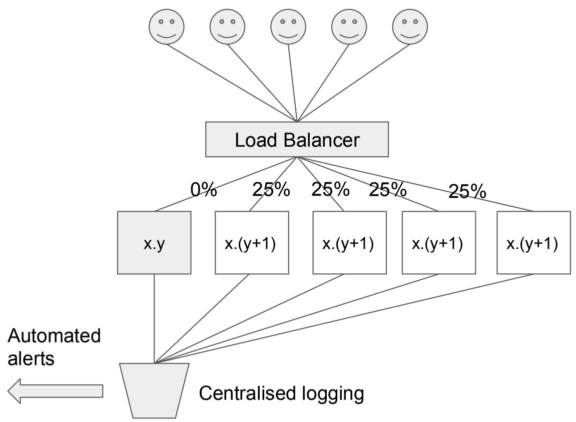
Rolling deploy: 4) Upgrade 0% instance



Rolling deploy: 5) Move traffic from old instance to new etc.

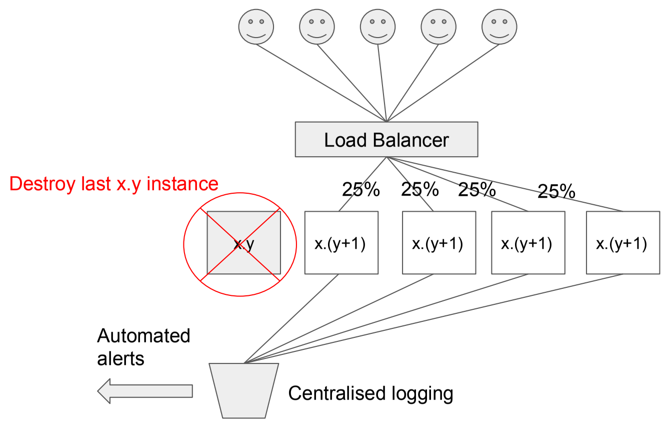


Rolling deploy: Repeat {move traffic old->new; upgrade old}



21

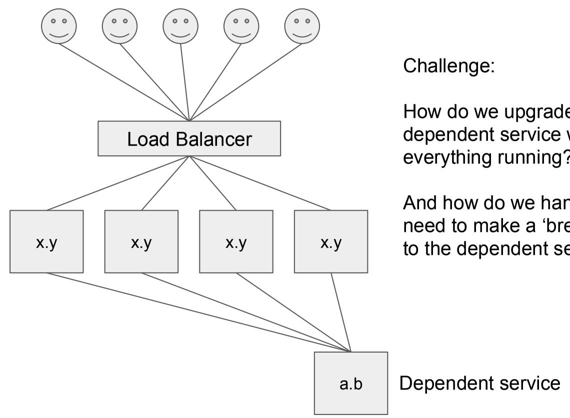
Rolling deploy: …



(If anything unexpected happens then can **pause** at any point; aim to 'roll forward' rather than 'rolling back'...)

22

Rolling deploy with service dependencies



Challenge:

How do we upgrade the dependent service while keeping everything running?

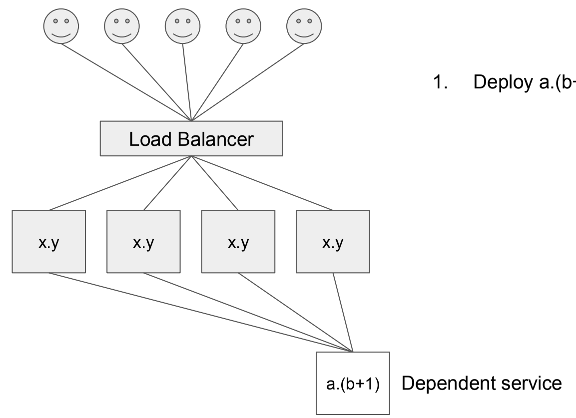
And how do we handle this if we need to make a 'breaking change' to the dependent service's API?

