UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Fabian Gilson

Software Engineering II

SENG301 - Reliability and resilience engineering

Lecture 8 - Semester 1, 2020

University of Canterbury | Te Whare Wānanga o Waitaha
Computer Science and Software Engineering

It is not a question of “if” but rather of “when”.

– adapted from Jean-Noël Colin

School Education Transformation Technology meet-up – 2020


Reliability engineering

A taxonomy

Randell, "Turing Memorial Lecture Facing Up to Faults", 2000

Humans cause faults leading to errors creating failures

human error inputting invalid data or misbehave

system fault characteristic leading to an error, *e.g.,* bug in code

system error visible effects of a misbehaviour of the system

system failure system is not able to produce expected results

Reliability improvement mechanisms

fault avoidance design and development processes, tools and guidelines

detection & correction testing, debugging and validation

fault tolerance system is designed to handle and recover from failures

A trade-off must be found between *corrections* and *consequences* cost

Availability and reliability

Those are different notions

availability probability to successfully access the system at a given time

reliability probability of failure-free operations

Works-as-designed problem

- system specifications may be **wrong**, *i.e.* not reflecting the user's truth
- system specifications may be **erroneous**, did someone proof-read them?

Reliability may be **subjective**

- concentration of errors in a **specific part** of the system
- systematic longer response time at **specific time**

→ may only affect a **subset of users**

Reliability metrics and requirements

Pretty common in safety-critical systems, but not only

POFOD probability of failure on demand, *e.g.,* services misbehave 1/1000 requests
ROCOF rate of occurrence of failures, *e.g.,* two failures per hour
AVAIL availability for servicing, *e.g.,* the magic 99.9%

Reliability requirements

functional specifying additional input checks, recovery or redundancy processes
non-funct. using above reliability metrics (or others)

Reliability requirements might not necessarily be system-wide

- don't try to reach the **99.9%** if it's **not critical**, because it's costly
- **runtime fixes** are sometimes economically preferable

Reliability metrics are also used to decide if the system is **ready for production**

Capacity management as software reliability

" Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil****. Yet we should not pass up our opportunities in that critical 3%. A* ***good programmer*** *will not be lulled into complacency by such reasoning, he* ***will be wise to look carefully at the critical code; but only after that code has been identified****. "*

Knuth, "Structured Programming with Go to Statements", 1974

Loose capacity requirements strategy

- carefully think about the **software architecture**, especially for input/output
- use **threads carefully**, *i.e.* starvation and deadlocks are bad
- write **dedicated tests** and **monitor** production system, if possible

Architectural strategies

Architectural patterns on top of common *in-code* fault detection processes

- **Protection systems**
  - → system specialised in monitoring the execution of another
  - → trigger alarms or invoke corrective programs
- **Self monitoring systems**
  - → concurrent computation
  - → mismatch detection and transfer of control
- **Multiversion programming**
  - → mainly for hardware-fault management, *e.g.*, triple-modular-redundancy

Reliability guidelines for programmers

visibility
- apply *need-to-know* principle

validitity
- check format and domain of input values, including *boundaries*
- explicitly or *regression-test-enabled* `asserts`

exceptions
- avoid errors to become system failures by *capturing* them

erring
- avoid error-prone constructs, *e.g. implicit* variables
- encapsulate *"nasty"* stuff

restart
- provide *recoverable milestones*

constants
- express *fixed* or *real-world* values with meaningful names
- self-documented code, compile-time verification

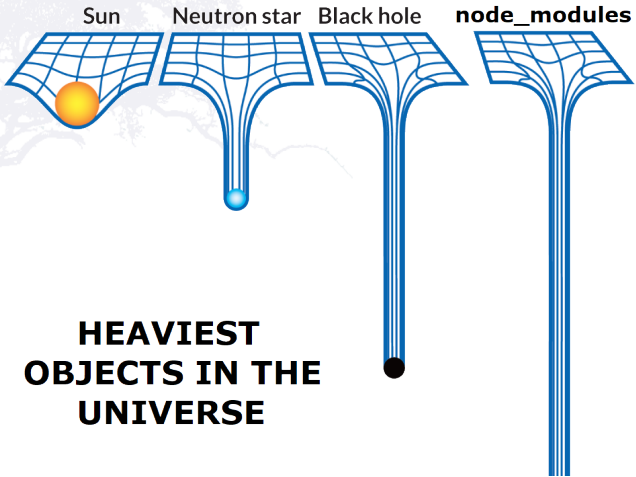
What can go wrong?