23

Rolling deploy with service dependencies

CONSTRAINTS:

a.(b+1) supports x.y
a.(b+1) supports x.(y+1)



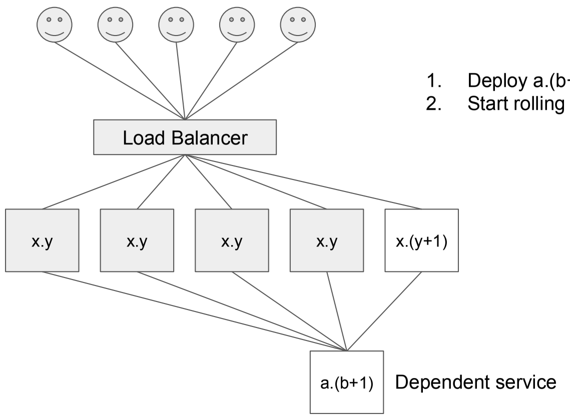
1. Deploy a.(b+1)

24

Rolling deploy with service dependencies

CONSTRAINTS:

a.(b+1) supports x.y
a.(b+1) supports x.(y+1)

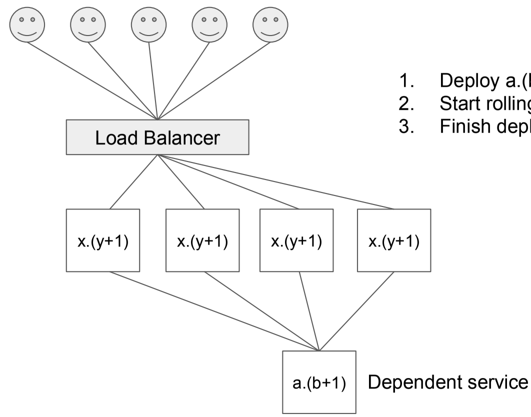


1. Deploy a.(b+1)
2. Start rolling out x.(y+1)

Rolling deploy with service dependencies

CONSTRAINTS:

a.(b+1) supports x.y
a.(b+1) supports x.(y+1)



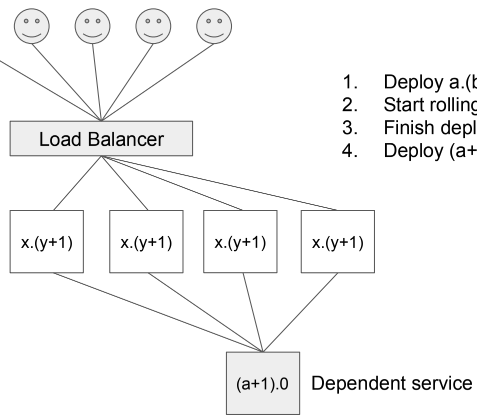
1. Deploy a.(b+1)
2. Start rolling out x.(y+1)
3. Finish deploy of x.(y+1)

Rolling deploy with service dependencies

CONSTRAINTS:

a.(b+1) supports x.y
a.(b+1) supports x.(y+1)

(a+1).0 supports x.(y+1)
[(a+1).0 doesn't have to support x.y]



1. Deploy a.(b+1)
2. Start rolling out x.(y+1)
3. Finish deploy of x.(y+1)
4. Deploy (a+1).0

Rolling deploy with service dependencies

CONSTRAINTS:

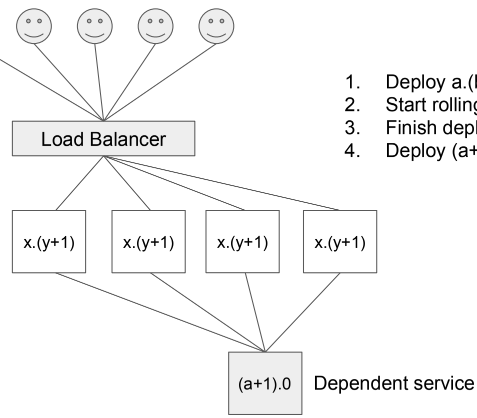
a.(b+1) supports x.y
a.(b+1) supports x.(y+1)

(a+1).0 supports x.(y+1)
[(a+1).0 doesn't have to support x.y]

We say:

a.(b+1)'s API is **backwards compatible** with a.b's API

(a+1).0's API can introduce a **breaking change**



1. Deploy a.(b+1)
2. Start rolling out x.(y+1)
3. Finish deploy of x.(y+1)
4. Deploy (a+1).0

On Automation: Infrastructure-as-Code

- Problem:
  - Manual deployments are time-consuming and error-prone. Subtle environmental differences cause bugs.
- Solution:
  - Write code to automate deployments, using Cloud APIs etc.
  - Put deployment code under version control, just like all other code
  - Have development teams write:
    - Application code
    - Code to test the application
    - **Code to deploy the application and its associated cloud infrastructure**
    - **Code to monitor the application and generate alerts**
- Frameworks like Terraform and CloudFormation help with this