Security engineering

I just put this here...

https://twitter.com/andybelldesign/status/1105440749687066624?s=09

... and this one too...



https://i.redd.it/tfugj4n3l6ez.png

Ladies and Gentleman, the Internet

The emergence of Internet in the 1990's created new threats on

confidentiality access to information by unauthorised people or programs

integrity corruption or damage to a system or to its data

availability disallow access to a system or to its data

Three levels of security to consider

infratructure computers and network services

application (group of) application systems

operational usage of the organisation's system

Security 101 for opened servers

Think about access and credentials of direct connections

- don't open random **ports** for any sort of services, *e.g.,* telnet, ftp, sql

→ use **secured** connections, *e.g.,* redirect http traffic to 443 (SSL)

- admin - admin is probably not a good **combination**
- direct **root login** is maybe worst, **sudoers** are not recommended

Think about the services running

- services run by sudoers/root users are *sinkholes*,
- carefully define **privileges** of service users

→ monitor their behaviour, *i.e.* right-side of **DevOps** loop

Front-end code and public repositories

- do not put any sensitive data in there, *e.g.,* application secrets or API keys

Propose security recommendations

Take 10 minutes to discuss about the following and come up with security recommendations

You are responsible to maintain a set of servers with multiple virtual machines (VM).

You are deploying a first VM with a web server to run your e-commerce website (PROD).

Another VM will host your database for your website (DB).

Your DevOps engineers have remote access to both the PROD and DB VMs to deploy new versions built from a CI pipeline running from another VM developers have access to.

Your CEO wants to be able to host an organisational (private) data repository on another VM. He knows about FTP and does not want to invest in costly software.

Other employees would need to access to the same repository to browse data, but they shouldn't be able to modify the content. They are also fine with FTP clients.

do not exist

Security threads may come from

- **ignorance** don't know about a potential risk
- **design** security has been disregarded
- **carelessness** bad job from developers or system engineers
- **trade-offs** not enough emphasis on security

Three levels of protection

- **access** provide the right level of authentication mechanism
- **assets** encrypt sensitive data and separate database from software
- **network** protect and monitor network and gateways

⚠ Resources need to be classified and threats identified

The four R of a Resilience engineering plan

Recognition - Resistance - Recovery - Reinstatement

recognition how an attack may target an identified resource, *aka asset*

resistance possible strategies to resist to each thread

recovery plan data, software and hardware recovery procedures

reinstatement define the process to bring the system back

Resilience plan is a design activity by itself

- identification of resilience requirements, *e.g.* availability under attack
- **backup** and **reinstating** procedures must be specified on top of the deployment
- appropriate *classification* of critical assets and how to work in **degraded mode**
- … and all of these should **be tested** too…

Modelling threats and misbehaviours

Misuse cases or stories

- analogous to regular stories or use case scenarios, but with a *"cracker perspective"*
- depict the misuse together with **protection** or **corrective** actions
- create a **map of vulnerabilities** (classification step prior to the four R's)

→ allows to identify **critical assets**

Write **dedicated security tests**

- for your own application **software**
- for **brute-type** access to data
- for any **interactions** with involved third-parties too

The Fourth of its name

Programmer Test Principles

https://medium.com/@kentbeck_7670/programmer-test-principles-d01c064d7934

Remember to keep notes

- what is it about? was it **relevant**? what are your **takeaways**?
- how does it relate to this course and SE in general?

All readings are on Learn in the **Resources** section


TO BE
CONTINUED...

Next episode

Let's find if we're on track or not

Oh no, the PO is in the place and wants more in her plate

What is a good team culture