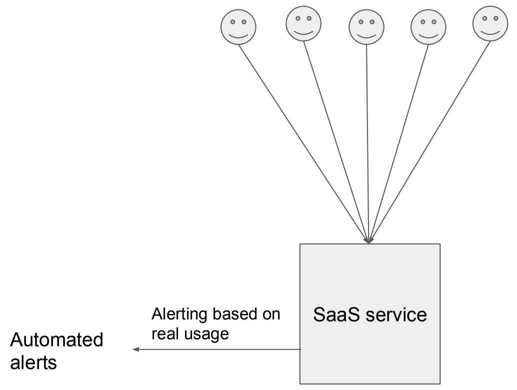
Other SaaS tools for managing quality

Rolling deploy + alerting is a very effective way of managing quality vs. big bang release.

(Insight: as long as we manage user impact, real users become an invaluable part of the QA process. NB: QA != Quality)

What other SaaS-specific tools are available for assuring quality?

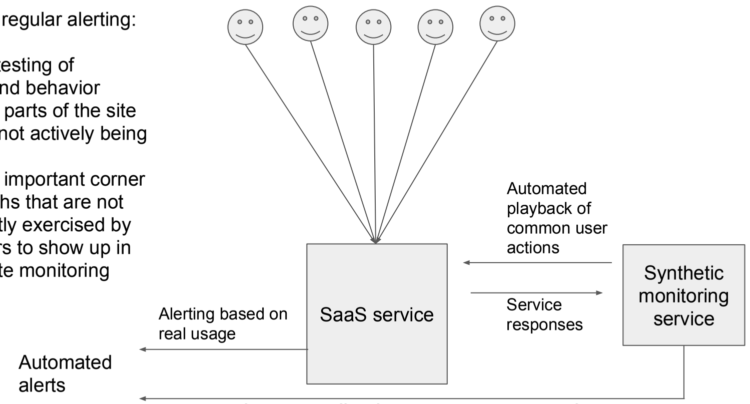
Synthetic monitoring



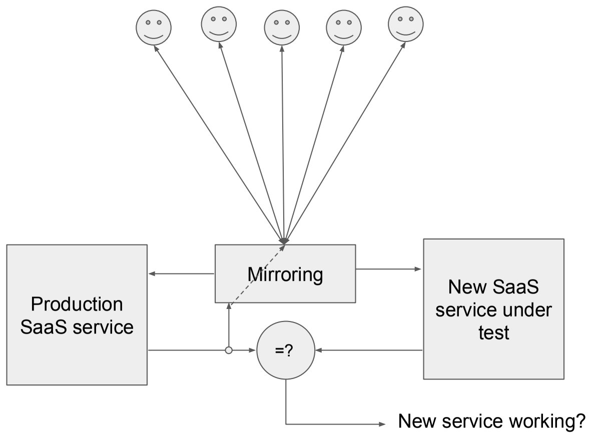
Synthetic monitoring

Complements regular alerting:

- Deeper testing of end-to-end behavior
- Can test parts of the site that are not actively being used
- Can test important corner case paths that are not sufficiently exercised by real users to show up in aggregate monitoring

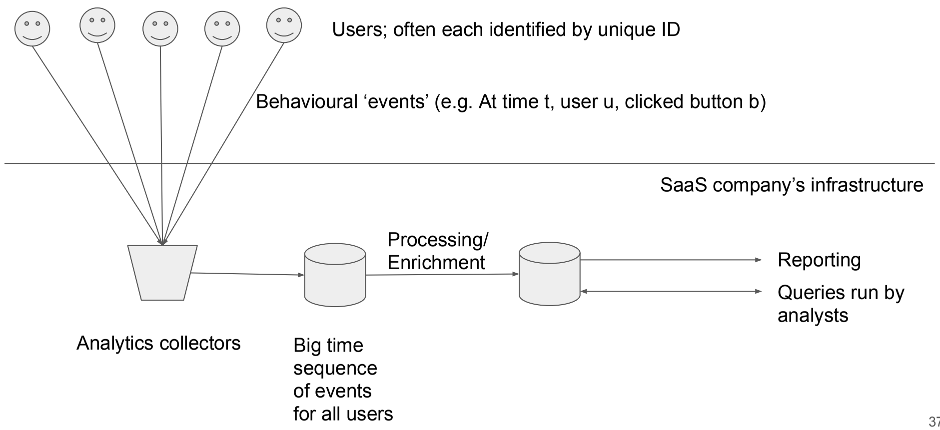


Traffic mirroring



Behavioural analytics and experiments

A simple behavioural analytics pipeline



What can we learn from the event logs?

- User/growth metrics:
  - Monthly Active Unique Users (MAU); Daily Active Unique Users (DAU)
- Engagement:
  - Time spent using the service
- Feature usage/growth/engagement metrics:
  - X% of users tried feature F at least once in the last month
  - Y% of users used feature F2 for at least 5 minutes last week
  - Feature F3 usage growing at Z% year-on-year
- Insights based on user segmentation:
  - Users who signed up in January 2018 exhibit an average 2% monthly churn
  - Female users aged between 20-25 are X% more likely to use feature F at least once

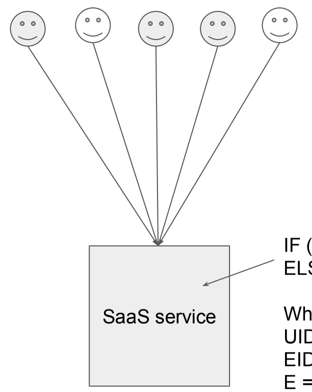
What else can we learn from the event logs?

- Correlations
  - Usage of feature F2 is correlated with usage of feature F1
  - Daily time spent on the platform is correlated with the number of days since sign-up
- But NOT cause and effect… At least not without an experiment framework.

How can we move from correlations to cause/effect?

- Run controlled experiments:
  - Determine hypothesis to test
  - Determine level of exposure, E, (% of users that will go into experiment group)
  - Bucket users into either experiment group (E%) or control group (100-E)%
  - Release a change to the experiment group only
  - Measure relevant metric(s) in both control group and experiment group and determine whether the observed **difference** is statistically significant
- By measuring difference between control and experiment groups we can have some confidence that the only meaningful difference is our 'change under test'
- Often pick low E and ramp up (e.g. 1%, 10%, 25%, 50%)
  - Similar to phased deploy alerting, but measures 'do users like it' rather than 'are there errors'
- Experiment throughput can quickly become limited by traffic volume

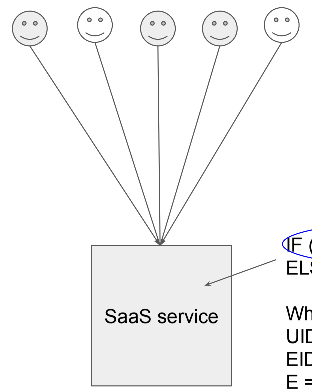
A/B test architecture



IF (hash(UID.EID) mod 100) < E then serve experiment variant
ELSE serve control variant

Where:
UID = User ID
EID = Experiment ID (one per experiment)
E = size of experiment group for experiment EID

A/B test architecture

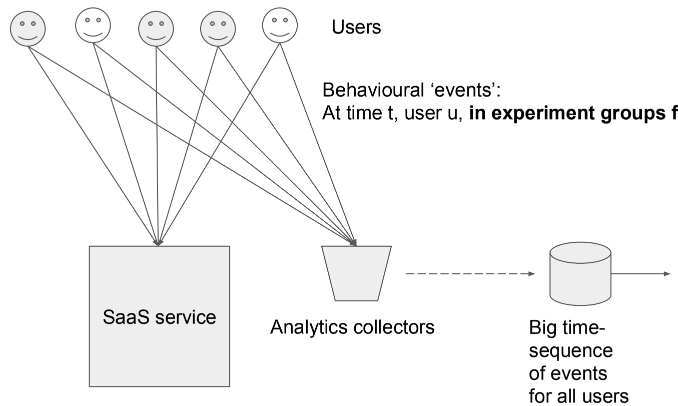


- Users *persistently* in a control or experiment group; don't 'flap'
- Users in existing experiment group remain in experiment group as E increased
- Works for multiple concurrent experiments (but be careful of independence assumptions)

IF (hash(UID.EID) mod 100) < E then serve experiment variant
ELSE serve control variant

Where:
UID = User ID
EID = Experiment ID (one per experiment)
E = size of experiment group for experiment EID

A/B test architecture



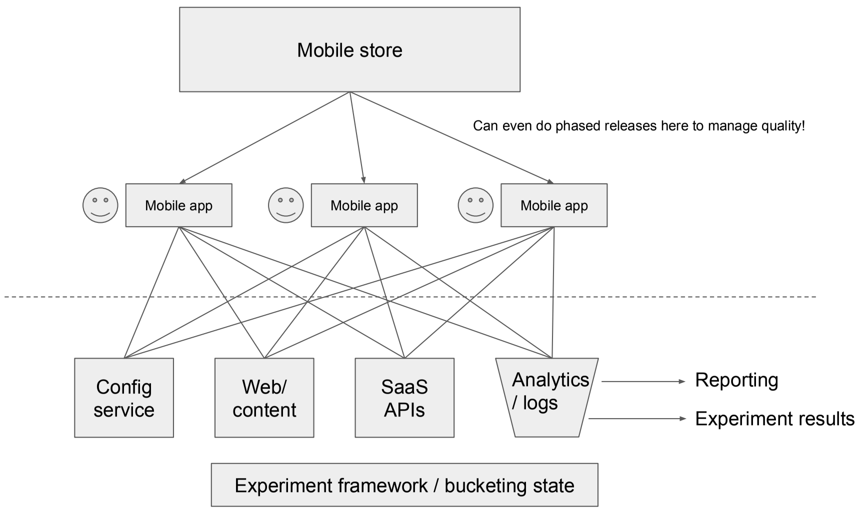
Behavioural 'events':
At time t, user u, **in experiment groups for EID1, EID5,** clicked button b

For each experiment, e, generate reports for metrics of interest segmented by (i) 'in EID_e'; and (ii) 'not in EID_e'. Compare these results for each metric and test statistical significance.

Hybrid apps/SaaS

Modern apps are often a hybrid of native, web, SaaS

- A mobile app you can download from a store ...
  - Native binaries can deliver lower latency, more controlled on-device experience
- … which accesses web services ...
  - For real-time interaction with other users, accessing live information, making payments, requesting services etc.
- … which may contain webviews
  - For flexible rendering of content, the structure of which doesn’t have to be specified within the mobile app itself



Summary

Summary

- Putting the manage/deploy/upgrade cycle to the software company is a profound change with far-reaching consequences:
  - Economically:
    - Reduces customer TCO and barriers to purchasing
    - Leads to better specialisation, and less duplication; creates new business models
  - Operationally:
    - Enables new ways of doing QA, which changes the economics of testing
    - Phased releases (which can take place over days if required, with flexibility to pause and fix at any time); live monitoring/alerting
    - Plus other techniques like traffic mirroring; synthetic monitoring
    - A continual game of chess: multiple projects, active phased releases, experiments ...
  - Enables building of higher quality software through increased visibility of user behavior. (N.B. with great power comes great responsibility!)
    - Behavioural analytics
    - Experiments

An introduction to software testing

Andrew Rice

Some problems can be detected statically

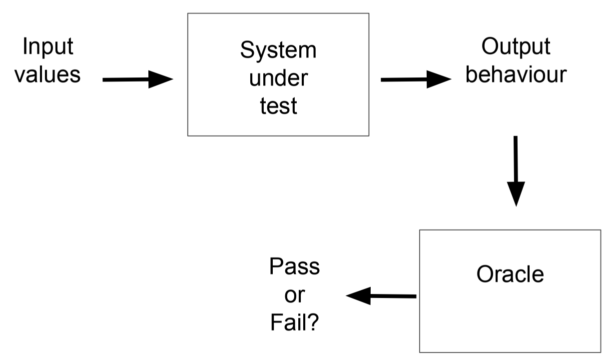
```
fun nth 0 (x::_) = x
 |  nth n (x::xs) = nth (n-1) x;
```


Many problems cannot

```
fun nth 0 (x::_) = x
 |  nth n (x::xs) = nth (n-1) xs;

var l = nth 10 [1,2,3];
```


Testing checks how software performs at run-time



Objectives

1. Identify different types of test
2. Be able to write a 'good' unit test
3. Know about some techniques for measuring test quality
4. Understand how testing fits into the software development process

Different types of test

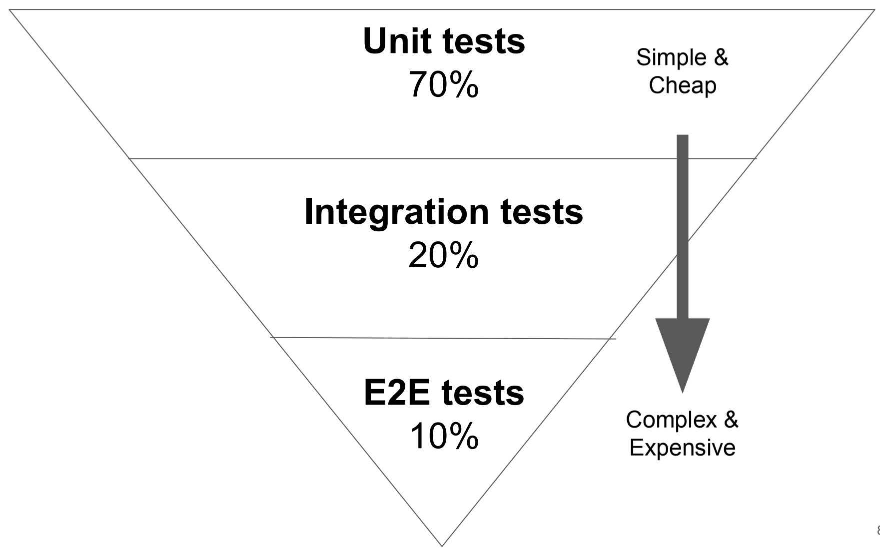
We will consider three kinds of testing

Unit tests
check isolated pieces of functionality

Integration tests
check that the parts of a system work together

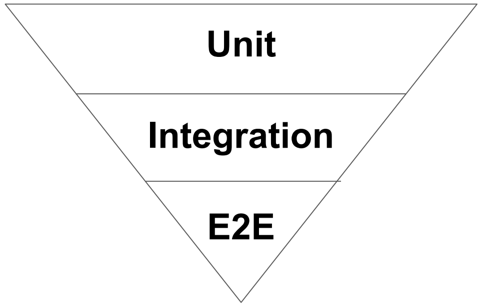
E2E (end-to-end) tests
simulate real-user scenarios

These form the 'testing pyramid'



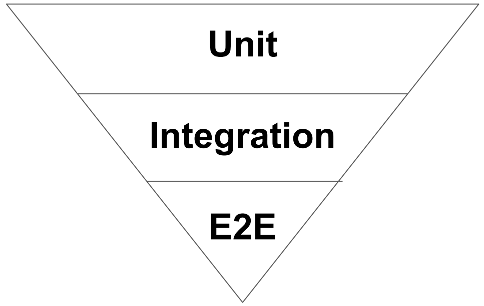
(1) What kind of test is this?

Testing whether clicking the logout button on a website clears the cookie set in the user's browser.



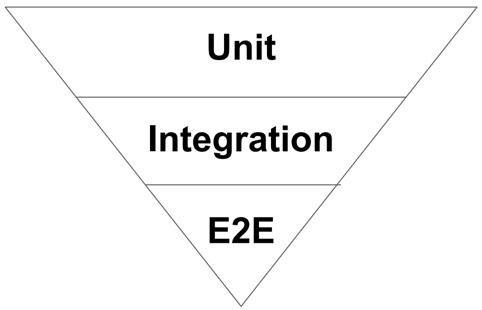
(2) What kind of test is this?

Testing that the `computeShortestPath` function returns a sensible result when there are negative edge-weights in the graph.



(3) What kind of test is this?

Testing whether the room booking system is able to query a user's calendar correctly



Unit testing demo

```
static long calculateAgeInDays(String dateOfBirth) {
    Instant dob = dateFormat.parse(dateOfBirth).toInstant();
    Instant currentTime = new Date().toInstant();
    Duration age = Duration.between(dob, currentTime);
    long ageInDays = age.toDays();
    if (ageInDays < 0) {
        return 0;
    }

    return ageInDays;
  }
}
```

Unit testing takeaway points

Design for test: dependency injection

Test naming

One property per test

Arrange, Act, Assert

Writing assertions

JUnit lifecycle

Using @Before vs constructors

Mocking can be used to simulate a dependency

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;

LinkedList mockedList = mock(LinkedList.class);

// can specify behaviour that you want
when(mockedList.get(0)).thenReturn("first");

mockedList.add("added");
// assert that things got called
verify(mockedList).add("added");
```


Integration and E2E tests are more complicated

Testing whether clicking the logout button on a website clears the cookie set in the user's browser

1. Start up a test instance of the server
2. Start a webdriver
3. Login to the site and collect the session cookie
4. Simulate a click on the logout button
5. Check the response from the server contains the directive to clear the cookie

A 'flaky' test will pass and fail on the same code

non-hermetic reliance on external systems

more complex tests tend to be more flaky

	% of tests that are flaky
All tests	1.65%
Java webdriver	10.45%
Android emulator	25.46%

https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html

Automated test generation can find unnoticed bugs

Many approaches

One example is random testing

- Generate inputs at random
- Use search to refine these inputs to make them more effective
- Check for 'bad things' like a buffer overflow
- See https://github.com/google/oss-fuzz - found thousands of security vulnerabilities in open source code

How good are my tests?

Code coverage detects how much code you execute

(Demo)

100% coverage does not mean bug-free!

```
public static void xPlusYMinusZ(double x, double y, double z) {
    double t = x + y;
    return t - z;
}

@Test
public void xPlusYMinusZ_correctlyCombines_smallNumbers() {
    double r = xPlusYMinusZ(2.0, 2.0, 2.0)
    // check floating point values with error tolerance...
    assertThat(r).isWithin(0.1).of(2.0);
}
```

This has 100% coverage but the code still has a bug...

Test coverage can use various properties

```
if (a == 0) {
    ...;
}
else {
    if (b) {
        ...;
    }
    if (c) {
        ...;
    }
}
```

Statement coverage: all lines were executed

Branch coverage: all decisions were explored at every branch

Path coverage: all paths through the program were taken

Data flow coverage: is every possible definition tested

Mutation testing can tell us how robust our tests are

Generate small changes to the program under test

- change + to a -
- change constant term
- negate a condition

Verify that this causes a test to fail

Integrating testing into your software engineering process

Defects in software are inevitable

Expect 1-25 errors per 1000 lines for delivered software

80% of errors are in 20% of the project's classes

See Steve McConnell, "Code Complete" 2nd edition, p521, p517

Defects in software are inevitable

Expect 1-25 errors per 1000 lines for delivered software

- when we find a problem we need to know we've fixed it
- once we fix a bug it needs to stay fixed

80% of errors are in 20% of the project's classes

- if we can't test everything then prioritise the error prone parts

See Steve McConnell, "Code Complete" 2nd edition, p521, p517

Continuous integration automatically runs tests

Don't want broken code committed to the repository

Run test suite on every change: can reject changes which break tests or just report

Regression testing preserves existing functionality

1. Write tests that exercise existing functionality
2. Develop new code
3. Run tests to check for regressions

Regression testing helps with bug fixing

1. Write test that reproduces bug
2. Check that it fails
3. Fix bug
4. Check that test passes

We can't run all the tests on every change

Google has 4.2 million tests and 150 million test executions every day

Need to deliver results to developers quickly

Need to manage the execution cost of running tests

See "The State of Continuous Integration Testing @Google"

Test suite minimisation

Choose a subset of tests which achieve coverage on the project

Test set selection

Choose a subset of tests which are appropriate for the change submitted

Test set prioritisation

Choose an ordering such that tests more likely to find a defect are run earlier

Example: test suite minimisation

Select a minimal subset of tests which maximise coverage over the project

NP-complete problem so use heuristics

If some test is the only test to satisfy a test requirement then it is an *essential test.*

1) Choose all the essential tests
2) Choose remaining tests greedily in order of coverage added

Test Driven Development uses tests as specification

1. Write tests which demonstrate the desired behaviour
2. Implement new functionality
3. Check tests now pass
4. Repeat

Pros: guarantees that you write tests and that your code is testable, tests can be written that directly describe the customer's requirements.

Cons: early commitment to how the project will work, changes in approach are hard, some areas are more important to test than others.

Objectives

1. Identify different types of test
2. Be able to write a 'good' unit test
3. Know about some techniques for measuring test quality
4. Understand how testing fits into the software development process

...program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence…

--- E. W. Dijkstra